ARQUITECTURA DE COMPUTADORES

Un enfoque cuantitativo

John L. Hennessy – David A. Patterson

McGraw Hill

ARQUITECTURA DE COMPUTADORES

Un enfoque cuantitativo

CONSULTORES EDITORIALES
AREA DE INFORMATICA Y COMPUTACION

Antonio Vaquero Sánchez
Catedrático de Informática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid
ESPAÑA

Gerardo Quiroz Vieyra
Ingeniero en Comunicaciones y Electrónica
Escuela Superior de Ingeniería Mecánica y Eléctrica IPN
Carter Wallace, S. A.
Universidad Autónoma Metropolitana
Docente DCSA
MEXICO

ARQUITECTURA DE COMPUTADORES
Un enfoque cuantitativo

John L. Hennessy
Universidad de Stanford

David A. Patterson
Universidad de California en Berkeley

Traducción
JUAN MANUEL SANCHEZ
Universidad Complutense de Madrid

Revisión técnica
ANTONIO GONZALEZ
MATEO VALERO
Universidad Politécnica de Cataluña

ANTONIO VAQUERO
Universidad Complutense de Madrid

McGraw-Hill

MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MEXICO
NUEVA YORK • PANAMA • SAN JUAN • SANTAFE DE BOGOTA • SANTIAGO • SAO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILAN • MONTREAL • NUEVA DELHI • PARIS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

ARQUITECTURA DE COMPUTADORES. UN ENFOQUE CUANTITATIVO

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 1993 respecto a la primera edición en español por McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.

Edificio Valrealty, 1ª. planta
Basauri, 17
28023 Aravaca (Madrid)

Traducido de la primera edición en inglés de
COMPUTER ARCHITECTURE. A QUANTITATIVE APPROACH

Copyright © MLMXC, por Morgan Kaufmann Publishers, Inc.

ISBN: 1-55860-069-8

ISBN: 84-7615-912-9
Depósito legal: M. 41.104-2002

Editor: Mariano J. Norte
Cubierta: F. Piñuela. Grafismo electrónico
Compuesto en: FER Fotocomposición, S. A.
Impreso en: LAVEL, Industria Gráfica, S. A.

PRINTED IN SPAIN - IMPRESO EN ESPAÑA

A Andrea, Linda y nuestros cuatro hijos

Marcas registradas

Las siguientes marcas registradas son propiedad de las siguientes organizaciones:

Alliant es una marca registrada de Alliant Computers.

AMD 29000 es una marca registrada de AMD.

TeX es una marca registrada de American Mathematical Society.

AMI 6502 es una marca registrada de AMI.

Apple I, Apple II y Macintosh son marcas registradas de Apple Computer, Inc.

ZS-1 es una marca registrada de Astronautics.

UNIX y UNIX F77 son marcas registradas de AT&T Bell Laboratories.

Turbo C es una marca registrada de Borland International.

The Cosmic Cube es una marca registrada de California Institute of Technology.

Warp, C.mmp y Cm* son marcas registradas de Carnegie-Mellon University.

CP3100 es una marca registrada de Conner Peripherals.

CDC 6600, CDC 7600, CDC STAR-100, CYBER-180, CYBER 180/990 y CYBER-205 son marcas registradas de Control Data Corporation.

Convex, C-1, C-2 y C series son marcas registradas de Convex.

CRAY-3 es una marca registrada de Cray Computer Corporation.

CRAY-1, CRAY-1S, CRAY-2, CRAY X-MP, CRAY X-MP/416, CRAY Y-MP, CFT77 V3.0, CFT y CFT2 V1.3a son marcas registradas de Cray Research.

Cydra 5 es una marca registrada de Cydrome.

CY7C601, 7C601, 7C604 y 7C157 son marcas registradas de Cypress Semiconductor.

Nova es una marca registrada de Data General Corporation.

HEP es una marca registrada de Denelcor.

CVAX, DEC, DECsystem, DECstation, DECstation 3100, DECsystem 10/20, fort, LP11, Massbus, MicroVAX-I, MicroVAX-II, PDP-8, PDP-10, PDP-11, RS-11M/IAS, Unibus, Ultrix, Ultrix 3.0, VAX, VAXstation, VAXstation 2000, VAXstation 3100, VAX-11, VAX-11/780, VAX-11/785, VAX Model 730, Model 750, Model 780, VAX 8600, VAX 8700, VAX 8800, VS FORTRAN V2.4 y VMS son marcas registradas de Digital Equipment Corporation.

BINAC es una marca registrada de Eckert-Mauchly Computer Corporation.

Multimax es una marca registrada de Encore Computers.

ETA 10 es una marca registrada de ETA Corporation.

SYMBOL es una marca registrada de Fairchild Corporation.

Pegasus es una marca registrada de Ferranti, Ltd.

Ferrari y Testarossa son marcas registradas de Ferrari Motors.

AP-120B es una marca registrada de Floating Point Systems.

Ford y Escort son marcas registradas de Ford Motor Co.

Gnu C Compiler es una marca registrada de Free Software Foundation.

M2361A, Super Eagle, VP100 y VP200 son marcas registradas de Fujitsu Corporation.

Chevrolet y Corvette son marcas registradas de General Motors Corporation.

HP Precision Architecture, HP 850, HP 3000, HP 3000/70, Apollo DN 300, Apollo DN 10000 y Precision son marcas registradas de Hewlett-Packard Company.

S810, S810/200 y S820 son marcas registradas de Hitachi Corporation.

Hyundai y Excel son marcas registradas de Hyundai Corporation.

432, 960 CA, 4004, 8008, 8080, 8086, 8087, 8088, 80186, 80286, 80386, 80486, iAPX 432, i860, Intel, Multibus, Multibus II e Intel Hypercube son marcas registradas de Intel Corporation.

Inmos y Transputer son marcas registradas de Inmos.

Clipper C100 es una marca registrada de Intergraph.

IBM, 360, 360/30, 360/40, 360/50, 360/65, 360/85, 360/91, 370, 370/135, 370/138, 370/145, 370/155, 370/158, 370/165, 370/168, 370-XA, ESA/370, System/360, System/370, 701, 704, 709, 801, 3033, 3080, 3080 series, 3080 VF, 3081, 3090, 3090/100, 3090/200, 3090/400, 3090/600, 3090/600S, 3090 VF, 3330, 3380, 3380D, 3380 Disk Model AK4, 3380J, 3390, 3880-23, 3990, 7030, 7090, 7094, IBM FORTRAN, ISAM, MVS, IBM PC, IBM PC-AT, PL.8, RT-PC, SAGE, Stretch, IBM SVS, Vector Facility y VM son marcas registradas de International Business Machines Corporation.

FutureBus es una marca registrada de Institute of Electrical and Electronic Engineers.

Lamborghini y Countach son marcas registradas de Nuova Automobili Ferrucio Lamborghini, SPA.

Lotus 1-2-3 es una marca registrada de Lotus Development Corporation.

MB8909 es una marca registrada de LSI Logic.

NuBus es una marca registrada de Massachusetts Institute of Technology.

Miata y Mazda son marcas registradas de Mazda.

MASM, Microsoft Macro Assembler, MS DOS, MS DOS 3.1 y OS/2 son marcas registradas de Microsoft Corporation.

MIPS, MIPS 120, MIPS/120A, M/500, M/1000, RC6230, RC6280, R2000, R2000A, R2010, R3000 y R3010 son marcas registradas de MIPS Computer Systems.

Delta Series 8608, System V/88 R32V1, VME bus, 6809, 68000, 68010, 68020, 68030, 68882, 88000, 88000 1.8.4m14, 88100 y 88200 son marcas registradas de Motorola Corporation.

Multiflow es una marca registrada de Multiflow Corporation.

National 32032 y 32x32 son marcas registradas de National Semiconductor Corporation.

Ncube es una marca registrada de Ncube Corporation.

SX/2, SX/3 y FORTRAN 77/SX V.040 son marcas registradas de NEC Information Systems.

NYU Ultracomputer es una marca registrada de New York University.

VAST-2 v.2.21 es una marca registrada de Pacific Sierra.

Wren IV, Imprimis, Sabre 97209 e IPI-2 son marcas registradas de Seagate Corporation.

Sequent, Balance 800, Balance 21000 y Symmetry son marcas registradas de Sequent Computers.

Silicon Graphics 4D/60, 4D/240 y Silicon Graphics 4D Series son marcas registradas de Silicon Graphics.

Stellar GS 1000, Stardent-1500 y Ardent Titan-I son marcas registradas de Stardent.

Sun 2, Sun 3, Sun 3/75, Sun 3/260, Sun 3/280, Sun 4, Sun 4/110, Sun 4/260, Sun 4/280, SunOS 4.0.3c, Sun 1.2 FORTRAN compiler, SPARC y SPARCstation 1 son marcas registradas de Sun Microsystems.

Synapse N+ es una marca registrada de Synapse.

Tandem y Cyclone son marcas registradas de Tandem Computers.

TI 8847 y TI ASC son marcas registradas de Texas Instruments Corporation.

Connection Machine y CM-2 son marcas registradas de Thinking Machines.

Burroughs 6500, B5000, B5500, D-machine, UNIVAC, UNIVAC I y UNIVAC 1103 son marcas registradas de UNISYS.

Spice y 4.2 BSD UNIX son marcas registradas de University of California, Berkeley.

Illiac, Illiac IV y Cedar son marcas registradas de University of Illinois.

Ada es una marca registrada de U.S. Government (Ada Joint Program Office).

Weitek 3364, Weitek 1167, WTL 3110 y WTL 3170 son marcas registradas de Weitek Computers.

Alto, Ethernet, PARC, Palo Alto Research Center, Smalltalk y Xerox son marcas registradas de Xerox Corporation.

Z-80 es una marca registrada de Zilog.

Prefacio

por C. Gordon Bell

Estoy encantado y satisfecho de escribir el prefacio de este libro decisivo.

Los autores han ido más allá de las contribuciones de Thomas al Cálculo y de Samuelson a la Economía. Han proporcionado el texto y referencia definitivos para el *diseño* y arquitectura de computadores. Para avanzar en computación, recomiendo encarecidamente a los editores que retiren los montones de libros sobre este tópico de manera que, rápidamente, pueda emerger una nueva raza de arquitecto/ingeniero. Este libro no eliminará los microprocesadores complejos y llenos de errores de las compañías de semicomputadores, pero acelerará la educación de ingenieros que podrán diseñarlos mejor.

El libro presenta las herramientas críticas para analizar los computadores uniprocesadores. Muestra al ingeniero en ejercicio cómo cambia la tecnología a lo largo del tiempo y ofrece las constantes empíricas que se necesitan para el diseño. Motiva al diseñador sobre la función, que es una orientación más agradable que la habitual lista exhaustiva de los mecanismos que un diseñador novel puede intentar incluir en un único diseño.

Los autores establecen un punto de partida para realizar análisis y comparaciones utilizando la máquina más importante de cada clase: computadores grandes (IBM 360), mini (DEC VAX), y micro PC (Intel 80×86). Con esta base, muestran la línea futura de procesadores paralelos segmentados más simples. Estas nuevas tecnologías se muestran como variantes de su procesador (DLX) útil pedagógicamente, pero altamente realizable. Los autores hacen énfasis en la independencia de la tecnología al medir el trabajo realizado por ciclo de reloj (paralelismo), y el tiempo para hacer el trabajo (eficiencia y latencia). Estos métodos también mejorarán la calidad de la investigación sobre nuevas arquitecturas y paralelismo.

Por ello, el libro es necesario que lo conozca cualquiera que trabaje en arquitectura o hardware, incluyendo arquitectos, ingenieros de sistemas de computadores y de circuitos integrados, e ingenieros de sistemas operativos y compiladores. Es útil especialmente para los ingenieros de software que escriben programas para computadores vectoriales y segmentados. Los empresarios y vendedores se beneficiarán al conocer las secciones de Falacias y Pifias del libro. El fracaso de muchos computadores —y ocasionalmente, de alguna compañía— se puede atribuir a ingenieros que fallan al comprender las sutilezas del diseño de computadores.

Los dos primeros capítulos establecen la esencia del diseño de computadores a través de medidas y de la comprensión de la relación precio/rendi-

miento. Estos conceptos se aplican a la arquitectura a nivel de lenguaje máquina y a la forma de medirla. Explican la implementación de procesadores e incluyen amplias discusiones sobre técnicas para diseñar procesadores vectoriales y segmentados. También hay capítulos dedicados a la jerarquía de memoria y a las, con frecuencia, despreciadas entradas/salidas. El capítulo final presenta oportunidades y preguntas sobre máquinas y directrices futuras. Ahora, necesitamos su siguiente libro sobre cómo construir estas máquinas.

La razón por la que este libro constituye un estándar sobre todos los anteriores y es improbable que sea sustituido en un futuro previsible es la comprensión, experiencia, gusto y *unicidad* de los autores. Han estimulado el cambio principal en arquitectura por su trabajo sobre los RISC (Patterson acuñó la palabra). Su investigación universitaria que llevó al desarrollo de productos en MIPS y Sun Microsystems; estableció importantes arquitecturas para los años 90. Así, han hecho análisis, evaluado compromisos, trabajado sobre compiladores y sistemas operativos, y visto que sus máquinas consiguen importancia con el uso. Además, como maestros, han visto que el libro es pedagógicamente sólido (y han solicitado opiniones de los demás a través del programa de examen sin precedentes Beta). Estoy convencido que éste será el libro de la década en sistemas de computadores. Quizás su mayor logro sea estimular a otros grandes arquitectos y diseñadores de sistemas de alto nivel (bases de datos, sistemas de comunicaciones, lenguajes y sistemas operativos) a escribir libros similares sobre sus dominios.

Yo ya he aprendido y me he divertido con el libro, y usted, seguramente, también.

C. Gordon Bell

Contenido

Prefacio vii
por G. GORDON BELL

Prólogo xv

Agradecimientos xxi

Sobre los autores xxvi

1 Fundamentos del diseño de computadores 3

1.1 Introducción 3
1.2 Definiciones de rendimiento 5
1.3 Principios cuantitativos del diseño de computadores 8
1.4 El trabajo de un diseñador de computadores 13
1.5 Juntando todo: el concepto de *jerarquía de memoria* 19
1.6 Falacias y pifias 22
1.7 Observaciones finales 23
1.8 Perspectiva histórica y referencias 24
Ejercicios 30

2 Rendimientos y coste 35

2.1 Introducción 35
2.2 Rendimiento 37
2.3 Coste 57
2.4 Juntando todo: precio/rendimiento de tres máquinas 71
2.5 Falacias y pifias 74
2.6 Observaciones finales 81
2.7 Perspectiva histórica y referencias 82
Ejercicios 86

3 Diseño de repertorios de instrucciones: alternativas y principios 95

3.1 Introducción 95
3.2 Clasificación de las arquitecturas a nivel lenguaje máquina 96
3.3 Almacenamiento de operandos en memoria: clasificación de las máquinas de registros de propósito general 98
3.4 Direccionamiento de memoria 101
3.5 Operaciones del repertorio de instrucciones 110
3.6 Tipo y tamaño de los operandos 117
3.7 El papel de los lenguajes de alto nivel y compiladores 118

3.8 Juntando todo: cómo los programas utilizan los repertorios de instrucciones 131
3.9 Falacias y pifias 133
3.10 Observaciones finales 136
3.11 Perspectiva histórica y referencias 136
Ejercicios 142

4 Ejemplos y medidas de utilización de los repertorios de instrucciones 149

4.1 Medidas de los repertorios de instrucciones: qué y por qué 149
4.2 La arquitectura VAX 152
4.3 La arquitectura 360/370 159
4.4 La arquitectura 8086 164
4.5 La arquitectura DLX 172
4.6 Juntando todo: medidas de utilización del repertorio de instrucciones 180
4.7 Falacias y pifias 196
4.8 Observaciones finales 198
4.9 Perspectiva histórica y referencias 200
Ejercicios 205

5 Técnicas básicas de implementación de procesadores 213

5.1 Introducción 213
5.2 Camino de datos del procesador 215
5.3 Pasos básicos de ejecución 216
5.4 Control cableado 218
5.5 Control microprogramado 222
5.6 Interrupciones y otros enredos 229
5.7 Juntando todo: control para DLX 236
5.8 Falacias y pifias 255
5.9 Observaciones finales 257
5.10 Perspectiva histórica y referencias 258
Ejercicios 262

6 Segmentación 269

6.1 ¿Qué es la segmentación? 269
6.2 Segmentación básica para DLX 270
6.3 Haciendo que funcione la segmentación 273
6.4 El principal obstáculo de la segmentación: riesgos de la segmentación 276
6.5 Qué hace difícil de implementar la segmentación 299
6.6 Extensión de la segmentación de DLX para manipular operaciones multiciclo 305
6.7 Segmentación avanzada: planificación dinámica de la segmentación 312
6.8 Segmentación avanzada: aprovechando más el paralelismo de nivel de instrucción 337
6.9 Juntando todo: un VAX segmentado 352
6.10 Falacias y pifias 359
6.11 Observaciones finales 362

6.12 Perspectiva histórica y referencias 363
Ejercicios 368

7 Procesadores vectoriales 377

7.1 ¿Por qué máquinas vectoriales? 377
7.2 Arquitectura vectorial básica 379
7.3 Dos aspectos del mundo real: longitud del vector y separación entre elementos 391
7.4 Un modelo sencillo para el rendimiento vectorial 396
7.5 Tecnología de compiladores para máquinas vectoriales 399
7.6 Mejorando el rendimiento vectorial 405
7.7 Juntando todo: evaluación del rendimiento de los procesadores vectoriales 412
7.8 Falacias y pifias 419
7.9 Observaciones finales 421
7.10 Perspectiva histórica y referencias 422
Ejercicios 426

8 Diseño de la jerarquía de memoria 433

8.1 Introducción: principio de localidad 433
8.2 Principios generales de jerarquía de memoria 434
8.3 Caches 438
8.4 Memoria principal 458
8.5 Memoria virtual 466
8.6 Protección y ejemplos de memoria virtual 473
8.7 Más optimizaciones basadas en el comportamiento de los programas 484
8.8 Tópicos avanzados. Mejora del rendimiento de memoria cache 490
8.9 Juntando todo: la jerarquía de memoria del VAX-11/780 512
8.10 Falacias y pifias 518
8.11 Observaciones finales 522
8.12 Perspectiva histórica y referencias 522
Ejercicios 528

9 Entradas/Salidas 537

9.1 Introducción 537
9.2 Predicción del rendimiento del sistema 539
9.3 Medidas de rendimiento de E/S 545
9.4 Tipos de dispositivos de E/S 551
9.5 Buses. Conectando dispositivos de E/S a CPU/memoria 569
9.6 Interfaz con la CPU 574
9.7 Interfaz con un sistema operativo 577
9.8 Diseño de un sistema de E/S 581
9.9 Juntando todo: el subsistema de almacenamiento IBM 3990 589
9.10 Falacias y pifias 597
9.11 Observaciones finales 603
9.12 Perspectiva histórica y referencias 603
Ejercicios 607

10 **Tendencias futuras** **615**

10.1 Introducción 615
10.2 Clasificación de Flynn de los computadores 616
10.3 Computadores SIMD. Flujo único de instrucciones, flujos múltiples de datos 616
10.4 Computadores MIMD. Flujos múltiples de instrucciones, flujos múltiples de datos 618
10.5 Las rutas a El Dorado 621
10.6 Procesadores de propósito especial 624
10.7 Direcciones futuras para los compiladores 625
10.8 Juntando todo: el multiprocesador Sequent Symmetry 627
10.9 Falacias y pifias 630
10.10 Observaciones finales. Evolución frente a revolución en arquitectura de computadores 632
10.11 Perspectiva histórica y referencias 633
Ejercicios 637

Apéndice A: Aritmética de computadores **641**
por DAVID GOLDBERG
Xerox Palo Alto Research Center

A.1 Introducción 641
A.2 Técnicas básicas de la aritmética entera 642
A.3 Punto flotante 652
A.4 Suma en punto flotante 657
A.5 Multiplicación en punto flotante 662
A.6 División y resto 664
A.7 Precisiones y tratamiento de excepciones 670
A.8 Aceleración de la suma entera 673
A.9 Aceleración de la multiplicacion y división enteras 682
A.10 Juntando todo 696
A.11 Falacias y pifias 700
A.12 Perspectiva histórica y referencias 701
Ejercicios 706

Apéndice B: Tablas completas de repertorios de instrucciones **711**

B.1 Repertorio de instrucciones de usuario del VAX 712
B.2 Repertorio de instrucciones del sistema/360 717
B.3 Repertorio de instrucciones del 8086 721

Apéndice C: Medidas detalladas del repertorio de instrucciones **727**

C.1 Medidas detalladas del VAX 728
C.2 Medidas detalladas del 360 729
C.3 Medidas detalladas del Intel 8086 731
C.4 Medidas detalladas del repertorio de instrucciones del DLX 732

Apéndice D: Medidas de tiempo frente a frecuencia **733**

D.1 Distribución de tiempos en el VAX-11/780 734

D.2 Distribución de tiempos en el IBM 370/168 734
D.3 Distribución de tiempos en un 8086 de un IBM PC 738
D.4 Distribución de tiempos en un procesador próximo a DLX 739

Apéndice E: Visión general de las arquitecturas RISC **743**

E.1 Introducción 743
E.2 Modos de direccionamiento y formatos de instrucción 744
E.3 Instrucciones: el subconjunto de DLX 746
E.4 Instrucciones: extensiones comunes a DLX 753
E.5 Instrucciones únicas del MIPS 756
E.6 Instrucciones únicas del SPARC 758
E.7 Instrucciones únicas del M88000 761
E.8 Instrucciones únicas del i860 763
E.9 Observaciones finales 767
E.10 Referencias 768

Definiciones de arquitectura de computadores, trivialidades, fórmulas y reglas empíricas 769
Notación de descripción hardware (y algunos operadores C estándares) 772
Estructura de la segmentación del DLX 773
Repertorio de instrucciones del DLX, lenguaje de descripción y segmentación del DLX 773

Referencias **775**

Indice **789**

Prólogo

Comencé en 1962 a escribir un sencillo libro con esta secuencia de capítulos, pero pronto me di cuenta que era más importante tratar en profundidad los temas mejor que pasar casi tocándolos ligeramente. El tamaño resultante ha hecho que cada capítulo, por sí mismo, contenga suficiente material para un curso semestral; por ello se ha hecho necesario publicar la serie en volúmenes separados...

Donald Knuth, *The Art of Computer Programming.*
Prólogo al Volumen I (de 7) (1968)

¿Por qué escribimos este libro?

¡Bienvenidos a este libro! ¡Estamos contentos de tener la oportunidad de comunicar con usted! Hay muchas cosas excitantes que ocurren en la arquitectura de computadores, pero somos conscientes que los materiales disponibles no hacen que la gente se dé cuenta de esto. Esta no es una ciencia aburrida de máquinas de papel que nunca funcionan. ¡No! Es una disciplina de gran interés intelectual, que requiere equilibrar las fuerzas del mercado y del coste/rendimiento, llevando a fallos gloriosos y a algunos éxitos notables. Y es difícil imaginar la excitación de ver a miles de personas utilizar la máquina que usted diseñó.

Nuestro objetivo principal al escribir este libro es ayudar a cambiar la forma en que se aprende la arquitectura de computadores. Pensamos que el campo ha cambiado desde que sólo se podía enseñar con definiciones e información histórica, hasta ahora que se puede estudiar con ejemplos reales y medidas reales. Pensamos que este libro es aconsejable para un curso sobre arquitectura de computadores, así como un libro de texto o referencia para ingenieros profesionales y arquitectos de computadores. Este libro incorpora una nueva aproximación para desmitificar la arquitectura de computadores —hace énfasis en una aproximación cuantitativa de las relaciones de coste/rendimiento. Esto no implica una aproximación demasiado formal, sino simplemente la que se basa en buen diseño de ingeniería. Para lograr esto, hemos incluido muchos datos sobre máquinas reales, para que el lector pueda comprender la dinámica de un diseño, de forma tanto cuantitativa como cualitativa. Un com-

ponente significativo de esta aproximación se puede encontrar en las colecciones de problemas al final de cada capítulo. Los ejercicios han constituido el núcleo de la educación en ciencia e ingeniería. Con la aparición de una base cuantitativa para enseñar arquitectura de computadores, nos damos cuenta que el campo tiene potencial para desplazarse hacia los fundamentos rigurosos cuantitativos de otras disciplinas.

Organización y selección de tópicos

Hemos adoptado una aproximación conservadora para la selección de tópicos, ya que hay muchas ideas interesantes en este campo. Mejor que intentar una visión general comprensiva de cada arquitectura que un lector pueda encontrar en la práctica o en la literatura actual, hemos seleccionado los conceptos fundamentales de arquitectura de computadores que, probablemente, se van a incluir en cualquier máquina nueva. Para tomar estas decisiones, un criterio clave ha sido hacer énfasis en ideas que se han examinado suficientemente como para ser explicadas en términos cuantitativos. Por ejemplo, nos concentramos en los uniprocesadores hasta el capítulo final, donde se describe un multiprocesador de memoria compartida orientado a bus. Pensamos que esta clase de arquitectura de computador aumentará en popularidad, pero a pesar de esta percepción, sólo respondía a nuestro criterio en un estrecho margen. Sólo recientemente se ha examinado esta clase de arquitectura en formas que nos permiten discutirla cuantitativamente; incluso hace poco tiempo todavía esto no se podía haber incluido. Aunque los procesadores paralelos en gran escala son de importancia obvia para el futuro, creemos que es necesaria una base firme en los principios del diseño de uniprocesadores, antes que cualquier ingeniero en ejercicio intente construir un computador mejor de cualquier organización; especialmente, incorporando múltiples uniprocesadores.

Los lectores familiarizados con nuestra investigación pueden esperar que este libro trate sólo sobre computadores de repertorio de instrucciones reducido (RISC). Esto es un juicio erróneo sobre el contenido del libro. Nuestra esperanza es que los principios de diseño y los datos cuantitativos de este libro restrinjan las discusiones sobre estilos de arquitectura a términos como «más rápido» o «más barato», de forma distinta a debates anteriores.

El material que hemos seleccionado se ha ampliado hasta una estructura consistente que se sigue en cada capítulo. Después de explicar las ideas de un capítulo, incluimos una sección de «Juntando todo», que liga las ideas expuestas para mostrar cómo se utilizan en una máquina real. A continuación, sigue una sección, titulada «Falacias y pifias», que permite que los lectores aprendan de los errores de los demás. Mostramos ejemplos de errores comunes y pifias en arquitectura que son difíciles de evitar aun cuando el lector sepa que le están esperando. Cada capítulo finaliza con una sección de «Observaciones finales», seguida por una sección de «Perspectiva histórica y referencias» que intenta dar crédito adecuado a las ideas del capítulo y un sentido de la historia circundante a las invenciones, presentando el drama humano del diseño de computadores. También proporciona referencias que el estudiante de arquitectura puede desear conseguir. Si tiene tiempo, le recomen-

damos leer algunos de los artículos clásicos en este campo, que se mencionan en estas secciones. Es divertido y educativo oír las ideas de boca de los creadores. Cada capítulo finaliza con Ejercicios, unos doscientos en total, que varían desde revisiones de un minuto a proyectos de un trimestre.

Un vistazo a la Tabla de contenidos muestra que ni la cantidad ni la profundidad del material es igual de un capítulo a otro. En los primeros capítulos, por ejemplo, tenemos material básico para asegurar una terminología y conocimientos comunes básicos. Al hablar con nuestros colegas, encontramos opiniones que varían ampliamente sobre los conocimientos básicos que tienen los lectores, el ritmo al que pueden captar el nuevo material, e incluso el orden en que se deben introducir las ideas. Nuestra hipótesis es que el lector está familiarizado con el diseño lógico, y tiene conocimiento, al menos, de un repertorio de instrucciones y conceptos básicos de software. El ritmo varía con los capítulos, siendo la primera mitad más pausada que la segunda. Las decisiones sobre la organización se tomaron en respuesta a las advertencias de los revisores. La organización final se seleccionó para seguir adecuadamente la mayoría de los cursos (¡además de Berkeley y Stanford!) con sólo pequeñas modificaciones. Dependiendo de sus objetivos, vemos tres caminos a través de este material:

Cobertura introductoria:	Capítulos 1, 2, 3, 4, 5, 6.1-6.5, 8.1-8.5, 9.1-9.5, 10, y A.1-A.3.
Cobertura intermedia:	Capítulos 1, 2, 3, 4, 5, 6.1-6.6, 6.9-6.12, 8.1-8.7, 8.9-8.12, 9, 10, A (excepto la división en la Sección A.9) y E.
Cobertura avanzada:	Leer todo, pero los Capítulos 3 y 5 y las Secciones A.1-A.2 y 9.3-9.4 pueden ser en su mayor parte una revisión, así que léalas rápidamente.

Desgraciadamente, no hay un orden ideal para los capítulos. Sería agradable conocer algo sobre segmentación (Cap. 6) antes de explicar repertorios de instrucciones (Caps. 3 y 4), por ejemplo, pero es difícil comprender la segmentación sin comprender el repertorio completo de instrucciones que se está segmentando. En versiones anteriores hemos intentado diversas ordenaciones de este material, y cada una tenía sus ventajas. Por ello, el material se ha escrito para que se pueda cubrir de varias formas. La organización ha probado ser suficientemente flexible para una gran variedad de secuencias de capítulos en el programa de examen «Beta» en 18 escuelas, donde el libro se utilizó con éxito. La única restricción es que algunos capítulos deben ser leídos en secuencia:

Capítulos 1 y 2

Capítulos 3 y 4

Capítulos 5, 6 y 7

Capítulos 8 y 9

Los lectores deberían comenzar con los Capítulos 1 y 2 y finalizar con el Capítulo 10, pero el resto se puede cubrir en cualquier orden. La única salvedad es que si se leen los Capítulos 5, 6 y 7 antes que los Capítulos 3 y 4, se debería examinar primero la Sección 4.5, ya que el repertorio de instrucciones de esta sección, DLX, se utiliza para ilustrar las ideas que aparecen en esos tres capítulos. Una descripción compacta de DLX y de la notación de descripción hardware que utilizamos puede encontrarse en la contraportada posterior. (Seleccionamos una versión modificada de C para nuestro lenguaje de descripción hardware debido a: su compacidad, al número de personas que conoce el lenguaje y a que no hay lenguaje de descripción común utilizado en libros que se puedan considerar prerrequisitos.)

Recomendamos encarecidamente a todos la lectura de los Capítulos 1 y 2. El Capítulo 1 es intencionadamente fácil de seguir, ya que puede leerse rápidamente, incluso por un principiante. Da algunos principios importantes que actúan como temas que guían la lectura de capítulos posteriores. Aunque pocos se saltarían la sección de rendimiento del Capítulo 2, algunos pueden estar tentados de saltar la sección de coste para ir a las «cuestiones técnicas» de los capítulos finales. Por favor, no lo haga. El diseño de computadores es casi siempre un intento de equilibrar coste y rendimiento, y pocos comprenden cómo el precio se relaciona con el coste, o cómo bajar en un 10 por 100 coste y precio, y de qué forma se minimiza la pérdida de rendimiento. Los fundamentos que se proporcionan, en la sección de coste del Capítulo 2, permiten que el coste/rendimiento sea la base de todos los compromisos en la segunda mitad del libro. Por otro lado, probablemente sea mejor dejar algunas cuestiones como material de referencia. Si el libro es parte de un curso, las clases pueden mostrar cómo utilizar los datos desde estos capítulos para tomar decisiones en el diseño de computadores. El Capítulo 4 quizá sea el mejor ejemplo de esto. Dependiendo de los conocimientos básicos, el lector puede estar ya familiarizado con parte del material, pero tratamos de incluir algunas peculiaridades para cada materia. La sección sobre microprogramación del Capítulo 5 será un repaso para muchos, por ejemplo, pero la descripción del impacto de las interrupciones sobre el control raramente se encuentra en otros libros.

También realizamos un esfuerzo especial en hacer este libro interesante para los ingenieros en ejercicio y estudiantes avanzados. Secciones de temas avanzados se encuentran en:

Capítulo 6, sobre segmentación (Secciones 6.7 y 6.8, que son aproximadamente la mitad del capítulo).

Capítulo 7, sobre vectores (el capítulo completo).

Capítulo 8, sobre diseño de jerarquías de memoria (Sección 8.8, que es aproximadamente un tercio del Capítulo 8).

Capítulo 10, sobre direcciones futuras (Sección 10.7, aproximadamente un cuarto de ese capítulo).

Aquellos que, por presión del tiempo, quieran saltar algunas de estas secciones, para hacer el salto más fácil, las secciones de «Juntando todo» de los Capítulos 6 y 8 son independientes de los temas avanzados.

El lector puede haber observado que el punto flotante se cubre en el Apéndice A en lugar de en un capítulo. Como esto es muy independiente del material restante, nuestra solución fue incluirlo como un apéndice cuando nuestras informaciones indicaron que un porcentaje significativo de los lectores tenían conocimientos sobre punto flotante.

Los apéndices restantes se incluyen tanto como referencias para los profesionales de los computadores como para los Ejercicios. El Apéndice B contiene los repertorios de instrucciones de tres máquinas clásicas: la IBM 360, el Intel 8086 y la DEC VAX. Los Apéndices C y D dan la mezcla de instrucciones de programas reales para estas máquinas más DLX, medidas por frecuencia de instrucciones o por frecuencia de tiempo. El Apéndice E ofrece una visión comparativa, más detallada, de algunas arquitecturas recientes.

Ejercicios, proyectos y software

La naturaleza opcional del material también se refleja en los Ejercicios. Los corchetes para cada pregunta (capítulo.sección) indican las secciones de texto de relevancia principal para responder la pregunta. Esperamos que esto ayude a los lectores a evitar ejercicios para los cuales no han leído la sección correspondiente, así como suministrar la fuente para revisión. Hemos adoptado la técnica de Donald Knuth de puntuar los Ejercicios. Las puntuaciones dan una estimación de la cantidad de esfuerzo que puede llevar un problema:

[10] Un minuto (lectura y comprensión).

[20] De quince a veinte minutos para la respuesta completa.

[25] Una hora para escribir la respuesta completa.

[30] Proyecto corto de programación: menos de un día completo de programación.

[40] Proyecto significativo de programación: dos semanas de tiempo de realización.

[50] Proyecto largo (de dos a cuatro semanas para dos personas).

[Discusión] Tópico para discusión con otros interesados en arquitectura de computadores.

Observaciones finales

Este libro es inusual, ya que no hay orden estricto en los nombres de los autores. La mitad de las veces aparecerá Hennessy y Patterson, tanto en este libro como en los anuncios, y la otra mitad aparecerá Patterson y Hennessy. Incluso aparecerán ambas formas en las publicaciones bibliográficas tales como *Libros en Impresión*. (Cuando referenciamos el libro, alternamos el orden de los autores.) Esto refleja la verdadera naturaleza colaboradora de este libro: juntos, reflexionamos sobre las ideas y métodos de presentación, después, individualmente, cada uno escribió una mitad de los capítulos y actuó como re-

visor del borrador de la otra mitad. (¡En efecto, el número final de páginas sugiere que cada uno escribió el mismo número de páginas!) Pensamos que la mejor forma de reflejar esta genuina cooperación es la de no ocultarse en la ambigüedad —una práctica que puede ayudar a algunos autores pero confunde a los bibliotecarios. Por ello, compartimos igualmente la culpa de lo que está a punto de leer.

JOHN HENNESSY DAVID PATTERSON

Enero 1990

Agradecimientos

Este libro se escribió con la ayuda de gran número de personas —tantas, en efecto, que algunos autores podrían detenerse aquí, indicando que eran demasiadas para ser nombradas. Sin embargo, declinamos utilizar esa excusa, porque se ocultaría la magnitud de la ayuda que necesitamos. Por ello, citamos a las 137 personas y cinco instituciones a las que tenemos que dar las gracias.

Cuando decidimos añadir un apéndice sobre punto flotante, que caracterizase el estándar del IEEE, pedimos a muchos colegas que nos recomendasen una persona que comprendiese ese estándar y que pudiese escribir bien y explicar con sencillez complejas ideas. **David Goldberg,** de Xerox Palo Alto Research Center, realizó todas estas tareas admirablemente, poniendo un nivel al que esperamos se encuentre el resto del libro.

Margo Seltzer de la U.C. Berkeley merece un reconocimiento especial. No sólo porque fue la primera profesora ayudante del curso en Berkeley que utilizó el material, sino porque reunió todo el software, benchmarks y trazas. También ejecutó las simulaciones de las caches y del repertorio de instrucciones que aparecen en el Capítulo 8. Le agradecemos su prontitud y fiabilidad al seleccionar partes de software y juntar todo en un paquete coherente.

Bryan Martin y Truman Joe de Stanford también merecen nuestro agradecimiento especial por leer rápidamente los Ejercicios de los primeros capítulos poco antes de publicar el libro. Sin su dedicación, los Ejercicios habrían estado considerablemente menos pulidos.

Nuestro plan para desarrollar este material fue ensayar primero las ideas a finales de 1988 en los cursos que impartíamos en Berkeley y Stanford. Creamos notas de clase; primero las ensayamos con los estudiantes de Berkeley (porque el año académico en Berkeley comienza antes que en Stanford), detectando algunos errores, y después exponiendo estas ideas a los estudiantes de Stanford. Esta puede no haber sido la mejor experiencia de sus vidas académicas, por ello deseamos dar las gracias a aquellos que «voluntariamente» fueron conejillos de indias; así como a los profesores ayudantes **Todd Narter, Margo Seltzer** y **Eric Williams,** que sufrieron las consecuencias del desarrollo de esta experiencia.

El siguiente paso del plan fue redactar un borrador del libro durante el invierno de 1989. Esperábamos hacerlo pasando a limpio las notas de clase, pero nuestra realimentación de los estudiantes y la reevaluación, que es parte de cualquier redacción, convirtió esto en una tarea mucho más amplia de lo que

esperábamos. Esta versión «Alpha» se envió a los revisores en la primavera de 1989. Nuestro agradecimiento especial a **Anoop Gupta** de la Universidad de Stanford y a **Forest Baskett** de Silicon Graphics, que utilizaron la versión «Alpha» para impartir una clase en Stanford en la primavera de 1989.

La arquitectura es un campo que tiene una cara académica y otra industrial. Nos relacionamos con ambos tipos de expertos para revisar el material de este libro. Los revisores académicos de la versión «Alpha» incluyen a **Thomas Casavant** de la Universidad de Purdue, **Jim Goodman** de la Universidad de Wisconsin en Madison, **Roger Keickhafer** de la Universidad de Nebraska, **Hank Levy** de la Universidad de Washington, **Norman Matloff** de la Universidad de California en Davis, **David Meyer** de la Universidad de Purdue, **Trevor Mudge** de la Universidad de Michigan, **Victor Nelson** de la Universidad de Auburn, **Richard Reid** de la Universidad del Estado de Michigan y **Mark Smotherman** de la Universidad de Clemson. También deseamos dar las gracias a aquellos que nos comentaron nuestro borrador a finales del año 1989: **Bill Dally** del MIT, y **Jim Goodman, Hank Levy, David Meyer** y **Joseph Pfeiffer** del Estado de Nuevo México. En abril de 1989, algunos de nuestros planes fueron examinados en un grupo de discusión que incluía a **Paul Barr** de la Universidad del Noreste, **Susan Eggers** de la Universidad de Washington, **Jim Goodman** y **Mark Hill** de la Universidad de Wisconsin, **James Mooney** de la Universidad de Virginia Oeste, y **Larry Wittie** de SUNY Stony Brook. Apreciamos sus valiosas sugerencias.

Antes de citar a los revisores industriales, nuestro especial agradecimiento a **Douglas Clark** de DEC, que nos dio más comentarios sobre la versión «Alpha» que los demás revisores juntos, y cuyas observaciones fueron siempre escritas cuidadosamente teniendo en cuenta nuestra naturaleza sensible. Otras personas que revisaron algunos capítulos de la versión «Alpha» fueron **David Douglas** y **David Wells** de Thinking Machines, **Joel Emer** de DEC, **Earl Killian** de MIPS Computer Systems Inc., y **Jim Smith** de Cray Research. **Earl Killian** también explicó los misterios del analizador del repertorio de las instrucciones «Pixie» y nos proporcionó una versión no publicada para recoger estadísticas de saltos.

También damos las gracias a **Maurice Wilkes** de Olivetti Research y a **C. Gordon Bell** de Stardent por ayudarnos a mejorar nuestras versiones de la historia de los computadores al final de cada capítulo.

Además de aquellos que, voluntariamente, leyeron muchos capítulos, también deseamos dar las gracias a los que nos hicieron sugerencias del material a incluir o revisaron la versión «Alpha» de los capítulos:

Capítulo 1: **Danny Hillis** de Thinking Machines por sus sugerencias sobre la asignación de recursos, de acuerdo con su contribución al rendimiento.

Capítulo 2: **Andy Bechtolsheim** de Sun Microsystems por sus sugerencias del precio frente al coste y las estimaciones de coste de las estaciones de trabajo; **David Hodges** de la Universidad de California en Berkeley, **Ed Hudson** y **Mark Johnson** de MIPS, **Al Marston** y **Jim Slager** de Sun, **Charles Stapper** de IBM, y **David Wells** de Thinking Machines por explicar el rendimiento y fabricación de los circuitos integrados: **Ken Lutz** de la U.C. Berlekey y el **servicio de circuitos integrados FAST** de USC/ISI por las citas de precios de los chips; **Andy Bechtolsheim** y **Nhan Chu** de Sun Microsystems, **Don Lewine** de Data General, y **John Mashey** y **Chris Rowen** de MIPS que también revisaron este capítulo.

Capítulo 4: **Tom Adams** de Apple y **Richard Zimmermann** de la Universidad del Estado de San Francisco por sus estadísticas del Intel 8086; **John Crawford** de Intel por revisar el 80×86 y otros materiales; **Lloyd Dickman** por revisar el material de la IBM 360.

Capítulo 5: **Paul Carrick** y **Peter Stoll** de Intel por las revisiones.

Capítulo 7: **David Bailey** de NASA Ames y **Norm Jouppi** de DEC por las revisiones.

Capítulo 8: **Ed Kelly** de Sun por su ayuda en la explicación de las alternativas de las DRAM y **Bob Cmelik** de Sun por las estadísticas de SPIX; **Anant Agarwal** de MIT, **Susan Eggers** de la Universidad de Washington, **Mark Hill** de la Universidad de Wisconsin en Madison, y **Steven Przybylski** de MIPS por el material de sus disertaciones; y **Susan Eggers** y **Mark Hill** por las revisiones.

Capítulo 9: **Jim Brady** de IBM por proporcionar referencias cuantitativas para datos sobre tiempo de respuesta y computadores IBM y revisar el capítulo; **Garth Gibson** de la Universidad de California en Berkeley por ayudarnos con las referencias del bus y por revisar el capítulo; **Fred Berkowitz** de Omni Solutions, **David Boggs** de DEC, **Pete Chen** y **Randy Katz** de la Universidad de California en Berkeley, **Mark Hill** de la Universidad de Wisconsin, **Robert Shomler** de IBM, y **Paul Taysom** de AT&T Bell Laboratories por las revisiones.

Capítulo 10: **C. Gordon Bell** de Stardent por su sugerencia de incluir un multiprocesador en la sección de «Juntando todo»; **Susan Eggers, Danny Hills** de Thinking Machines, y **Shreekant Thakkar** de Sequent Computer por las revisiones.

Apéndice A: Los datos sobre REM IEEE y la reducción de argumentos en la Sección A.6, así como el teorema $p \leqslant (q-1)/2$ (pág. 671) se tomaron de notas de clase no publicadas de **William Kahan** de la U.C. Berlekey (y no sabemos de ninguna fuente publicada que contenga una explicación de estos aspectos). El dato de SDRWAVE es de **David Hough** de Sun Microsystems. **Mark Birman** de Weitek Corporation, **Merrick Darley** de Texas Instruments, y **Mark Johnson** de MIPS proporcionaron información sobre el: 3364, 8847 y R3010, respectivamente. **William Kahan** también leyó un borrador de este capítulo e hizo muchos comentarios útiles. También agradecemos a **Forrest Brewer** de la Universidad de California en Santa Barbara, **Milos Ercegovac** de la Universidad de California en Los Angeles, **Bill Shannon** de Sun Microsystems, y **Behrooz Shirazi** de la Universidad Metodista del Southern por las revisiones.

Aunque muchos nos advirtieron de que ahorrásemos esfuerzos y publicásemos el libro lo antes posible, perseguíamos el objetivo de publicar el libro lo más claro posible con la ayuda de un grupo adicional de personas involucradas en la ronda final de revisiones. Este libro no sería tan útil sin la ayuda de instructores atrevidos, profesores ayudantes, y estudiantes trabajadores, que aceptaron el papel del examen «Beta» en el programa de examen de clase; realizamos cientos de cambios como resultado del test «Beta». Las instituciones e instructores del test «Beta» fueron:

Universidad de Carnegie-Mellon	**Daniel Siewiorek**
Universidad de Clemson	**Mark Smotherman**
Universidad de Cornell	**Keshav Pingali**
Universidad del Estado de Pennsylvania	**Mary Jane Irwin/Bob Owens**
Universidad del Estado de San Francisco	**Vojin Oklobdzija**
Universidad Sureste del Estado de Missouri	**Anthony Duben**
Universidad Meridional Metodista	**Behrooz Shirazi**
Universidad de Stanford	**John Hennessy**
Universidad del Estado de Nueva York de Stony Brook	**Larry Wittie**

Universidad de California de Berkeley	**Vojin Oklobdzija**
Universidad de California de Los Angeles	**David Rennels**
Universidad de California de Santa Cruz	**Daniel Helman**
Universidad de Nebraska	**Roger Kieckhafer**
Universidad de Carolina del Norte de Chapel Hill	**Akhilesh Tyagi**
Universidad de Texas de Austin	**Josep Rameh**
Universidad de Waterloo	**Bruno Preiss**
Universidad de Wisconsin de Madison	**Mark Hill**
Universidad de Washington (St. Louis)	**Mark Franklin**

Mención especial merecen **Daniel Helman, Mark Hill, Mark Smotherman** y **Larry Wittie** que fueron especialmente generosos con sus sugerencias. A la compilación de las soluciones de los ejercicios para los instructores de los cursos contribuyeron **Evan Tick** de la Universidad de Oregon, **Susan Eggers** de la Universidad de Washington, y **Anoop Gupta** de la Universidad de Stanford.

Las clases en SUNY Stony Brook, Carnegie-Mellon, Stanford, Clemson y Wisconsin nos proporcionaron el mayor número de descubrimientos de errores en la versión «Beta». Nos gustaría hacer notar que se encontraron numerosos errores y los corrigieron las siguientes personas: **Michael Butler, Rohit Chandra, David Cummings, David Filo, Carl Feynman, John Heinlein, Jim Quinlan, Andras Radics, Peter Schnorf** y **Malcolm Wing.**

Además de los exámenes de clase, también pedimos nuevamente ayuda a nuestros amigos de la industria. Nuestro agradecimiento a **Jim Smith** de Cray Research por su minuciosa revisión y sus previsoras sugerencias del texto completo «Beta». Las siguientes personas también nos ayudaron a mejorar la versión «Beta», y se lo agradecemos:

Ben Hao de Sun Microsystems por revisar la versión «Beta» completa.

Ruby Lee de Hewlett-Packard y **Bob Supnik** de DEC por revisar algunos capítulos.

Capítulo 2: **Steve Goldstein** de Ross Semiconductor y **Sue Stone** de Cypress Semiconductor por las fotografías y la oblea del CY7C601 (págs. 61-62); **John Crawford** y **Jacque Jarve** de Intel por las fotografías y obleas del Intel 80486 (págs. 60-62); y **Dileep Bhandarkar** de DEC por ayudarnos con la versión VMS de Spice y TeX utilizadas en los Capítulos 2-4.

Capítulo 6: **John DeRosa** de DEC por ayudarnos con la segmentación del 8600.

Capítulo 7: **Corinna Lee** de la Universidad de California en Berkeley por las medidas de los Cray X-MP e Y-MP y por las revisiones.

Capítulo 8: **Steve Przybylski** de MIPS por las revisiones.

Capítulo 9: **Dave Anderson** de Imprimis por las revisiones y por suministrarnos material sobre tiempos de acceso al disco; **Jim Brady** y **Robert Shomler** de IBM por las revisiones, y **Pete Chen** de Berkeley por las sugerencias en las fórmulas de rendimiento del sistema.

Capítulo 10: **C. Gordon Bell** por las revisiones, incluyendo algunas sugerencias sobre las clasificaciones de las máquinas MIMD, y **David Douglas** y **Danny Hillis** de Thinking Machines por las explicaciones sobre procesadores paralelos del futuro.

Apéndice A: **Mark Birman** de Weitek Corporation, **Merrick Darley** de Texas Instruments, y **Mark Johnson** de MIPS por las fotografías y planos de planta de los chips (págs. 641); y **David Chenevert** de Sun Microsystems por las revisiones.

Apéndice E: Se añadió después de la versión «Beta», y agradecemos a las siguientes personas sus revisiones: **Mitch Alsup** de Motorola, **Robert Garner** y **David Weaver** de Sun Microsystems, **Earl Killian** de MIPS Computer Systems, y **Les Kohn** de Intel.

Aunque hemos hecho todo lo posible para eliminar los errores y para corregir los señalados por los revisores, ¡sólo nosotros somos responsables de todo lo que queda!

También queremos agradecer a la «**Defense Advanced Research Projects Agency**» por soportar nuestra investigación durante muchos años. Esa investigación fue la base de muchas ideas que aparecen en este libro. En particular, queremos dar las gracias a los gestores actuales y anteriores del programa: **Duane Adams, Paul Losleben, Mark Pullen, Steve Squires, Bill Bandy** y **John Toole.**

Nuestro agradecimiento a **Richard Swan** y sus colegas de DEC Western Research Laboratory por proporcionarnos un escondrijo para escribir las versiones «Alpha» y «Beta», y a **John Ousterhout** de la U.C. Berkeley, que siempre estuvo dispuesto (e incluso un poco impaciente) para actuar como abogado del diablo por los méritos de las ideas de este libro durante este viaje de Berkeley a Palo Alto. También damos las gracias a **Thinking Machines Corporation** por proporcionarnos un refugio durante la revisión final.

Este libro no se podría haber publicado sin un editor. **John Wakerley** nos dio valiosos consejos sobre cómo encontrar un editor. Seleccionamos a Morgan Kaufmann Publishers, Inc., y no nos arrepentimos de esa decisión. (¡No todos los autores piensan de esta forma acerca de su editor!) Comenzando con notas de clase justo después del Día de Año Nuevo de 1989, completamos la versión «Alpha» en cuatro meses. En los tres meses siguientes recibimos revisiones de 55 personas y terminamos la versión «Beta». Después de los exámenes de clase con 750 estudiantes a finales de 1989 y más revisiones de la industria, conseguimos la versión final justo antes de las Navidades de 1989. Con todo, el libro estuvo disponible en marzo de 1990. No conocemos otro editor que llevase el mismo paso con una planificación rigurosa. Deseamos agradecer a **Shirley Jowell** el aprendizaje sobre segmentación y riesgos en la segmentación y ver cómo aplicarlos para editarlos. Nuestro agradecimiento más caluroso a nuestro editor **Bruce Spatz** por su guía y su humor en nuestra aventura de escritores. También queremos dar las gracias a los miembros de la amplia familia de Morgan Kaufmann: **Walker Cunningham** por la edición técnica, **David Lance Goines** por el diseño de la cubierta, **Gary Head** por el diseño del libro, **Linda Medoff** por la edición de producción y copia, **Fifth Street Computer Services** por el mecanografiado y **Paul Medoff** por las pruebas de lectura y asistencia de producción.

También debemos dar las gracias a nuestro «staff» de la universidad, **Darlene Hadding, Bob Miller, Margaret Rowland** y **Terry Lessard-Smith** por los innumerables «faxes» y correos express, así como por encargarse del trabajo en Stanford y Berkeley mientras trabajamos en el libro. **Linda Simko** y **Kim Barger** de Thinking Machines también nos proporcionaron numerosos correos express durante el otoño.

Finalmente, gracias a nuestras familias por su paciencia a lo largo de largas noches y tempranas mañanas de lectura, mecanografiado y abandono.

Sobre los autores

DAVID A. PATTERSON (Universidad de California en Berkeley) ha enseñado arquitectura de computadores desde su incorporación al cuerpo docente en 1977, y actualmente es presidente de la división de Informática. Ha sido consultor de algunas compañías, incluyendo Digital Equipment Corporation, Intel y Sun Microsystems. El doctor Patterson es también «Corporate Fellow» de Thinking Machines y miembro del «Scientific Advisory Board» de Data General. Ha sido galardonado por la Universidad de California con el «Distinguished Teaching Award» y es «Fellow» del Institute of Electrical and Electronic Engineers.

En Berkeley dirigió el diseño e implementación del RISC I, probablemente el primer computador VLSI de repertorio de instrucciones reducido. Más tarde, sus proyectos produjeron varias generaciones de RISC, así como dos «distinguished disertation awards» de la Association for Computing Machinery (ACM). Esta investigación constituyó los fundamentos de la arquitectura SPARC, actualmente utilizada por AT&T, Fujitsu, ICL, LSI Logic, Philips, Sun, TI y Xerox. Su investigación actual se centra en el desarrollo de nuevos sistemas de entrada/salida para incrementar el rendimiento de las CPU.

JOHN L. HENNESSY (Universidad de Stanford) ha sido miembro del cuerpo docente desde 1977. Es director del Computer Systems Laboratory en Stanford, donde enseña arquitectura de computadores y supervisa un grupo de estudiantes de doctorado. Es el titular actual de la cátedra Willard R. e Inez Kerr Bell de la Escuela de Ingeniería.

El área de investigación original de Hennessy fue la optimización de compiladores. Su grupo de investigación en Stanford desarrolló muchas de las técnicas que ahora se usan comercialmente. En 1981 comenzó el proyecto MIPS en Stanford con un grupo de estudiantes graduados. Después de completar el proyecto en 1984, dejó un año la universidad y fue uno de los fundadores de MIPS Computer Systems. La arquitectura RISC desarrollada en MIPS ha sido adoptada por una serie de compañías, entre las que se encuentran DEC, Honeywell-Bull, NEC, Siemens, Silicon Graphics y Tandem. Continúa ejerciendo como jefe científico en MIPS. Su investigación actual en Stanford se dirige al área del diseño y explotación de multiprocesadores.

Y ahora algo completamente diferente.

Circo volante de Monty Python.

1.1 Introducción

1.2 Definiciones de rendimiento

1.3 Principios cuantitativos del diseño de computadores

1.4. El trabajo de un diseñador de computadores

1.5 Juntando todo: el concepto de jerarquía de memoria

1.6 Falacias y pifias

1.7 Observaciones finales

1.8 Perspectiva histórica y referencias

Ejercicios

1 Fundamentos del diseño de computadores

1.1 Introducción

La tecnología de computadores ha progresado increíblemente en los últimos cincuenta años. En 1945, no había computadores de programa almacenado. Hoy, con unos pocos miles de dólares se puede comprar un computador personal con más prestaciones, más memoria principal y más memoria de disco que un computador que, en 1965, costaba un millón de dólares. Este rápido crecimiento en las prestaciones es consecuencia de los avances en la tecnología utilizada en la construcción de computadores y de las innovaciones en los diseños de los mismos. El aumento del rendimiento de las máquinas se indica en la Figura 1.1. Mientras que las mejoras tecnológicas han sido bastante constantes, el progreso en la obtención de mejores arquitecturas ha sido mucho menos consistente. Durante los veinticinco primeros años de los computadores electrónicos, ambas fuerzas contribuían de forma importante; pero durante los veinte últimos años, los diseñadores de computadores han dependido enormemente de la tecnología de circuitos integrados. El crecimiento del rendimiento durante este período varía del 18 al 35 por 100 por año, dependiendo de la clase de computador.

Entre todas las líneas de computadores, la velocidad de crecimiento de los grandes computadores (*mainframes*) es la que más se debe a la tecnología —la mayoría de las innovaciones en organización y arquitectura se introdujeron en estas máquinas hace muchos años. Los supercomputadores han crecido gracias a las mejoras tecnológicas y arquitectónicas (ver Cap. 7). Los avances en minicomputadores han incluido formas innovadoras de implementar arquitecturas, así como la adopción de muchas de las técnicas de los grandes

computadores. El crecimiento del rendimiento de los microcomputadores ha sido el más rápido, en cierto modo porque estas máquinas aprovecharon las ventajas que ofrece la tecnología de circuitos integrados. Además, desde 1980, la tecnología de microprocesadores ha sido la tecnología elegida para las nuevas arquitecturas y las nuevas implementaciones de arquitecturas antiguas.

Dos cambios significativos en el mercado de computadores han hecho más fácil que antes el éxito comercial de las nuevas arquitecturas. En primer lugar, la eliminación virtual de la programación en lenguaje ensamblador ha reducido drásticamente la necesidad de la compatibilidad del código objeto. En segundo lugar, la creación de sistemas operativos estandarizados, independientes de los vendedores, como por ejemplo UNIX, ha reducido el coste y riesgo de lanzar al mercado una nueva arquitectura. Por consiguiente ha habido un

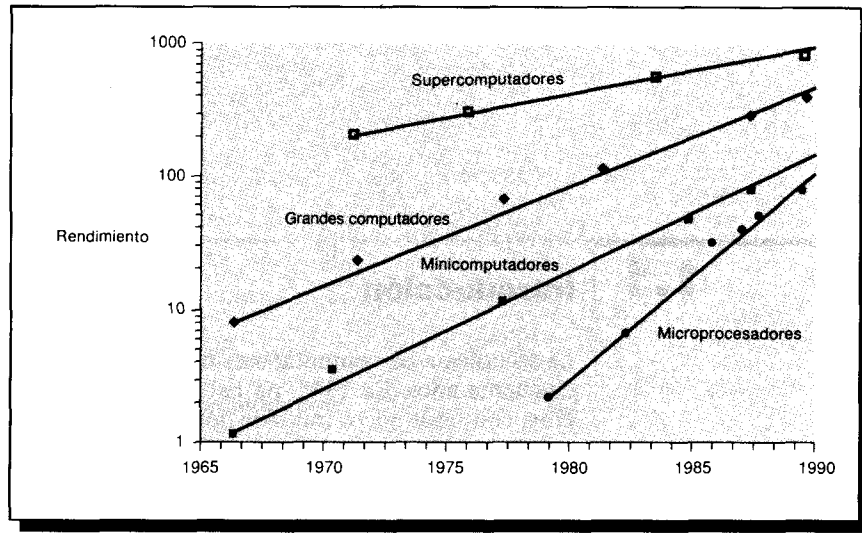


FIGURA 1.1 Crecimiento del rendimiento a lo largo de los últimos años de las diferentes clases de computadores. El eje vertical muestra el rendimiento relativo y el horizontal el año de introducción. Las clases de computadores están definidas, básicamente por su coste. Los *supercomputadores* son los más caros —desde, aproximadamente, un millón a decenas de millones de dólares. Diseñados principalmente para aplicaciones científicas, son las máquinas de más alto rendimiento. Los *computadores grandes* (mainframes) son máquinas de propósito general de altas prestaciones, normalmente cuestan más de medio millón de dólares y, como máximo, unos pocos millones de dólares. Los *minicomputadores* son máquinas de tamaño medio que cuestan entre cincuenta mil dólares y diez veces esa cantidad. Finalmente, los *microcomputadores* varían desde los pequeños computadores personales que cuestan unos pocos miles de dólares a las grandes y potentes estaciones de trabajo que cuestan cincuenta mil o más dólares. La velocidad de crecimiento en el rendimiento de los supercomputadores, minicomputadores y ordenadores grandes ha sido del 20 por 100 por año, mientras que la velocidad de crecimiento en el rendimiento para los microprocesadores ha sido, aproximadamente, del 35 por 100 por año.

renacimiento en el diseño de computadores: hay muchas nuevas compañías trabajando en nuevas direcciones arquitectónicas, con las nuevas familias de computadores que emergen —mini-supercomputadores, microprocesadores de alto rendimiento, supercomputadores gráficos y un amplio rango de multiprocesadores— a mayor velocidad que nunca.

Comenzando en 1985, la industria de computadores vio un nuevo estilo de arquitectura aprovechando esta oportunidad e iniciando un período en el cual el rendimiento ha aumentado a una velocidad mucho más rápida. Al aunar los avances en la tecnología de circuitos integrados, las mejoras en la tecnología de compiladores y las nuevas ideas arquitectónicas, los diseñadores pudieron crear una serie de máquinas que mejoraban el rendimiento, en un factor de casi 2, cada año. Estas ideas están ahora proporcionando una de las mejoras en rendimiento más significativamente sostenidas en los últimos veinte años. Esta mejora ha sido posible al haber tenido en cuenta una serie de importantes avances tecnológicos junto a un mejor conocimiento empírico sobre la utilización de los computadores. De esta fusión ha emergido un estilo de diseño de computadores basado en datos empíricos, experimentación y simulación. Este estilo y aproximación al diseño de computadores se reflejará en este texto.

Continuar las mejoras en costo y rendimiento de los últimos veinticinco a cincuenta años requerirá innovaciones continuas en el diseño de computadores, y los autores piensan que las innovaciones estarán basadas en esta aproximación cuantitativa de la arquitectura de computadores. Por consiguiente, este libro se ha escrito no sólo para documentar este estilo de diseño, sino también para estimular a que el lector contribuya en este campo.

1.2 Definiciones de rendimiento

Para familiarizar al lector con la terminología y conceptos de este libro, este capítulo introduce algunos términos e ideas clave. Ejemplos de las ideas mencionadas aquí aparecen a lo largo del libro, y algunas de ellas —segmentación *(pipelining),* jerarquía de memoria, rendimiento de la CPU y medida de costes— son el núcleo de capítulos completos. Comencemos con definiciones de rendimiento relativo.

Cuando se dice que un computador es más rápido que otro, ¿qué queremos significar? El usuario del computador puede decir que un computador es más rápido cuando ejecuta un programa en menos tiempo, mientras que el director de un centro de cálculo puede decir que un computador es más rápido cuando completa más tareas en una hora. El usuario del computador está interesado en reducir el *tiempo de respuesta* —el tiempo transcurrido entre el comienzo y el final de un evento— denominado también *tiempo de ejecución* o *latencia.* El director del centro de cálculo está interesado en incrementar la *productividad (throughput)* —la cantidad total de trabajo realizado en un tiempo determinado— a veces denominado *ancho de banda.* Normalmente, los términos «tiempo de respuesta», «tiempo de ejecución» y «productividad» se utilizan cuando se está desarrollando una tarea de cálculo completa. Los términos «latencia» y «ancho de banda» casi siempre se eligen cuando se ha-

bla de un sistema de memoria. Todos estos términos aparecerán a lo largo de este texto.

Ejemplo

¿Las siguientes mejoras en rendimiento incrementan la productividad, hacen disminuir el tiempo de respuesta, o ambas cosas?

1. Ciclo de reloj más rápido.
2. Múltiples procesadores para tareas separadas (tratamiento del sistema de reservas de una compañía aérea, para un país, por ejemplo).
3. Procesamiento paralelo de problemas científicos.

Respuesta

La disminución del tiempo de respuesta, habitualmente, mejora la productividad. Por consiguiente, 1 y 3 mejoran el tiempo de respuesta y la productividad. En 2, ninguna tarea funciona más rápida, por tanto, sólo incrementa la productividad.

A veces estas medidas se describen mejor con distribuciones de probabilidad en lugar de con valores constantes. Por ejemplo, consideremos el tiempo de respuesta para completar una operación de E/S en un disco. El tiempo de respuesta depende de una serie de factores no determinísticos, como lo que el disco esté haciendo en el instante de la petición de E/S y del número de tareas que están esperando acceder al disco. Debido a que estos valores no son fijos, tiene más sentido hablar de tiempo medio de respuesta de un acceso al disco. De igual forma, la productividad efectiva del disco —el número de datos que realmente va o viene del disco por unidad de tiempo— no es un valor constante. En la mayor parte de este texto, trataremos el tiempo de respuesta y la productividad como valores determinísticos, aunque esto cambiará en el Capítulo 9, cuando hablemos de E/S.

Cuando se comparan alternativas de diseño, con frecuencia, queremos relacionar el rendimiento de dos máquinas diferentes, por ejemplo X e Y. La frase «X es más rápida que Y» se utiliza aquí para significar que el tiempo de respuesta o tiempo de ejecución es inferior en X que en Y para una tarea dada. En particular, «X es *n por 100* más rápido que Y» significa

$$\frac{\text{Tiempo de ejecución}_Y}{\text{Tiempo de ejecución}_X} = 1 + \frac{n}{100}$$

Como el tiempo de ejecución es el recíproco del rendimiento, se mantiene la siguiente relación:

$$1 + \frac{n}{100} = \frac{\text{Tiempo de ejecución}_Y}{\text{Tiempo de ejecución}_X} = \frac{\dfrac{1}{\text{Rendimiento}_Y}}{\dfrac{1}{\text{Rendimiento}_X}} = \frac{\text{Rendimiento}_X}{\text{Rendimiento}_Y}$$

Algunas personas consideran un incremento en el rendimiento, *n*, como la diferencia entre el rendimiento de la máquina más rápida y la más lenta, dividido por el rendimiento de la máquina más lenta. Esta definición de *n* es exactamente equivalente a nuestra primera definición, como podemos ver:

$$n = 100\left(\frac{\text{Rendimiento}_X - \text{Rendimiento}_Y}{\text{Rendimiento}_Y}\right)$$

$$\frac{n}{100} = \frac{\text{Rendimiento}_X}{\text{Rendimiento}_Y} - 1$$

$$1 + \frac{n}{100} = \frac{\text{Rendimiento}_X}{\text{Rendimiento}_Y} = \frac{\text{Tiempo de ejecución}_Y}{\text{Tiempo de ejecución}_X}$$

La frase «la productividad de X es el 30 por 100 superior que la de Y» significa que el número de tareas completadas por unidad de tiempo en la máquina X es 1,3 veces el número de tareas completadas en la máquina Y.

Ejemplo

Si la máquina A ejecuta un programa en diez segundos y la máquina B ejecuta el mismo programa en quince segundos, ¿cuál de las siguientes sentencias es verdadera?

- A es el 50 por 100 más rápida que B.
- A es el 33 por 100 más rápida que B.

Respuesta

Que la máquina A sea el *n por 100* más rápida que la máquina B puede expresarse como

$$\frac{\text{Tiempo de ejecución}_B}{\text{Tiempo de ejecución}_A} = 1 + \frac{n}{100}$$

o

$$n = \frac{\text{Tiempo de ejecución}_B - \text{Tiempo de ejecución}_A}{\text{Tiempo de ejecución}_A} \cdot 100$$

Así que,

$$\frac{15 - 10}{10} \cdot 100 = 50$$

A es, por tanto, el 50 por 100 más rápida que B.

Para ayudar a prevenir malentendidos —y debido a la falta de definiciones consistentes para «más rápido que» y «más lento que»— nunca utilizaremos la frase «más lento que» en una comparación cuantitativa de rendimiento.

Como rendimiento y tiempo de ejecución son recíprocos, incrementar el rendimiento hace decrecer el tiempo de ejecución. Para ayudar a evitar confusiones entre los términos «incrementar» y «decrementar», habitualmente, diremos «mejorar el rendimiento» o «mejorar el tiempo de ejecución» cuando queramos significar *incremento* de rendimiento y *disminución* de tiempo de ejecución.

Productividad y latencia interactúan de forma diferente en los diseños de computadores. Una de las interacciones más importantes se presenta en la segmentación *(pipelining)*. La *segmentación* es una técnica de implementación que mejora la productividad al solapar la ejecución de múltiples instrucciones; la segmentación se explica con detalle en el Capítulo 6. La segmentación de instrucciones es análoga a utilizar una línea de ensamblaje para fabricar coches. En una línea de ensamblaje, se pueden tardar ocho horas en construir un coche completo, pero si hay ocho pasos en la línea de ensamblaje, cada hora se fabrica un nuevo coche. En la línea de ensamblaje, no se ve afectada la latencia para construir un coche, pero la productividad aumenta proporcionalmente con el número de etapas de la línea, si todas las etapas son de la misma duración. El hecho de que la segmentación en los computadores tenga algún gasto por etapa incrementa la latencia en cierta cantidad para cada etapa del cauce.

1.3 Principios cuantitativos del diseño de computadores

Esta sección introduce algunas reglas y observaciones importantes para diseñar computadores.

Acelerar el caso común

Quizá el principio más importante y generalizado del diseño de computadores sea acelerar el caso común: al realizar un diseño, favorecer el caso frecuente sobre el infrecuente. Este principio también se aplica cuando se determina cómo emplear recursos, ya que el impacto de hacer alguna ocurrencia más rápida es mucho mayor si la ocurrencia es frecuente. Mejorar el evento frecuente en lugar del evento raro, evidentemente, también ayudará a aumentar el rendimiento. Además, el caso frecuente es, a menudo, más simple y puede realizarse de forma más rápida que el caso infrecuente. Por ejemplo, cuando sumamos dos números en la *unidad central de proceso* (CPU), podemos esperar que el desbordamiento *(overflow)* sea una circunstancia infrecuente y, por tanto, podemos mejorar el rendimiento optimizando el caso más común de ausencia de *desbordamiento.* Este hecho puede ralentizar la situación en la que se presente un *desbordamiento,* pero si este caso es infrecuente, el rendimiento global mejorará al optimizar el caso normal.

Veremos muchos casos de este principio a lo largo de este texto. Al aplicar este sencillo principio, hemos de decidir cuál es el caso frecuente y cómo se puede mejorar el rendimiento haciendo este caso más rápido. Una ley fun-

damental, denominada *Ley de Amdahl,* puede utilizarse para cuantificar este principio.

Ley de Amdahl

El aumento de rendimiento que puede obtenerse al mejorar alguna parte de un computador puede calcularse utilizando la Ley de Amdahl. La *Ley de Amdahl* establece que la mejora obtenida en el rendimiento al utilizar algún modo de ejecución más rápido está limitada por la fracción de tiempo que se pueda utilizar ese modo más rápido.

La Ley de Amdahl define la ganancia de rendimiento o aceleración *(speedup)* que puede lograrse al utilizar una característica particular. ¿Qué es la aceleración? Supongamos que podemos hacer una mejora en una máquina que cuando se utilice aumente su rendimiento. La aceleración *(speedup)* es la relación

$$\text{Aceleración de rendimiento} = \frac{\text{Rendimiento de la tarea completa utilizando la mejora cuando sea posible}}{\text{Rendimiento de la tarea completa sin utilizar la mejora}}$$

Alternativamente:

$$\text{Aceleración de rendimiento} = \frac{\text{Tiempo de ejecución de la tarea sin utilizar la mejora}}{\text{Tiempo de ejecución de la tarea completa utilizando la mejora cuando sea posible}}$$

La aceleración nos indica la rapidez con que se realizará una tarea utilizando una máquina con la mejora con respecto a la máquina original.

Ejemplo

Considerar el problema de viajar desde Nevada a California a través de las montañas de Sierra Nevada y del desierto de Los Angeles. Hay disponibles varios tipos de vehículos, pero, desgraciadamente, el viaje se realiza a través de áreas ecológicamente sensibles por las montañas que hay que atravesar. Se emplean veinte horas en recorrer a pie las montañas. Sin embargo, existe la posibilidad de recorrer las últimas 200 millas en un vehículo de alta velocidad. Hay cinco formas de completar la segunda parte del viaje:

1. Ir a pie a una velocidad media de 4 millas por hora.
2. Montar en bicicleta a una velocidad media de 10 millas por hora.
3. Conducir un «Hyundai Excel» en el cual la velocidad media es de 50 millas por hora.
4. Conducir un «Ferrari Testarossa» en el cual la velocidad media es de 120 millas por hora.

5. Conducir un vehículo oruga en el cual la velocidad media es de 600 millas por hora.

¿Cuánto se tardará en realizar el viaje completo utilizando estos vehículos, y cuál es el aumento de velocidad si se toma como referencia el recorrido a pie de la distancia completa?

Respuesta Podemos encontrar la solución determinando cuánto durará la segunda parte del viaje y sumando ese tiempo a las veinte horas necesarias para cruzar las montañas. La Figura 1.2 muestra la efectividad de utilizar los diversos modos, mejorados, de transporte.

La Ley de Amdahl nos da una forma rápida de calcular la aceleración, que depende de dos factores:

1. La fracción del tiempo de cálculo de la máquina original que pueda utilizarse para aprovechar la mejora. En el ejemplo anterior la fracción es $\frac{50}{70}$. Este valor, que llamaremos $\text{Fracción}_{\text{mejorada}}$, es siempre menor o igual que 1.
2. La optimización lograda por el modo de ejecución mejorado; es decir, cuánto más rápido con la que se ejecutaría la tarea si *solamente* se utilizase el modo mejorado. En el ejemplo anterior este valor aparece en la columna etiquetada «aceleración en el desierto». Este valor es el tiempo del modo original con respecto al tiempo del modo mejorado y es siempre mayor que 1. Llamaremos a este valor $\text{Aceleración}_{\text{mejorada}}$.

El tiempo de ejecución utilizando la máquina original con el modo mejorado será el tiempo empleado utilizando la parte no mejorada de la máquina más el tiempo empleado utilizando la parte mejorada.

Vehículo para la segunda parte del viaje	Horas de la segunda parte del viaje	Aceleración en el desierto	Horas del viaje completo	Aceleración en el viaje completo
A pie	50,00	1,0	70,00	1,0
Bicicleta	20,00	2,5	40,00	1,8
Excel	4,00	12,5	24,00	2,9
Testarossa	1,67	30,0	21,67	3,2
Vehículo oruga	0,33	150,0	20,33	3,4

FIGURA 1.2 Las relaciones de aceleración obtenidas para los diferentes medios de transporte depende fuertemente de que hay que cruzar a pie las montañas. La aceleración en el desierto —una vez que se han atravesado las montañas— es igual a la velocidad utilizando el vehículo designado dividido por la velocidad a pie; la columna final muestra la rapidez del viaje completo cuando se compara con el viaje a pie.

$$\text{Tiempo de ejecución}_{\text{nuevo}} = \text{Tiempo de ejecución}_{\text{antiguo}} \cdot$$

$$\cdot \left((1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}} \right)$$

La aceleración global es la relación de los tiempos de ejecución:

$$\text{Aceleración}_{\text{global}} = \frac{\text{Tiempo de ejecución}_{\text{antiguo}}}{\text{Tiempo de ejecución}_{\text{nuevo}}} =$$

$$= \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}}}$$

Ejemplo

Suponer que estamos considerando una mejora que corra diez veces más rápida que la máquina original, pero sólo es utilizable el 40 por 100 del tiempo. ¿Cuál es la aceleración global lograda al incorporar la mejora?

Respuesta

$$\text{Fracción}_{\text{mejorada}} = 0{,}4$$

$$\text{Aceleración}_{\text{mejorada}} = 10$$

$$\text{Aceleración}_{\text{global}} = \frac{1}{0{,}6 + \frac{0{,}4}{10}} = \frac{1}{0{,}64} \approx 1{,}56$$

La Ley de Amdahl expresa la ley de rendimientos decrecientes: la mejora incremental en la aceleración conseguida por una mejora adicional en el rendimiento de una parte del cálculo disminuye tal como se van añadiendo mejoras. Un corolario importante de la Ley de Amdahl es que si una mejora sólo es utilizable por una fracción de una tarea, no podemos aumentar la velocidad de la tarea más que el recíproco de 1 menos esa fracción.

Un error común al aplicar la Ley de Amdahl es confundir «fracción de tiempo convertido para utilizar una mejora» y «fracción de tiempo después de que se utiliza la mejora». Si, en lugar de medir el tiempo que podría **utilizar** la mejora en un cálculo, midiésemos el tiempo **después** que se ha utilizado la mejora, los resultados serían incorrectos. (Intentar el Ejercicio 1.8 para cuantificar el error.)

La Ley de Amdahl puede servir como guía para ver cómo una mejora aumenta el rendimiento y cómo distribuir los recursos para mejorar la relación coste/rendimiento. El objetivo, claramente, es emplear recursos de forma proporcional al tiempo que se requiere en cada parte.

Ejemplo

Supongamos que se quiere mejorar la velocidad de la CPU de nuestra máquina en un factor de cinco (sin afectar al rendimiento de E/S) por cinco veces el coste. Supongamos también que la CPU se utiliza el 50 por 100 del tiempo, y que el tiempo restante la CPU está esperando las E/S. Si la CPU supone un tercio del coste total del computador, ¿el incremento de la velocidad de la CPU en un factor de cinco es una buena inversión desde un punto de vista coste/rendimiento?

Respuesta

La aceleración obtenida es

$$\text{Aceleración} = \frac{1}{0{,}5 + \frac{0{,}5}{5}} = \frac{1}{0{,}6} = 1{,}67$$

La nueva máquina costaría

$$\frac{2}{3} \cdot 1 + \frac{1}{3} \cdot 5 = 2{,}33 \text{ veces la máquina original}$$

Como el incremento de coste es mayor que la mejora de rendimiento, este cambio no mejora la relación coste/rendimiento.

Localidad de referencia

Aunque la Ley de Amdahl es un teorema que se aplica a cualquier sistema, otras observaciones importantes provienen de las propiedades de los programas. La propiedad más importante, que regularmente explotamos de un programa, es la *localidad de referencia:* los programas tienden a reutilizar los datos e instrucciones que han utilizado recientemente. Una regla empírica, muy corroborada, es que un programa emplea el 90 por 100 de su tiempo de ejecución en sólo el 10 por 100 del código. Una implicación de la localidad es que, basándose en el pasado reciente del programa, se puede predecir con una precisión razonable qué instrucciones y datos utilizará un programa en el futuro próximo.

Para examinar la localidad, se midieron algunos programas para determinar qué porcentaje de las instrucciones eran responsables del 80 y 90 por 100 de las instrucciones ejecutadas. Los datos se muestran en la Figura 1.3, y los programas se describen con detalle en el capítulo siguiente.

La localidad de referencia también se aplica a los accesos a los datos, aunque no tan fuertemente como a los accesos al código. Se han observado dos tipos diferentes de localidad. La *localidad temporal* específica que los elementos accedidos recientemente, probablemente, serán accedidos en un futuro próximo. La Figura 1.3 muestra un efecto de la localidad temporal. La *localidad espacial* establece que los elementos cuyas direcciones son próximas tienden a ser referenciados juntos en el tiempo. Veremos estos principios aplicados más tarde en este capítulo, y ampliamente en el Capítulo 8.

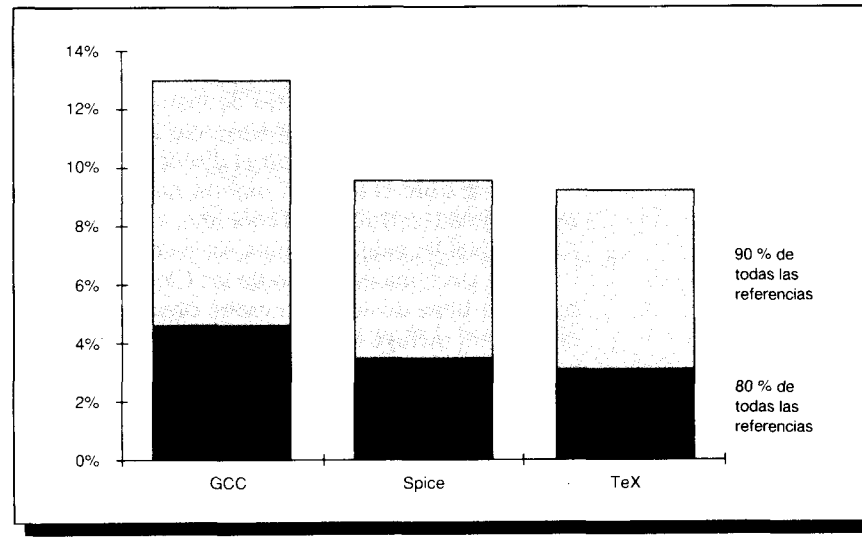


FIGURA 1.3 Este dibujo muestra el porcentaje de las instrucciones que son responsables del 80 y 90 por 100 de las ejecuciones de instrucciones. Por ejemplo, menos del 4 por 100 de las instrucciones del programa Spice (llamadas también instrucciones *estáticas)* representan el 80 por 100 de las instrucciones dinámicamente ejecutadas, mientras que menos del 10 por 100 de las instrucciones estáticas contabilizan el 90 por 100 de las instrucciones ejecutadas. Menos de la mitad de las instrucciones estáticas se ejecutan al menos una vez en cualquier ejecución —en Spice sólo el 30 por 100 de las instrucciones se ejecutan una o más veces. Descripciones detalladas de los programas y sus entradas se dan en la Figura 2.17 (pág. 71).

1.4 El trabajo de un diseñador de computadores

Un arquitecto de computadores diseña máquinas para ejecutar programas. La tarea de diseñar un computador presenta muchos aspectos, entre los que se incluyen el diseño del repertorio de instrucciones, la organización funcional, el diseño lógico y la implementación. La implementación puede abarcar el diseño de circuitos integrados (IC), encapsulamiento, potencia y disipación térmica. Habría que optimizar el diseño de la máquina en estos niveles. Esta optimización requiere estar familiarizado con un amplio rango de tecnologías, desde los compiladores y sistemas operativos al diseño lógico y encapsulamiento.

Algunas personas utilizan el término *arquitectura de computadores* para denominar solamente el diseño del repertorio de instrucciones. Los demás aspectos del diseño los referencian como «implementación», insinuando, con frecuencia, que la implementación no es interesante o es menos estimulante.

Los autores piensan que este punto de vista, además de ser incorrecto, es también responsable de los errores que se cometen en el diseño de nuevos repertorios de instrucciones. El trabajo del arquitecto o diseñador va mucho más allá del diseño del repertorio de instrucciones, y los obstáculos técnicos que surgen en otros aspectos del proyecto son ciertamente tan sugestivos como los que se encuentran al realizar el diseño del repertorio de instrucciones.

En este libro el término *arquitectura a nivel lenguaje máquina* se refiere al repertorio de instrucciones concreto, visibles por el programador. La arquitectura a nivel lenguaje máquina sirve como frontera entre «software» y «hardware», y ese tema es el foco de los Capítulo 3 y 4. La implementación de una máquina tiene dos componentes: organización y «hardware». El término *organización* incluye los aspectos de alto nivel del diseño de un computador, tal como sistema de memoria, estructura del bus y diseño interno de la CPU. Por ejemplo, dos máquinas con la misma arquitectura a nivel lenguaje máquina, pero con organizaciones diferentes, son la VAX-11/780 y la VAX 8600. El término *Hardware* se utiliza para referenciar las cosas específicas de una máquina. Esto incluye el diseño lógico detallado y la tecnología de encapsulamiento de la máquina. Este libro hace énfasis en la arquitectura a nivel lenguaje máquina y en la organización. Dos máquinas, con idénticas arquitecturas a nivel lenguaje máquina y organizaciones casi idénticas, que se diferencian principalmente a nivel hardware, son la VAX-11/780 y la 11/785; la 11/785 utilizó una tecnología de circuitos integrados mejorada para conseguir mayor frecuencia de reloj y presentaba pequeños cambios en el sistema de memoria. En este libro la palabra «arquitectura» está pensada para cubrir los tres aspectos del diseño de un computador.

Requerimientos funcionales

Los arquitectos de computadores deben diseñar un computador que cumpla ciertos requerimientos funcionales con determinadas ligaduras de precio y rendimiento; con frecuencia, también tienen que determinar los requerimientos funcionales y, en este caso, puede ser una tarea de gran magnitud. Los requerimientos pueden ser características específicas, inspiradas por el mercado. El software de aplicaciones conduce, con frecuencia, a la elección de ciertos requerimientos funcionales, al determinar cómo se utilizará la máquina. Si existe un gran cuerpo de software para cierta arquitectura a nivel lenguaje máquina, el arquitecto puede decidir que una nueva máquina implemente un repertorio de instrucciones ya existente. La presencia de un gran mercado, para una clase particular de aplicaciones, podría estimular a los diseñadores a incorporar requerimientos que hagan a la máquina competitiva en ese mercado. La Figura 1.4 (vér pág. 15) resume algunos de los requerimientos que son necesarios tener en cuenta a la hora de diseñar una nueva máquina. Muchos de estos requerimientos y características se examinarán en profundidad en los capítulos siguientes.

Muchos de los requerimientos de la Figura 1.4 representan un mínimo nivel de soporte. Por ejemplo, los sistemas operativos modernos utilizan memoria virtual y protección de memoria. Este requerimiento establece un mínimo nivel de soporte, sin el cual la máquina no sería viable. Cualquier

Requerimientos funcionales	Características típicas requeridas o soportadas
Area de aplicación	Objetivo del computador.
Próposito especial	Rendimiento más alto para aplicaciones específicas (Cap. 10).
Propósito general	Rendimiento equilibrado para un rango de tareas.
Científica	Punto flotante de alto rendimiento (Apéndice A).
Comercial	Soporte para COBOL (aritmética decimal), soporte para bases de datos y tratamiento de transacciones.
Nivel de compatibilidad software	Determina la cantidad de software existente para la máquina (Cap. 10).
En lenguaje de programación	Más flexible para el diseñador, necesita nuevo compilador.
Código objeto o binario compatible	La arquitectura está completamente definida —poca flexibilidad—, pero no necesita inversión en software ni en portar programas.
Requerimientos del sistema operativo (SO)	Características necesarias para soportar el SO escogido.
Tamaño del espacio de direcciones	Característica muy importante (Cap. 8); puede limitar aplicaciones.
Gestión de memoria	Requerida para SO modernos; puede ser plana, paginada, segmentada (Cap. 8).
Protección	Diferentes SO y necesidades de aplicación: protección de páginas frente a protección de segmentos (Cap. 8).
Cambio de contexto	Requerido para interrumpir y recomenzar un programa; el rendimiento varía (Cap. 5).
Interrupciones y traps	Tipos de soporte impactan sobre el diseño hardware y SO (Cap. 5).
Estándares	Ciertos estándares pueden ser requeridos por el mercado.
Punto flotante	Formato y aritmética: IEEE, DEC, IBM (Apéndice A).
Bus de E/S	Para dispositivos de E/S: VME, SCSI; NuBus, Futurebus (Cap. 9).
Sistemas operativos	UNIX, DOS o patente del vendedor.
Redes	Soporte requerido para distintas redes: Ethernet, FDDI (Cap. 9).
Lenguajes de programación	Lenguajes (ANSI C, FORTRAN 77, ANSI COBOL) afectan al repertorio de instrucciones.

FIGURA 1.4 Resumen de algunos de los requerimientos funcionales más importantes con que se enfrenta un arquitecto. *La columna de la izquierda describe la clase de requerimiento, mientras que la columna* de la derecha da ejemplos de características específicas que pueden ser necesarias. Examinaremos estos requerimientos de diseño con más detalle en capítulos posteriores.

hardware adicional por encima de tales mínimos puede ser evaluado desde el punto de vista del coste/rendimiento.

La mayoría de los atributos de un computador —soporte hardware para diferentes tipos de datos, rendimiento de diferentes funciones, etc.— pueden ser evaluados en base al coste/rendimiento para el mercado pensado. La siguiente sección explica cómo deben tenerse en cuenta estas cuestiones.

Equilibrar software y hardware

Una vez que se ha establecido un conjunto de requerimientos funcionales, el arquitecto debe intentar de optimizar el diseño. Que el diseño elegido sea óptimo, depende, por supuesto, de la métrica elegida. Las métricas más comunes involucran coste y rendimiento. Dado algún dominio de aplicaciones, se puede intentar cuantificar el rendimiento de la máquina por un conjunto de programas que se escogen para representar ese dominio de aplicaciones. (Veremos cómo medir el rendimiento y qué aspectos afectan al coste y precio en el siguiente capítulo.) Otros requerimientos medibles pueden ser importantes en algunos mercados; la fiabilidad y tolerancia a fallos son, con frecuencia, cruciales en los entornos de tratamiento de transacciones.

A lo largo de este texto haremos énfasis en la optimización de coste/rendimiento de la máquina. Esta optimización es, en gran parte, una respuesta a la pregunta de ¿cómo se implementa mejor una funcionalidad requerida? Las implementaciones hardware y software de una determinada característica tienen diferentes ventajas. Las ventajas principales de una implementación software son el bajo coste de los errores, más fácil diseño, y actualización más simple. El hardware ofrece el rendimiento como una única ventaja, aunque las implementaciones hardware no son siempre más rápidas —un algoritmo superior en software puede superar un algoritmo inferior implementado en hardware. El equilibrio entre hardware y software nos llevará a la mejor máquina para las aplicaciones de interés.

A veces, un requerimiento específico puede necesitar, efectivamente, la inclusión de soporte hardware. Por ejemplo, una máquina que vaya a ejecutar aplicaciones científicas con cálculos intensivos en punto flotante necesitará con toda seguridad hardware para las operaciones en punto flotante. Esto no es una cuestión de funcionalidad, sino de rendimiento. Podría utilizarse software basado en punto flotante, pero es tan lento que esa máquina no sería competitiva. El punto flotante soportado en hardware es, de hecho, el requerimiento para el mercado científico. Por comparación, considérese la construcción de una máquina para soportar aplicaciones comerciales escritas en COBOL. Tales aplicaciones utilizan frecuentemente operaciones con decimales y cadenas; por tanto, muchas arquitecturas han incluido instrucciones para estas funciones. Otras máquinas han soportado estas funciones utilizando una combinación de software y operaciones sobre enteros y lógicas «estándares». Esto es un ejemplo clásico de un compromiso entre implementación hardware y software, y no hay una solución única correcta.

Al elegir entre dos diseños, un factor que un arquitecto debe considerar es el de su complejidad. Los diseños complejos necesitan más tiempo de realización, prolongando el tiempo de aparición en el mercado. Esto significa que

un diseño que emplee más tiempo necesitará tener mayor rendimiento para que sea competitivo. En general, es más fácil tratar con la complejidad en software que en hardware, sobre todo porque es más fácil depurar y cambiar el software. Por eso, los diseñadores pueden tratar de desplazar la funcionalidad del hardware al software. Por otra parte, las decisiones en el diseño de la arquitectura a nivel lenguaje máquina y en la organización pueden afectar a la complejidad de la implementación así como a la complejidad de los compiladores y sistemas operativos de la máquina. El arquitecto siempre debe ser consciente del impacto que el diseño que elija tendrá sobre el tiempo de diseño del hardware y del software.

Diseñar para perdurar a nuevas tendencias

Si una arquitectura ha de tener éxito, debe ser diseñada para que sobreviva a los cambios en la tecnología hardware, tecnología software y aplicaciones características. El diseñador debe ser consciente, especialmente de las tendencias en la utilización del computador y de la tecnología de los computadores. Después de todo, una nueva arquitectura a nivel lenguaje máquina que tenga éxito puede durar decenas de años —el núcleo de la IBM 360 ha sido utilizado desde 1964. El arquitecto debe planificar para que los cambios tecnológicos puedan incrementar con éxito la vida de una máquina.

Para planificar la evolución de una máquina, el diseñador debe ser especialmente consciente de los rápidos cambios que experimentan las tecnologías de implementación. La Figura 1.5 muestra algunas de las tendencias más importantes en las tecnologías del hardware. Al escribir este libro, el énfasis está en los principios de diseño que pueden ser aplicados con las nuevas tecnologías y considerando las futuras tendencias tecnológicas.

Estos cambios tecnológicos no son continuos sino que, con frecuencia, se presentan en pasos discretos. Por ejemplo, los tamaños de las DRAM (memoria dinámica de acceso aleatorio) aumentan siempre en factores de 4, debido a la estructura básica de diseño. Entonces, en lugar de duplicarse cada uno o dos años, la tecnología DRAM se cuadruplica cada tres o cuatro años. Estos cambios en la tecnología llevan a situaciones que pueden habilitar una técnica de implementación que, anteriormente, era imposible. Por ejemplo, cuando la tecnología MOS logró ubicar entre 25 000 y 50 000 transistores en un único circuito integrado, fue posible construir un microprocesador de 32 bits en una sola pastilla integrada. Al eliminar los cruces entre chips dentro de la CPU, fue posible un enorme incremento en el coste/rendimiento. Este diseño era sencillamente impensable hasta que la tecnología alcanzó un cierto grado. Estos umbrales en la tecnología no son raros y tienen un impacto significativo en una amplia variedad de decisiones de diseño.

El arquitecto también necesitará estar al corriente de las tendencias en software y de cómo los programas utilizarán la máquina. Una de las tendencias software más importantes es la creciente cantidad de memoria utilizada por los programas y sus datos. La cantidad de memoria necesaria por el programa medio ha crecido en un factor de 1,5 a 2 por año. Esto se traduce en un consumo de bits de direcciones a una frecuencia de 1/2 bit a 1 bit por año. Subestimar el crecimiento del espacio de direcciones es, con frecuencia, la ra-

Tecnología	Tendencias de rendimiento y densidad
Tecnología de CI lógicos	El número de transistores en un chip aumenta aproximadamente el 25 por 100 por año, duplicándose en tres años. La velocidad de los dispositivos aumenta casi a esa rapidez.
DRAM semiconductora	La densidad aumenta en un 60 por 100 por año, cuadruplicándose en tres años. La duración del ciclo ha mejorado muy lentamente, decreciendo aproximadamente una tercera parte en diez años.
Tecnología de disco	La densidad aumenta aproximadamente el 25 por 100 por año, duplicándose en tre años. El tiempo de acceso ha mejorado un tercio en diez años.

FIGURA 1.5 Las tendencias en las tecnologías de implementación de los computadores muestran los rápidos cambios con que los diseñadores deben enfrentarse. Estos cambios pueden tener un impacto dramático sobre los diseñadores cuando afectan a decisiones a largo plazo, como la arquitectura a nivel lenguaje máquina. El coste por transistor para la lógica y el coste por bit para la memoria de disco o de semiconductores disminuye muy próximamente a la velocidad a la cual aumenta la densidad. Las tendencias en los costes se consideran con más detalle en el capítulo siguiente. En el pasado, la tecnología de las DRAM (memoria dinámica de acceso aleatorio) ha mejorado más rápidamente que la tecnología de la lógica. Esta diferencia es debida a las reducciones en el número de transistores por celda DRAM y a la creación de tecnología especializada para las DRAM. Como la mejora de estas fuentes disminuya, la densidad creciente en la tecnología de la lógica y en la de memorias llegarán a ser comparables.

zón principal por la que una arquitectura a nivel lenguaje máquina debe ser abandonada. (Para una discusión posterior, ver el Capítulo 8 sobre jerarquía de memoria.)

Otra tendencia software importante en los últimos veinte años, ha sido la sustitución del lenguaje ensamblador por los lenguajes de alto nivel. Esta tendencia ha jugado un papel importante para los compiladores y en el redireccionamiento de arquitecturas hacia el soporte del compilador. La tecnología de compiladores ha ido mejorando uniformemente. Un diseñador debe comprender esta tecnología y la dirección en la cual está evolucionando, ya que los compiladores se han convertido en la interfaz principal entre el usuario y la máquina. Hablaremos de los efectos de la tecnología de compiladores en el Capítulo 3.

Un cambio fundamental en la forma de programar puede demandar cambios en una arquitectura para soportar, eficientemente, el modelo de programación. Pero la emergencia de nuevos modelos de programación se presenta a una velocidad mucho más lenta que las mejoras en la tecnología de compiladores: en oposición a los compiladores, que mejoran anualmente, los cambios significativos en los lenguajes de programación se presentan una vez por década.

Cuando un arquitecto ha comprendido el impacto de las tendencias hardware y software en el diseño de la máquina, entonces puede considerar la pregunta de cómo equilibrar la máquina. ¿Cuánta memoria es necesario planificar para conseguir la velocidad elegida de la CPU? ¿Cuántas E/S se necesitarán? Para intentar dar alguna idea de cómo constituir una máquina equilibrada, Case y Amdahl acuñaron dos reglas empíricas que, habitualmente, se utilizan juntas. La regla establece que una máquina de 1-MIPS *(millón de instrucciones por segundo)* está equilibrada cuando tiene 1 megabyte de memoria y una productividad de E/S de 1 megabit por segundo. Esta regla proporciona un punto de partida razonable para diseñar un sistema equilibrado, pero debe ser refinada al medir el rendimiento de la máquina cuando esté ejecutando las aplicaciones pensadas.

1.5 Juntando todo: el concepto de jerarquía de memoria

En las secciones «Juntando todo» que aparecen casi al final de cada capítulo, mostramos ejemplos reales que utilizan los principios expuestos en ese capítulo. En este primer capítulo, explicamos una idea clave de los sistemas de memoria que será nuestro único foco de atención en el Capítulo 8.

Para comenzar esta sección, echemos un vistazo a un simple axioma del diseño hardware: *más pequeño es más rápido.* Las partes más pequeñas de hardware, generalmente, son más rápidas que las más grandes. Este sencillo principio es aplicable, particularmente, a las memorias, por dos razones diferentes. Primero, en las máquinas de alta velocidad, la propagación de la señal es una causa importante de retardo; las memorias más grandes tienen más retardo de señal y necesitan más niveles para decodificar las direcciones. Segundo, en muchas tecnologías se pueden obtener memorias más pequeñas, que son más rápidas que memorias más grandes. Esto es así, básicamente, porque el diseñador puede utilizar más potencia por celda de memoria en un diseño más pequeño. Las memorias más rápidas están, generalmente, disponibles en números más pequeños de bits por chip, en cualquier instante de tiempo, pero con un coste sustancialmente mayor por byte.

Incrementar el ancho de banda de la memoria y decrementar la latencia de acceso a memoria son cruciales para el rendimiento del sistema, y muchas de las técnicas de organización que explicamos se enfocarán sobre estas dos métricas. ¿Cómo podemos mejorar estas dos medidas? La respuesta se encuentra al combinar los principios explicados en este capítulo junto con la regla: más pequeño es más rápido.

El principio de localidad de referencia dice que el dato más recientemente utilizado, probablemente, será accedido de nuevo en el futuro próximo. Favorecer los accesos a estos datos mejorará el rendimiento. Por ello, trataremos de encontrar elementos recientemente accedidos en la memoria más rápida. Debido a que las memorias más pequeñas son más rápidas, queremos utilizar memorias más pequeñas para que los elementos más recientemente accedidos

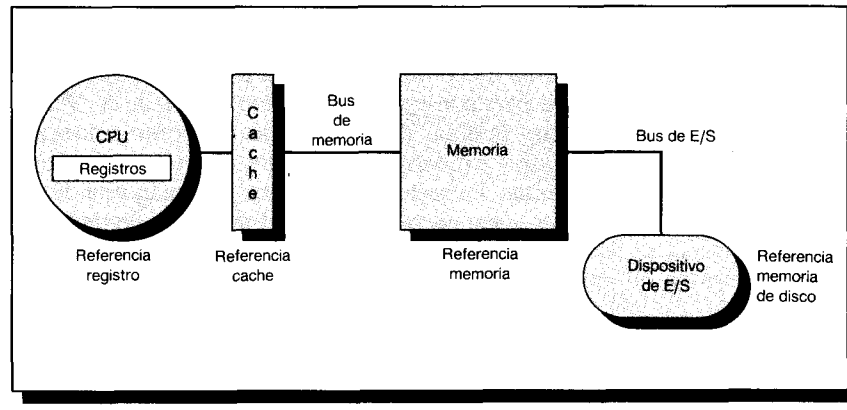


FIGURA 1.6 Niveles de una jerarquía típica de memoria. Cuando nos alejamos de la CPU, la memoria del nivel se hace más grande y más lenta.

estén próximos a la CPU y memorias sucesivamente mayores (y más lentas) cuando nos alejamos de la CPU. Este tipo de organización se denomina *jerarquía de memoria.* En la Figura 1.6 se muestra una jerarquía de memoria multinivel. Dos niveles importantes de la jerarquía de memoria son la memoria cache y la memoria virtual.

Una *cache* es una memoria pequeña y rápida localizada cerca de la CPU que contiene la mayor parte del código o de los datos recientemente accedidos. Cuando la CPU no encuentra un dato que necesita en la cache, se produce una *fallo de cache (cache miss),* y el dato se recupera de la memoria principal y se ubica en la cache. Esto, habitualmente, hace que se detenga la CPU hasta que el dato esté disponible.

De igual forma, no todos los objetos referenciados por un programa necesitan residir en memoria principal. Si el computador tiene *memoria virtual,* entonces algunos objetos pueden residir en el disco. El espacio de direcciones, normalmente, está descompuesto en bloques de tamaño fijo, denominados *páginas.* En cada instante, cada página reside o en la memoria principal o en el disco. Cuando la CPU referencia un elemento de una página que no está presente en la memoria cache o en la principal, se produce un *fallo de página (page fault),* y la página completa es transferida del disco a la memoria principal. Las memorias cache y principal tienen entre sí la misma relación que la memoria principal y el disco.

Tamaños típicos de cada nivel, en la jerarquía de memoria, y sus tiempos de acceso se muestran en la Figura 1.7. Aunque las memorias de disco y principal son habitualmente configurables, el número de registros y el tamaño de la cache, normalmente, se fijan para una implementación. La Figura 1.8 muestra estos valores para tres máquinas que se explican en este texto.

Debido a la localidad y a la mayor velocidad de las memorias más pequeñas, una jerarquía de memoria puede mejorar sustancialmente el rendimiento.

Nivel	1	2	3	4
Denominación	Registros	Cache	Memoria principal	Memoria de disco
Tamaño típico	< 1 KB	< 512 KB	< 512 MB	> 1 GB
Tiempo de acceso (en ns)	10	20	100	20 000 000
Ancho de banda (en MB/s)	800	200	133	4
Gestionado por	Compilador	Hardware	Sistema operativo	Sistema operativo/ usuario
Respaldado por	Cache	Memoria principal	Disco	Cinta

FIGURA 1.7 Los niveles típicos en la jerarquía ralentizan y son mayores cuando nos alejamos de la CPU. Los tamaños son típicos para una gran estación de trabajo o minicomputador. El tiempo de acceso se da en nanosegundos. El ancho de banda se da en MB por segundo, suponiendo caminos de 32 bits entre los niveles de la jerarquía de memoria. Cuando nos desplazamos a niveles inferiores de la jerarquía, el tiempo de acceso aumenta, haciendo factible gestionar menos sensiblemente las transferencias. Los valores mostrados son típicos en 1990 y, sin duda, cambiarán en el tiempo.

Máquina	Tamaño de registro	Tiempo de acceso a registro	Tamaño de cache	Tiempo de acceso a cache
VAX 11/780	16 32-bits	100 ns	8 KB	200 ns
VAXstation 3100	16 32-bits	40 ns	1 KB en la pastilla 64 KB fuera de la pastilla	125 ns
DECstation 3100	32-enteros 32-bits 16-punto flotante 64-bits	30 ns	64 KB instrucciones; 64 KB datos	60 ns

FIGURA 1.8 Tamaños y tiempos de acceso para los niveles de cache y registro de la jerarquía varían dramáticamente entre tres máquinas diferentes.

Ejemplo

Supongamos un computador con una memoria pequeña, de alta velocidad, que contenga 2 000 instrucciones. Suponer que el 10 por 100 de las instrucciones son responsables del 90 por 100 de los a instrucciones y que los accesos a ese 10 por 100 son uniformes. (Es decir, cada una de las instrucciones, del 10 por 100 más utilizado, se ejecuta un número igual de veces.) Si tenemos un programa con 50 000 instrucciones y sabemos qué 10 por 100 del programa se utiliza más intensamente, ¿qué fracción de los accesos a las instrucciones puede colocarse en la memoria de alta velocidad?

Respuesta

El 10 por 100 de 50 000 es 5 000. Por consiguiente, podemos colocar los 2/5 del 90 por 100 o el 36 por 100 de las instrucciones buscadas.

¿Es muy significativo el impacto de la jerarquía de memoria? Consideremos un ejemplo simplificado para ilustrar su impacto. Aunque evaluaremos las jerarquías de memoria de una forma mucho más precisa en el Capítulo 8, este ejemplo rudimentario ilustra el impacto potencial.

Ejemplo

Supongamos que una cache es cinco veces más rápida que la memoria principal, y supongamos que la cache puede ser utilizada el 90 por 100 del tiempo. ¿Qué aumento de velocidad se logrará al utilizar la cache?

Respuesta

Esto es una simple aplicación de la Ley de Amdahl.

$$\text{Aceleración} = \frac{1}{(1 - \%\text{ de tiempo que la cache puede ser usada}) + \dfrac{\%\text{ de tiempo que la cache puede ser usada}}{\text{Aceleración al utilizar la cache}}}$$

$$\text{Aceleración} = \frac{1}{(1 - 0{,}9) + \dfrac{0{,}9}{5}}$$

$$\text{Aceleración} = \frac{1}{0{,}28} \approx 3{,}6$$

Por consiguiente, obtenemos una aceleración de aproximadamente 3,6.

1.6 Falacias y pifias

El propósito de esta sección, que se encontrará en cada capítulo, es explicar algunas creencias o conceptos erróneos comúnmente mantenidos. A estas creencias erróneas las denominamos *falacias.* Cuando explicamos una falacia, tratamos de dar un contraejemplo. También explicamos *pifias (pitfalls)* —errores fácilmente cometidos. Con frecuencia, las pifias son generalizaciones de principios que son ciertos en un contexto limitado. El propósito de estas secciones es ayudar a evitar que el lector cometa estos errores en las máquinas que diseñe.

Pifia: ignorar el inexorable progreso del hardware cuando se planifica una nueva máquina.

Supongamos que se planea introducir una máquina en tres años, y se pretende que la máquina se venda muy bien porque es dos veces más rápida que ninguna de las actualmente disponibles. Desgraciadamente, hay muchas posibilidades de que la máquina no se venda bien, porque la velocidad de cre-

cimiento del rendimiento en la industria producirá máquinas del mismo rendimiento. Por ejemplo, suponiendo un crecimiento anual del 25 por 100 en el rendimiento, una máquina de rendimiento x puede esperarse que tenga un rendimiento $1{,}25^3x = 1{,}95x$ en tres años. ¡Su máquina no tendría esencialmente ninguna ventaja de rendimiento! En las compañías de computadores, muchos proyectos se cancelan porque no prestan atención a esta regla, o porque el proyecto se retrasa y el rendimiento de la máquina retardada estará por debajo de la media industrial. Aunque este fenómeno puede ocurrir en cualquier industria, las rápidas mejoras en coste/rendimiento convierten esto en un asunto importante en la industria de computadores.

Falacia: el hardware es siempre más rápido que el software.

Aunque una implementación hardware de una característica bien definida y necesaria sea más rápida que una implementación software, la funcionalidad proporcionada por el hardware es, con frecuencia, más general que las necesidades del software. Así, un compilador puede seleccionar una secuencia de instrucciones más sencillas que realicen el trabajo requerido de forma más eficiente que la instrucción hardware más general. Un buen ejemplo es la instrucción MVC (transferir carácter) en la arquitectura del IBM 360. Esta instrucción es muy general y transfiere hasta 256 bytes de datos entre dos direcciones arbitrarias. La fuente y el destino pueden comenzar en cualquier dirección —y pueden incluso solaparse. En el peor caso, el hardware debe transferir cada vez un byte; determinar la existencia del peor caso requiere un análisis considerable cuando se decodifica la instrucción.

Debido a que la instrucción MVC es muy general, incurre en un gasto que, con frecuencia, es innecesario. Una implementación software puede ser más rápida si puede eliminar este gasto. Las medidas han mostrado que las transferencias no solapadas son 50 veces más frecuentes que las solapadas y que la transferencia media no solapada es solamente de 8 bytes. De hecho, más de la mitad de las transferencias no solapadas transfieren únicamente un solo byte. Una secuencia de dos instrucciones que cargue un byte en un registro y lo almacene después en memoria es, como mínimo, dos veces más rápida que MVC cuando transfiere un solo byte. Esto ilustra la regla de hacer rápido el caso frecuente.

1.7 Observaciones finales

La tarea del diseñador de computadores es compleja: determinar qué atributos son importantes para una nueva máquina, después, diseñar una máquina para maximizar el rendimiento cumpliendo las restricciones de coste. El rendimiento puede ser medido como productividad o como tiempo de respuesta; debido a que algunos entornos favorecen una medida sobre otra, esta distinción debe tenerse en cuenta cuando se evalúan alternativas. La Ley de Amdahl es una herramienta valiosa para ayudar a determinar el aumento de rendimiento que puede tener una arquitectura mejorada. En el capítulo siguiente, examinaremos cómo medir el rendimiento y qué propiedades tienen mayor impacto sobre el coste.

Saber los casos que son más frecuentes es crítico para mejorar el rendimiento. En los Capítulo 3 y 4, examinaremos el diseño y uso de un repertorio de instrucciones, examinando las propiedades comunes de la utilización del repertorio de instrucciones. Basándose en las medidas de los repertorios de instrucciones, pueden establecerse compromisos para decidir qué instrucciones son las más importantes y qué casos hay que tratar de hacer rápidos.

En los Capítulos 5 y 6 examinaremos los fundamentos de diseño de CPU, comenzando con una máquina secuencial sencilla y desplazándonos a implementaciones segmentadas *(pipelined)*. El Capítulo 7 está dedicado a aplicar estas ideas al cálculo científico de alta velocidad en la forma de máquinas vectoriales. La Ley de Amdahl será nuestra guía a lo largo del Capítulo 7.

Hemos visto cómo una propiedad fundamental de los programas —el principio de localidad— puede ayudarnos a construir computadores más rápidos, permitiéndonos hacer efectivo el uso de memorias pequeñas y rápidas. En el Capítulo 8, volveremos a las jerarquías de memoria, examinando en profundidad el diseño de la cache y el soporte para memoria virtual. El diseño de jerarquías de memoria de altas prestaciones se ha convertido en un componente clave del diseño moderno de computadores. El Capítulo 9 trata de un aspecto estrechamente relacionado —sistemas de E/S—. Como vemos, cuando se utiliza la Ley de Amdahl para evaluar la relación coste/rendimiento, no es suficiente mejorar solamente el tiempo de CPU. Para obtener una máquina equilibrada, debemos también aumentar el rendimiento de las E/S.

Finalmente, en el Capítulo 10, examinaremos las directrices actuales de la investigación enfocándolas al procesamiento paralelo. Todavía no está clara la forma en que estas ideas afectarán a los tipos de máquinas diseñadas y utilizadas en el futuro. Lo que está claro es que una aproximación experimental y empírica en el diseño de nuevos computadores será la base de un crecimiento continuado y espectacular del rendimiento.

1.8 Perspectiva histórica y referencias

Si... la historia... nos enseña algo, es que el hombre en su búsqueda del conocimiento y progreso, es firme y no puede ser frenado.

John F. Kennedy, Conferencia en la Universidad de Rice, 12 de septiembre, 1962.

Una sección de perspectivas históricas cierra cada capítulo del texto. Esta sección proporciona algunos antecedentes históricos en relación con alguna de las ideas clave presentadas en el capítulo. Los autores pueden seguir el desarrollo de una idea a través de una serie de máquinas o describir algunos proyectos importantes. Esta sección también contendrá referencias para el lector interesado en examinar el desarrollo inicial de una idea, o máquina, o interesado en lecturas adicionales.

Los primeros computadores electrónicos

J. Presper Eckert y John Mauchly en la Moore School de la Universidad de Pennsylvania construyeron el primer computador electrónico de propósito general del mundo. Esta máquina, denominada ENIAC (Electronic Numerical Integrator and Calculator), fue financiada por la Armada de Estados Unidos y estuvo operativa durante la Segunda Guerra Mundial, aunque no se dio a conocer al público hasta 1946. La ENIAC fue una máquina de propósito general utilizada para calcular tablas de fuego de artillería. Unos cien pies de largo por ocho pies y medio de alto y algunos de ancho, la máquina era enorme —mucho mayor que el tamaño de cualquier computador construido hoy día. Cada uno de los 20 registros de 10 dígitos tenía una longitud de dos pies. En total, tenía 18 000 tubos de vacío.

Aunque el tamaño era de dos órdenes de magnitud mayor que las máquinas construidas hoy día, era de tres órdenes de magnitud más lenta, en realizar una suma tardaba 200 microsegundos. La ENIAC permitía bifurcaciones condicionales y era programable, lo que la distinguía claramente de las primeras calculadoras. La programación se hacía manualmente conectando cables y pulsando interruptores. Los datos se suministraban en tarjetas perforadas. La programación para los cálculos normales requería desde media hora a un día entero. La ENIAC fue una máquina de propósito general limitada principalmente por una pequeña cantidad de memoria y programación tediosa.

En 1944, John von Neumann se incorporó al proyecto ENIAC. El grupo quería mejorar la forma en que se introducían los programas y pensaron en almacenar los programas como números; von Neumann ayudó a cristalizar las ideas y escribió un memorándum proponiendo un computador de programa almacenado denominado EDVAC (Electronic Discrete Variable Automatic Computer). Herman Goldstine distribuyó el memorándum y le puso el nombre de von Neumann, a pesar de la consternación de Eckert y Mauchly, cuyos nombres se omitieron. Este memorándum ha servido como base para el término comúnmente utilizado «computador von Neumann». Los autores y algunos de los primeros inventores en el campo de los computadores piensan que este término da demasiado crédito a von Neumann, que redactó las ideas, y muy poco a los ingenieros Eckert y Mauchly, que trabajaron en las máquinas. Por esta razón, este término no aparecerá en este libro.

En 1946, Maurice Wilkes, de la Universidad de Cambridge, visitó la Moore School para asistir a la última parte de una serie de lecturas sobre desarrollos de computadores electrónicos. Cuando volvió a Cambridge, Wilkes decidió embarcarse en un proyecto para construir un computador de programa almacenado denominado EDSAC, siglas de Electronic Delay Storage Automatic Calculator. El EDSAC llegó a estar operativo en 1949 y fue el primer computador de programa almacenado del mundo, operativo a escala completa [Wilkes, Wheeler, y Gill, 1951; Wilkes, 1985]. (Un pequeño prototipo denominado Mark I, que fue construido en la Universidad de Manchester y estuvo operativo en 1948, podría denominarse la primera máquina operativa de programa almacenado.) El EDSAC tenía una arquitectura basada en acumulador. Este estilo de máquina llegó a ser popular hasta los primeros años de la década de los setenta, y los repertorios de instrucciones parecían extraordina-

riamente similares al del EDSAC. (El Capítulo 3 comienza con un breve resumen del repertorio de instrucciones del EDSAC.)

En 1947, Eckert y Mauchly solicitaron una patente sobre computadores electrónicos. El decano de la Moore School, al pedir que la patente se traspasase a la universidad, pudo haber ayudado a Eckert y Mauchly a decidir su marcha. Su partida paralizó el proyecto EDVAC, que no llegó a estar operativo hasta 1952.

En 1946, Goldstine se unió a von Neumann en el Instituto de Estudios Avanzados de Princeton. Junto con Arthur Burks, realizaron un informe (1946) basado en el memorándum escrito anteriormente. El artículo trataba de la máquina IAS construida por Julian Bigelow en el Instituto de Estudios Avanzados de Princeton. Tenía un total de 1024 palabras de 40 bits y era aproximadamente diez veces más rápida que la ENIAC. El grupo pensó en los posibles usos de la máquina, publicó una serie de informes y animó a los visitantes. Estos informes y visitantes inspiraron el desarrollo de una serie de nuevos computadores. El artículo de Burks, Goldstine y von Neumann fue increíble para su época. Al leerlo hoy, nunca se pensaría que este artículo prominente se escribiese hace más de cuarenta años, cuando la mayor parte de los conceptos sobre arquitectura vistos en los modernos computadores se explican allí.

Recientemente ha habido cierta controversia con respecto a John Atanasoff, que construyó un computador electrónico, de pequeña escala, a principios de los años cuarenta [Atanasoff, 1940]. Su máquina, diseñada en la Universidad del Estado de Iowa, fue un computador de propósito especial que nunca llegó a ser completamente operativo. Mauchly visitó brevemente a Atanasoff antes de que se construyese el ENIAC. La presencia de la máquina de Atanasoff, junto a los retrasos en registrar las patentes del ENIAC (el trabajo fue clasificado y las patentes no pudieron ser registradas hasta después de la guerra) y la distribución del artículo EDVAC de von Neumann, fueron utilizados para acabar con la patente de Eckert-Mauchly [Larson, 1973]. Aunque la controversia todavía pone en duda el papel de Atanasoff, a Eckert y Mauchly se les reconoce generalmente por haber construido el primer computador electrónico de propósito general [Stern, 1980]. Otra máquina pionera, digna de crédito, fue la máquina de propósito especial construida por Konrad Zuse en Alemania a finales de los años treinta y principios de los cuarenta. Esta máquina era electromecánica y, debido a la guerra, nunca fue realizada completamente.

En el mismo período de tiempo que el ENIAC, Howard Aiken construyó en Harvard un computador electromecánico denominado Mark-I. A la Mark-I le siguió una máquina de relés, la Mark-II, y un par de máquinas de tubos de vacío, la Mark-III y la Mark-IV. Estas últimas fueron construidas después de las primeras máquinas de programa almacenado. Debido a que tenían memorias separadas para instrucciones y datos, las máquinas fueron consideradas como reaccionarias por los defensores de los computadores de programa almacenado. El término *arquitectura Harvard* fue acuñado para describir este tipo de máquina. Aunque claramente diferente del sentido original, este término se utiliza hoy para aplicarlo a las máquinas con una sola memoria principal pero con caches de datos e instrucciones separadas.

El proyecto Whirlwind [Redmond y Smith, 1980] se comenzó en el MIT

en 1947 y estaba dirigido a aplicaciones de tratamiento de señales de rádar en tiempo real. Aunque condujo a algunos inventos, su innovación arrolladora fue la creación de las memorias de núcleos magnéticos. El Whirlwind tenía 2 048 palabras de núcleos magnéticos de 16 bits. Los núcleos magnéticos se utilizaron como tecnología de la memoria principal al comienzo de los años treinta.

Desarrollos comerciales

En diciembre de 1947, Eckert y Mauchly formaron la Eckert-Mauchly Computer Corporation. Su primera máquina, la BINAC, fue construida por Northrop y fue presentada en agosto de 1949. Después de algunas dificultades financieras, fueron absorbidos por Remington-Rand, donde construyeron la UNIVAC I, diseñada para ser vendida como un computador de propósito general. El primer desarrollo apareció en junio de 1951; la UNIVAC I se vendió por 250 000 dólares y fue el primer computador comercial con éxito —¡se construyeron 48 sistemas! Hoy día, esta primitiva máquina, junto con otros muchos computadores populares, pueden verse en el Computer Museum de Boston, Massachusetts.

IBM, que antes se había dedicado a negocios de tarjetas perforadas y automatización de oficinas, no comenzó a construir computadores hasta 1950. El primer computador IBM, el IBM 701, fue construido en 1952 y, eventualmente, vendió 19 unidades. A principios de los años cincuenta, mucha gente era pesimista sobre el futuro de los computadores, pensando que el mercado y oportunidades para estas máquinas «altamente especializadas» eran bastante limitados.

Algunos libros que describen los primeros días de la computación fueron escritos por los pioneros [Wilkes, 1985; Goldstine, 1972]. Hay numerosas historias independientes, construidas a menudo en torno a las personas implicadas [Slater, 1987; Shurkin, 1984], así como un diario, *Annals of the History of Computing,* dedicado a la historia de la computación.

La historia de algunos de los computadores inventados después de 1960 puede encontrarse en los Capítulos 3 y 4 (el IBM 360, el DEC VAX, el Intel 80x86, y las primeras máquinas RISC), el Capítulo 6 (los procesadores segmentados, donde se incluye el CDC 6600) y el Capítulo 7 (procesadores vectoriales, donde se incluyen los procesadores TI ASC, CDC Star y Cray).

Generaciones de computadores. Un breve resumen de la historia de los computadores

Desde 1952, ha habido miles de nuevos computadores que han utilizado un amplio rango de tecnologías y ofrecido un amplio espectro de capacidades. En un intento de dar una perspectiva a los desarrollos, la industria ha intentado agrupar los computadores en generaciones. Esta clasificación, con frecuencia, está basada en la tecnología de implementación que se ha utilizado en cada generación, como se muestra en la Figura 1.9. Normalmente, cada generación de computadores es de ocho a diez años, aunque la duración y fecha de co-

Generación	Fechas	Tecnología	Nuevo producto principal	Nuevas compañías y máquinas
1	1950-1959	Tubos de vacío	Computador electrónico comercial	IBM 701, UNIVAC I
2	1960-1968	Transistores	Computadores baratos	Burroughs 6500, NCR, CDC 6600, Honeywell
3	1969-1977	Circuito integrado	Minicomputador	50 nuevas compañías: DEC PDP-11, Data General Nova
4	1978-199?	LSI y VLSI	Computadores personales y estaciones de trabajo	Apple II, Apollo DN 300, Sun 2
5	199?-	¿Procesamiento paralelo?	Multiprocesadores	??

FIGURA 1.9 Las generaciones de computadores están habitualmente determinadas por el cambio en la tecnología de implementación dominante. Normalmente, cada generación ofrece la oportunidad de crear una nueva clase de computadores y se crean nuevas compañías de computadores. Muchos investigadores piensan que el procesamiento paralelo utilizando microprocesadores de alto rendimiento será la base de la quinta generación de computadores.

mienzo —especialmente para las generaciones recientes— están a debate. Por convenio, la primera generación se toma en los computadores electrónicos comerciales, en lugar de en las máquinas mecánicas o electromecánicas que les precedieron.

Desarrollo de los principios explicados en este capítulo

Lo que quizá es el principio básico fue planteado originalmente por Amdahl [1977] y está relacionado con las limitaciones sobre la aceleración en el contexto del procesamiento paralelo:

Una conclusión equitativamente obvia que puede ser considerada en este punto es que el esfuerzo realizado para lograr altas velocidades de procesamiento paralelo es superfluo a menos que venga acompañado por logros en velocidades de procesamiento secuencial de, aproximadamente, la misma magnitud (pág. 485).

Amdahl planteó esta ley concentrándose en las implicaciones de aumentar la velocidad de sólo una parte de la computación. La ecuación básica puede ser utilizada como una técnica general para medir la aceleración y efectividad-coste de cualquier mejora.

La memoria virtual apareció primero en una máquina llamada ATLAS,

diseñada en Gran Bretaña en 1962 [Kilburn y cols., 1982]. El IBM 360/85, introducido a finales de los años sesenta, fue la primera máquina comercial que utilizó una cache, pero parece que la idea fue utilizada en varias máquinas construidas en Gran Bretaña a principios de los años sesenta (ver la discusión en el Capítulo 8).

Knuth [1971] publicó las observaciones originales sobre la localidad de los programas:

Los programas, normalmente, tienen un perfil muy desigual, con unos pocos picos agudos. Como aproximación preliminar, parece ser que la enésima sentencia más importante de un programa, desde el punto de vista del tiempo de ejecución, contabiliza aproximadamente $(a-1)a^{-n}$ *del tiempo de ejecución, para algún «a» y para algún «n» pequeño. También encontramos que menos del 4 por 100 de un programa, generalmente, representa más de la mitad de su tiempo de ejecución* (pág. 105).

Referencias

AMDAHL, G. M. [1967]. «Validity of the single processor approach to achieving large scale computing capabilities», *Proc. AFIPS 1967 Spring Joint Computer Conf. 30* (April), Atlantic City. N.J., 483-485.

ATANASOF, J. V. [1940]. «Computing machine for the solution of large systems of linear equations», Internal Report, Iowa State University.

BELL, C. G. [1984]. «The mini and micro industries», *IEEE Computer* 17:10 (October) 14-30.

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN [1946]. «Preliminary discussion of the logical design of an electronic computing instrument», Report to the U.S. Army Ordenance Department, p. 1; also appears in *Papers of John von Neumann,* W. Aspray and A. Burks, eds., The MIT Press, Cambridge, Mass. and Tomash Publishers, Los Angeles, Calif., 1987, 97-146.

GOLDSTINE, H. H. [1972]. *The Computer: From Pascal to von Neumann,* Princeton University Press, Princeton, N.J.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, AND F. H. SUMNER [1982]. «One-level storage system», reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York.

KNUTH, D. E. [1971]. «An empirical study of FORTRAN programs», *Software Practice and Experience,* Vol. 1, 105-133.

LARSON, JUDGE E. R. [1973]. Findings of Fact, Conclusions of Law, and Order for Judgment», File No. 4-67, Civ. 138, *Honeywell v. Sperry Rand and Illinois Scientific Development,* U.S. District Court for the District of Minnesota, Fourth Division (October 19).

REDMOND, K. C. AND T. M. SMITH [1980]. *Project Whirlwind–The History of a Pioneer Computer,* Digital Press, Boston, Mass.

SHURKIN, J. [1984]. *Engines of the Mind: A History of the Computer,* W. W. Norton, New York.

SLATER, R. [1987]. *Portraits in Silicon,* The MIT Press, Cambridge, Mass.

STERN, N. [1980]. «Who invented the first electronic digital computer», *Annals of the History of Computing* 2:4 (October) 375-376.

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer,* The MIT Press, Cambridge, Mass.

WILKES, M. V., D. J. WHEELER, AND S. GILL [1951]. *The Preparation of Programs for an Electronic Digital Computer,* Addison-Wesley Press, Cambridge, Mass.

EJERCICIOS

1.1 [10/10/10/12/12/12] <1.1,1.2> Se dan los tiempos de ejecución en segundos del *(benchmark)* * «Linpack» y de 10 000 iteraciones del benchmark Dhrystone» (ver Fig. 2.5, pág. 50) en algunos modelos VAX:

Modelo	Año de construcción	Tiempo de ejecución de Linpack (segundos)	Tiempo de ejecución de Dhrystone (10 000 iteracciones) (segundos)
VAX-11/780	1978	4,90	5,69
VAX 8600	1985	1,43	1,35
VAX 8550	1987	0,695	0,96

a) ¿Cuántas veces es más rápido el 8600 que el 780 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

b) [10] ¿Cuántas veces es más rápido el 8550 que el 8600 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

c) [10] ¿Cuántas veces es más rápido el 8550 que el 780 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

d) [12] ¿Cuál es el crecimiento medio del rendimiento por año entre el 780 y el 8600 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

e) [12] ¿Cuál es el crecimiento medio de rendimiento por año entre el 8600 y el 8550 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

f) [12] ¿Cuál es el crecimiento medio de rendimiento por año entre el 780 y el 8550 utilizando Linpack? ¿Qué ocurre cuando se utiliza Dhrystone?

1.2-1.5 Para las cuatro siguientes preguntas, supongamos que se está considerando mejorar una máquina añadiéndole un modo vectorial. Cuando se ejecuta un cálculo en modo vectorial, es 20 veces más rápido que en el modo normal de ejecución. Llamamos al porcentaje de tiempo que puede emplearse el modo vectorial *porcentaje de vectorización.*

1.2 [20] <1.3> Dibujar un gráfico donde se muestre la aceleración como porcentaje del cálculo realizado en modo vectorial. Rotular el eje y con «aceleración neta» y el eje x con «porcentaje de vectorización».

1.3 [10] <1.3> ¿Qué porcentaje de vectorización se necesita para conseguir una aceleración de 2?

1.4 [10] <1.3> ¿Qué porcentaje de vectorización se necesita para conseguir la mitad de la aceleración máxima alcanzable utilizando el modo vectorial?

(*) *Nota del traductor.* En este contexto, *benchmark* significa programa de referencia para medida de rendimiento o eficacia. Se utiliza para evaluar el software o hardware de un computador o ambos. También se utiliza para comparar diferentes computadores bajo determinadas características, número de instrucciones, tamaño de memoria, etc. A lo largo del texto, aparecerá repetidamente la palabra *benchmark.*

1.5 [15] <1.3> Supongamos que hemos medido el porcentaje de vectorización de programas, obteniendo que es del 70 por 100. El grupo de diseño hardware dice que puede duplicar la velocidad de la parte vectorizada con una inversión significativa de ingeniería adicional. Se desea saber si el equipo de compilación puede incrementar la utilización del modo vectorial como otra aproximación para incrementar el rendimiento. ¿Qué incremento en el porcentaje de vectorización (relativo a la utilización actual) se necesitará para obtener la misma ganancia de rendimiento? ¿Qué inversión es recomendable?

1.6 [12/12] <1.1,1.4> Hay dos equipos de diseño en dos compañías diferentes. La gestión de la compañía más pequeña y más agresiva pide un ciclo de diseño de dos años para sus productos. La gestión de la compañía más grande y menos agresiva apuesta por un ciclo de diseño de cuatro años. Supongamos que en el mercado de hoy día se demanda 25 veces el rendimiento de un VAX-11/780.

a) [12] ¿Cuáles deberían ser los objetivos de rendimiento para cada producto, si las frecuencias de crecimiento necesarias son el 30 por 100 por año?

b) [12] Supongamos que las compañías acaban de empezar a utilizar DRAM de 4 megabits. Supongamos que se mantienen las frecuencias de crecimiento de la Figura 1.5 (pág. 18), ¿qué tamaños de DRAM hay que planificar para utilizar en estos proyectos? Obsérvese que el crecimiento de las DRAM es discreto, y que en cada generación son cuatro veces mayores que en la generación anterior.

1.7 [12] <1.3> Se están considerando dos diseños alternativos para una memoria de instrucciones: utilizar chips rápidos y caros o chips más baratos y más lentos. Si se utilizan los chips lentos se puede lograr el doble de anchura del bus de memoria y la búsqueda de dos instrucciones, cada una de una palabra, cada dos ciclos de reloj. (Con los chips más rápidos y caros, el bus de memoria solamente puede buscar una palabra cada ciclo de reloj.) Debido a la localidad espacial, cuando se busquen dos palabras, con frecuencia, se necesitarán ambas. Sin embargo, en el 25 por 100 de los ciclos de reloj no se utilizará una de las dos palabras buscadas. Comparar el ancho de banda de memoria de estos dos sistemas

1.8 [15/10] <1.3> Supongamos —como en el ejemplo de la Ley de Amdahl al final de la página 11— que realizamos una mejora en un computador que aumenta el modo de ejecución en un factor de 10. El modo mejorado se utiliza el 50 por 100 del tiempo, medido como porcentaje del tiempo de ejecución cuando se utiliza el modo mejorado, en lugar de como se ha definido en este capítulo: el porcentaje del tiempo de ejecución **sin** la mejora.

a) [15] ¿Cuál es la aceleración que hemos obtenido del modo rápido?

b) [10] ¿Qué porcentaje del tiempo de ejecución original se ha convertido al modo rápido?

1.9 [15/15] <1.5> Supongamos que se está construyendo una máquina con una jerarquía de memoria para instrucciones (no preocuparse por los accesos a datos). Supongamos que el programa sigue la regla 90-10 y que los accesos en el 10 por 100 superior y el 90 por 100 inferior están uniformemente distribuidos; es decir, el 90 por 100 del tiempo se emplea sobre un 10 por 100 del código y el otro 10 por 100 del tiempo se emplea sobre el otro 90 por 100 del código. Se pueden utilizar tres tipos de memoria en la jerarquía de memoria:

Tipo de memoria	Tiempo de acceso	Costo por palabra
Local, rápida	1 ciclo de reloj	$ 0,10
Principal	5 ciclos de reloj	$ 0,01
Disco	5 000 ciclos de reloj	$ 0,0001

Se tienen exactamente 100 programas, cada uno de 1 000 000 de palabras, y todos los programas deben estar en disco. Supongamos que cada vez sólo corre un programa, y que el programa completo debe estar cargado en memoria principal. En la jerarquía de memoria se pueden gastar hasta 30 000 dólares.

a) [15] ¿Cuál es la forma óptima de distribuir el presupuesto suponiendo que cada palabra debe estar colocada estáticamente en la memoria rápida o en la memoria principal?

b) [15] Ignorando el tiempo de la primera carga del disco, ¿cuál es el número medio de ciclos para que un programa haga referencia a memoria en su jerarquía? (Esta medida importante se denomina tiempo medio de acceso a memoria en el Capítulo 8.)

1.10 [30] <1.3,1.6> Determinar una máquina que tenga una implementación rápida y otra lenta de una característica —por ejemplo, un sistema con y sin hardware de punto flotante—. Medir la aceleración obtenida cuando se utilice la implementación rápida con un bucle sencillo que utilice la característica. Encontrar un programa real que haga uso de la característica y medir la aceleración. Utilizando este dato, calcular el porcentaje del tiempo que se utiliza la característica.

1.11 [Discusión] <1.3,1.4> Con frecuencia, las ideas para aumentar la velocidad de los procesadores aprovechan algunas propiedades especiales que tienen ciertas clases de aplicaciones. Por ello, la aceleración lograda con una mejora puede estar disponible solamente en ciertas aplicaciones. ¿Cómo se decidiría realizar tal mejora? ¿Qué factores deben ser más relevantes en la decisión? ¿Podrían estos factores medirse o estimarse razonablemente?

Recordar que el tiempo es oro.

Ben Franklin, *Advertencia a los jóvenes hombres de negocios.*

2.1 Introducción

2.2 Rendimiento

2.3 Coste

2.4 Juntando todo: precio/rendimiento de tres máquinas

2.5 Falacias y pifias

2.6 Observaciones finales

2.7 Perspectiva histórica y referencias

Ejercicios

2 Rendimiento y coste

2.1 Introducción

¿Por qué los ingenieros diseñan computadores diferentes? ¿Por qué los utiliza la gente? ¿Cómo los clientes se deciden por un computador en vez de por otro? ¿Hay una base racional para estas decisiones? Si es así, ¿pueden los ingenieros utilizar esta base para diseñar mejores computadores? Estas son algunas de las preguntas planteadas en este capítulo.

Una manera de aproximarse a estas preguntas es ver cómo se han utilizado en otros campos de diseño y aplicar aquellas soluciones, por analogía, a nuestro propio campo. El automóvil, por ejemplo, puede proporcionar una fuente útil de analogías para explicar los computadores: podemos decir que las CPU son como los motores, los supercomputadores como los automóviles de carreras exóticos, y las CPU rápidas con memorias lentas son como buenos motores en chasis pobres.

Las medidas estándares del rendimiento proporcionan una base para la comparación, conduciendo a mejoras del objeto medido. Las carreras ayudaron a determinar qué vehículo y conductor eran más rápidos, pero era muy difícil separar la destreza del conductor del rendimiento del automóvil. Algunas pruebas estándares sobre rendimiento evolucionaron eventualmente, como por ejemplo:

- Tiempo hasta que el automóvil alcanza una velocidad determinada, normalmente 60 millas por hora
- Tiempo para cubrir una distancia dada, normalmente 1/4 de milla
- Velocidad tope en una superficie plana

Marca y modelo	Mes de prueba	Precio (probado)	s (0-60)	s (1/4 mi.)	Velocidad máxima	Freno (80-0)	Derrape g	Fuel MPG
Chevrolette Corvette	2-88	34 034$	6,0	14,6	158	225	0,89	17,5
Ferrari Testarossa	10-89	145 580$	6,2	14,2	181	261	0,87	12,0
Ford Escort	7-87	5 675$	11,2	18,8	95	286	0,69	37,0
Hyndai Excel	10-86	6 965$	14,0	19,4	80	291	0,73	29,9
Lamborghini Countach	3-86	118 000$	5,2	13,7	173	252	0,88	10,0
Mazda Miata	7-89	15 550$	9,2	16,8	116	270	0,83	25,5

FIGURA 2.1 Resumen cuantitativo del coste/rendimiento de automóvil. Estos datos fueron tomados del número de octubre de 1989 de *Road and Track,* pág. 26. Una sección «Road Test Summary» se encuentra en cada número de la revista.

Las medidas estándares permiten a los diseñadores seleccionar cuantitativamente entre alternativas, con lo que se logra un progreso ordenado en un campo.

Los automóviles resultaron tan populares que aparecieron revistas que alimentaban el interés por los nuevos automóviles y ayudaban a los lectores a decidir qué automóvil comprar. Aunque estas revistas tienen siempre artículos dedicados a describir las impresiones de conducir un nuevo automóvil —la experiencia cualitativa—, a lo largo del tiempo han conseguido una base cuantitativa para realizar comparaciones, como ilustra la Figura 2.1.

El rendimiento, coste de compra y coste de operación dominan estos resúmenes. Rendimiento y coste también forman la base racional para decidir qué computador seleccionar. Por tanto, los diseñadores de computadores deben comprender rendimiento y coste si quieren diseñar computadores a los que la gente considere dignas selecciones.

Igual que no hay un objetivo sencillo para los diseñadores de automóviles, tampoco lo hay para los diseñadores de computadores. En un extremo, el *diseño de alto rendimiento* no escatima costes para conseguir sus objetivos. Los supercomputadores de Cray, así como los vehículos deportivos de Ferrari y Lamborghini, caen en esta categoría. En el otro extremo está el *diseño de bajo coste,* donde se sacrifica rendimiento para conseguir un coste más bajo. Los computadores como los clónicos del IBM PC junto con sus automóviles equivalentes, como el Ford Escort y el Hyundai Excel, caen en esta categoría. Entre estos extremos está el *diseño de coste/rendimiento* donde el diseñador equilibra el coste frente al rendimiento. Ejemplos de la industria de minicomputadores o de estaciones de trabajo tipifican las clases de compromisos con los que los diseñadores de Corvette y Miata se sentirían confortables.

En este terreno medio, donde no se desprecia ni el coste ni el rendimiento, es donde centraremos nuestra atención. Comenzaremos examinando el rendimiento, la medida del ideal del diseñador, antes de describir la agenda contable —el coste.

2.2 Rendimiento

El tiempo es la medida del rendimiento del computador: el computador que realiza la misma cantidad de trabajo en el mínimo tiempo es el más rápido. El *tiempo de ejecución* de un programa se mide en segundos por programa. El rendimiento se mide frecuentemente como una frecuencia de eventos por segundo, ya que tiempo más bajo significa mayor rendimiento. Tendemos a desdibujar esta distinción y hablamos del rendimiento como tiempo o como velocidad, incluyendo refinamientos como mejora del rendimiento en lugar de utilizar los adjetivos mayor (para velocidades) o menor (para el tiempo).

Pero el tiempo se puede definir de formas distintas dependiendo de lo que queramos contar. La definición más directa de tiempo se denomina tiempo de reloj, tiempo de respuesta, o *tiempo transcurrido (elapsed time).* Esta es la latencia para completar una tarea, incluyendo accesos a disco, accesos a memoria, actividades de entrada/salida, gastos del sistema operativo —todo. Sin embargo, como en multiprogramación, la CPU trabaja sobre otro programa mientras espera las E/S y, necesariamente, puede no minimizar el tiempo transcurrido de un programa; es necesario un término que tenga en cuenta esta actividad. El tiempo de CPU reconoce esta distinción y mide el tiempo que la CPU está calculando **sin** incluir el tiempo de espera para las E/S o para ejecutar otros programas. (Obviamente, el tiempo de respuesta visto por el usuario es el tiempo transcurrido del programa, no el tiempo de CPU.) El tiempo de CPU, además, puede dividirse en el tiempo empleado por la CPU en el programa: *tiempo de CPU del usuario,* y tiempo empleado por el sistema operativo realizando tareas requeridas por el programa: *tiempo de CPU del sistema.*

Estas distinciones están reflejadas en el comando «time» de UNIX, que devuelve lo siguiente:

```
90,7u 12,9s 2:39 65%
```

El tiempo de CPU del usuario es de 90,7 segundos, el tiempo de CPU del sistema es de 12,9 segundos, el tiempo transcurrido es de 2 minutos y 39 segundos (159 segundos), y el porcentaje de tiempo transcurrido, que es el tiempo de CPU, es (90,7 + 12,9)/159 ó 65%. Más de un tercio del tiempo ocupado en este ejemplo transcurrió esperando las E/S o ejecutando otros programas, o ambas cosas. Muchas medidas ignoran el tiempo de CPU del sistema debido a la poca precisión de las automedidas de los sistemas operativos y a la injusticia de incluir el tiempo de CPU del sistema cuando se comparan rendimientos entre máquinas con códigos de sistema diferentes. Por otro lado, el código del sistema de algunas máquinas es el código del usuario en otras y ningún programa corre sin que se ejecute el sistema operativo sobre el hardware, lo que es un argumento a favor de utilizar la suma del tiempo de CPU del usuario y del tiempo de CPU del sistema.

En la discusión actual, se mantiene la distinción entre el rendimiento basado en el tiempo transcurrido y el basado en el tiempo de CPU. El término *rendimiento del sistema* se utiliza para referenciar el tiempo transcurrido en

un sistema **no cargado,** mientras que el *rendimiento de la CPU* se refiere al tiempo de CPU del **usuario.** En este capítulo nos concentraremos en el rendimiento de la CPU.

Rendimiento de la CPU

La mayoría de los computadores se construyen utilizando un reloj que funciona a una frecuencia constante. Estos eventos discretos de tiempo se denominan pulsos, pulsos de reloj, períodos de reloj, relojes, *ciclos* o *ciclos de reloj.* Los diseñadores de computadores referencian el tiempo de un período de reloj por su duración (por ejemplo, 10 ns) o por su frecuencia (por ejemplo, 100 MHz).

El tiempo de CPU para un programa puede expresarse entonces de dos formas:

Tiempo de CPU =
= Ciclos de reloj de CPU para un programa · Duración del ciclo de reloj

o

$$\text{Tiempo de CPU} = \frac{\text{Ciclos de reloj de CPU para un programa}}{\text{Frecuencia de reloj}}$$

Obsérvese que no tendría sentido mostrar el tiempo transcurrido como función de la duración del ciclo de reloj, ya que la latencia de los dispositivos de E/S, normalmente, es independiente de la frecuencia de reloj de la CPU.

Además del número de ciclos de reloj para ejecutar un programa, también podemos contar el número de instrucciones ejecutadas —la longitud del camino de instrucciones o el *recuento de instrucciones*—. Si conocemos el número de ciclos de reloj y el recuento de instrucciones podemos calcular el número medio de *ciclos de reloj por instrucción* (CPI):

$$\text{CPI} = \frac{\text{Ciclos de reloj de CPU para un programa}}{\text{Recuento de instrucciones}}$$

Esta medida del rendimiento de la CPU proporciona una nueva percepción en diferentes estilos de repertorios de instrucciones e implementaciones.

Al transponer el «recuento de instrucciones» en la fórmula anterior, los ciclos de reloj pueden definirse como recuento de instrucciones · CPI. Esto nos permite utilizar el CPI en la fórmula del tiempo de ejecución:

$$\begin{matrix}\text{Tiempo} \\ \text{de CPU}\end{matrix} = \text{Recuento de instrucciones} \cdot \text{CPI} \cdot \text{Duración del ciclo de reloj}$$

o

$$\text{Tiempo de CPU} = \frac{\text{Recuento de instrucciones} \cdot \text{CPI}}{\text{Frecuencia de reloj}}$$

Expandiendo la primera fórmula en las unidades de medida mostradas obtenemos:

$$\frac{\text{Instrucciones}}{\text{Programa}} \cdot \frac{\text{Ciclos de reloj}}{\text{Instrucción}} \cdot \frac{\text{Segundos}}{\text{Ciclo de reloj}} =$$

$$= \frac{\text{Segundos}}{\text{Programa}} = \text{Tiempo de CPU}$$

Como demuestra esta fórmula, el rendimiento de la CPU depende de tres características: ciclo de reloj (o frecuencia), ciclos de reloj por instrucción, y recuento de instrucciones. No se puede cambiar ninguna de ellas sin tener en cuenta las demás, ya que las tecnologías básicas involucradas al cambiar una característica también son interdependientes:

Frecuencia de reloj—Tecnología hardware y organización

CPI—Organización y arquitectura a nivel lenguaje máquina

Recuento de instrucciones—Arquitectura del nivel lenguaje máquina y tecnología de compiladores

A veces es útil, al diseñar la CPU, calcular el número total de ciclos de reloj de la CPU como:

$$\text{Ciclos de reloj de la CPU} = \sum_{i=1}^{n} (\text{CPI}_i \cdot \text{I}_i)$$

donde I_i representa el número de veces que se ejecuta la instrucción i en un programa y CPI_i representa el número medio de ciclos de reloj para la instrucción i. Esta forma puede utilizarse para expresar el tiempo de CPU como

$$\text{Tiempo de CPU} = \sum_{i=1}^{n} (\text{CPI}_i \cdot \text{I}_i) \cdot \text{Duración del ciclo de reloj}$$

y el CPI global como:

$$\text{CPI} = \frac{\sum_{i=1}^{n} (\text{CPI}_i \cdot \text{I}_i)}{\text{recuento de instrucciones}} = \sum_{i=1}^{n} \left(\text{CPI}_i \cdot \frac{\text{I}_i}{\text{recuento de instrucciones}} \right)$$

La última forma del cálculo de CPI multiplica cada CPI_i individual por la fracción de ocurrencias en un programa.

CPI_i debe medirse, y no calcularse a partir de una tabla al final del manual de referencia, ya que debe incluir fallos de cache y demás ineficiencias del sistema de memoria.

Tener siempre en cuenta que la medida real del rendimiento del computador es el tiempo. Cambiar el repertorio de instrucciones para disminuir el recuento de instrucciones; por ejemplo, puede conducir a una organización con un ciclo de reloj de mayor duración que contrarresta las mejoras en el

recuento de instrucciones. Cuando se comparan dos máquinas se deben examinar los tres componentes para comprender el rendimiento relativo.

Ejemplo

Suponer que estamos considerando dos alternativas para una instrucción de salto condicional;

CPU A. Una instrucción de comparación inicializa un código de condición y es seguida por un salto que examina el código de condición.

CPU B. Se incluye una comparación en el salto.

En ambas CPU, la instrucción de salto condicional emplea 2 ciclos de reloj, y las demás instrucciones 1. (Obviamente, si el CPI es 1,0 excepto en los saltos de este sencillo ejemplo, estamos ignorando las pérdidas debidas al sistema de memoria; ver la falacia de la pág. 77.) En la CPU A, el 20 por 100 de todas las instrucciones ejecutadas son saltos condicionales; como cada salto necesita una comparación, otro 20 por 100 de las instrucciones son comparaciones. Debido a que la CPU A no incluye la comparación en el salto, su ciclo de reloj es un 25 por 100 más rápido que el de la CPU B. ¿Qué CPU es más rápida?

Respuesta

Como ignoramos todas las prestaciones del sistema, podemos utilizar la fórmula del rendimiento de la CPU: CPI_A es ((0,20 · 2) (0,80 · 1)) o 1.2, ya que el 20 por 100 son saltos que emplean 2 ciclos de reloj y el resto 1. La duración de ciclo de $reloj_B$ es 1,25 · Duración ciclo de $reloj_A$, ya que A es un 25 por 100 más rápido. El rendimiento de la CPU A es entonces

$$\begin{aligned}\text{Tiempo CPU}_A &= \text{Recuento de instrucciones}_A \cdot 1{,}2 \cdot \text{Duración ciclo de reloj}_A \\ &= 1{,}20 \cdot \text{Recuento de instrucciones}_A \cdot \text{Duración ciclo de reloj}_A\end{aligned}$$

Las comparaciones no se ejecutan en la CPU B, por tanto 20/80 o el 25 por 100 de las instrucciones ahora son bifurcaciones, que emplean 2 ciclos de reloj, y el 75 por 100 restante emplean 1 ciclo. CPI_B es entonces ((0,25 · 2) + (0,75 · 1)) o 1,25. Como la CPU B no ejecuta comparaciones, la $\text{Cuenta de instrucciones}_B$ es 0,80 · $\text{Cuenta de instrucciones}_A$. El rendimiento de la CPU B es

$$\begin{aligned}\text{Tiempo CPU}_B &= (0{,}80 \cdot \text{Recuento de instrucciones}_A) \cdot 1{,}25 \cdot (1{,}25 \cdot \\ &\quad \cdot \text{Duración ciclo de reloj}_A) \\ &= 1{,}25 \cdot \text{Recuento de instrucciones}_A \cdot \text{Duración ciclo de reloj}_A\end{aligned}$$

Bajo estas hipótesis, la CPU A, con un ciclo de reloj más corto, es más rápida que la CPU B, que ejecuta un número menor de instrucciones.

Ejemplo

Después de ver el análisis, un diseñador consideró que, volviendo a trabajar en la organización, la diferencia de las duraciones de los ciclos de reloj podía reducirse, fácilmente, un 10 por 100. ¿Qué CPU es más rápida ahora?

Respuesta

El único cambio de la respuesta anterior es que la Duración del ciclo de reloj$_B$ es ahora 1,10 · Duración del ciclo de reloj$_A$, ya que A es exactamente el 10 por 100 más rápido. El rendimiento de la CPU A es entonces

$$\text{Tiempo CPU}_A = 1{,}20 \cdot \text{Recuento de instrucciones}_A \cdot \text{Duración ciclo de reloj}_A$$

El rendimiento de la CPU B es ahora

$$\begin{aligned}\text{Tiempo CPU}_B &= (0{,}80 \cdot \text{Recuento de instrucciones}_A) \cdot 1{,}25 \cdot (1{,}10 \cdot \\ &\quad \cdot \text{Duración ciclo de reloj}_A) \\ &= 1{,}10 \cdot \text{Recuento de instrucciones}_A \cdot \text{Duración ciclo de reloj}_A\end{aligned}$$

Con esta mejora, ahora es más rápida la CPU B, que ejecuta menos instrucciones.

Ejemplo

Supongamos que estamos considerando otro cambio en un repertorio de instrucciones. La máquina, inicialmente, sólo tiene instrucciones de carga y de almacenamiento en memoria, y, después, todas las operaciones se realizan en los registros. Tales máquinas se denominan máquinas de *carga/almacenamiento (load/store)* (ver Cap. 3). En la Figura 2.2 se dan medidas de la máquina de carga/almacenamiento que muestran la frecuencia de instrucciones, denominada *mezcla de instrucciones (instruction mix),* y número de ciclos de reloj por instrucción.

Operación	Frecuencia	Cuenta de ciclos de reloj
ops ALU	43 %	1
Cargas	21 %	2
Almacenamientos	12 %	2
Saltos	24 %	2

FIGURA 2.2 Ejemplo de frecuencia de instrucciones. También se da el CPI para cada clase de instrucción. (Esta frecuencia proviene de la columna GCC de la Figura C.4 del Apéndice C, redondeada hasta que contabiliza el 100 por 100 de las instrucciones.)

Supongamos que el 25 por 100 de las operaciones de la *unidad aritmética lógica* (ALU) utilizan directamente un operando cargado que no se utiliza de nuevo.

Proponemos añadir instrucciones a la ALU que tengan un operando fuente en memoria. Estas nuevas *instrucciones de registro-memoria* emplean 2 ciclos de reloj. Supongamos que el repertorio extendido de instrucciones incrementa en 1 el número de ciclos de reloj para los saltos, pero sin afectar a la duración del ciclo de reloj. (El Capítulo 6, sobre segmentación, explica por qué añadiendo instrucciones de registro-memoria puede ralentizar los saltos.) ¿Mejorará este cambio el rendimiento de la CPU?

Respuesta La pregunta es si la nueva máquina es más rápida que la antigua. Utilizamos la fórmula del rendimiento de la CPU ya que ignoramos las prestaciones del sistema. El CPI original se calcula multiplicando las dos columnas de la Figura 2.2:

$$\text{CPI}_{\text{antiguo}} = (0{,}43 \cdot 1 + 0{,}21 \cdot 2 + 0{,}12 \cdot 2 + 0{,}24 \cdot 2) = 1{,}57$$

El rendimiento de la $\text{CPU}_{\text{antigua}}$ es entonces

$$\begin{aligned}\text{Tiempo CPU}_{\text{antigua}} &= \text{Recuento de instrucciones}_{\text{antigua}} \cdot 1{,}57 \cdot \text{Duración ciclo de reloj}_{\text{antiguo}} \\ &= 1{,}57 \cdot \text{Recuento de instrucciones}_{\text{antigua}} \cdot \text{Duración ciclo de reloj}_{\text{antiguo}}\end{aligned}$$

Veamos primero la fórmula para $\text{CPI}_{\text{nuevo}}$ y después explicaremos los componentes:

$$\text{CPI}_{\text{nuevo}} =$$

$$= \frac{(0{,}43 - 0{,}25 \cdot 0{,}43)) \cdot 1 + (0{,}21 \cdot 0{,}43)) \cdot 2 +}{1 - (0{,}25 \cdot 0{,}43)}$$

$$\frac{+ (0{,}25 \cdot 0{,}43) \cdot 2 + 0{,}12 \cdot 2 + 0{,}12 \cdot 2 + 0{,}24 \cdot 3}{1 - (0{,}25 \cdot 0{,}43)}$$

El 25 por 100 de las instrucciones de la ALU (que son el 43 por 100 de todas las instrucciones ejecutadas) pasan a ser instrucciones de registro-memoria, lo que afecta a los 3 primeros componentes del numerador. Hay $(0{,}25 \cdot 0{,}43)$ menos operaciones de la ALU, $(0{,}25 \cdot 0{,}43)$ menos cargas, y $(0{,}25 \cdot 0{,}43)$ nuevas instrucciones registro-memoria de la ALU. El resto del numerador permanece igual excepto los saltos que emplean 3 ciclos de reloj en lugar de 2. Dividimos por la nueva cuenta de instrucciones, que es $0{,}25 \cdot 43$ por 100 menor que la antigua. Al simplificar esta ecuación:

$$\text{CPI}_{\text{nuevo}} = \frac{1{,}703}{0{,}893} = 1{,}908$$

Como la duración del ciclo de reloj es inalterable, el rendimiento de la CPU es

$$\begin{aligned}\text{Tiempo CPU}_{\text{nueva}} &= (0{,}893 \cdot \text{Recuento de instrucciones}_{\text{antigua}}) \cdot 1{,}908 \cdot \\ &\quad \cdot \text{Duración ciclo de reloj}_{\text{antiguo}} \\ &= 1{,}703 \cdot \text{Recuento de instrucciones}_{\text{antigua}} \cdot \\ &\quad \cdot \text{Duración ciclo de reloj}_{\text{antiguo}}\end{aligned}$$

Utilizando estas suposiciones, la respuesta a nuestra pregunta es no. Es una mala idea añadir instrucciones registro-memoria, porque no contrarrestan el aumento del tiempo de ejecución debido a saltos más lentos.

MIPS y errores de utilización

En la búsqueda de una medida estándar del rendimiento de los computadores se ha adoptado una serie de medidas populares, con el resultado de que algunos términos inocentes se han secuestrado de un entorno bien definido y forzado a un servicio para el cual nunca fueron pensados. La posición de los autores es que la única medida fiable y consistente del rendimiento es el tiempo de ejecución de los programas reales, y que las demás alternativas propuestas al tiempo como métrica o a los programas reales como «items» medidos han conducido, eventualmente, a afirmaciones erróneas o incluso a errores en el diseño de los computadores. Primero se muestran los peligros de algunas alternativas populares a nuestra advertencia.

Una alternativa al tiempo como métrica son los MIPS, o *millones de instrucciones por segundo.* Para un programa dado, los MIPS son sencillamente

$$\text{MIPS} = \frac{\text{Recuento de instrucciones}}{\text{Tiempo de ejecución} \cdot 10^6} = \frac{\text{Frecuencia de reloj}}{\text{CPI} \cdot 10^6}$$

Algunos encuentran adecuada la fórmula de más a la derecha, ya que la frecuencia de reloj es fija para una máquina y el CPI, habitualmente, es un número pequeño, de forma distinta a la cuenta de instrucciones o al tiempo de ejecución. La relación de los MIPS con el tiempo es:

$$\text{Tiempo de ejecución} = \frac{\text{Recuento de instrucciones}}{\text{MIPS} \cdot 10^6}$$

Como los MIPS son una frecuencia de operaciones por unidad de tiempo, el rendimiento puede especificarse como el inverso del tiempo de ejecución, de forma que máquinas más rápidas tendrán una mayor frecuencia de MIPS.

La buena noticia sobre los MIPS es que son fáciles de comprender, especialmente por un cliente, y máquinas más rápidas significan un mayor número de MIPS, lo cual coincide con la intuición. El problema, cuando se utilizan los MIPS como medida para hacer comparaciones, es triple:

- Los MIPS son dependientes del repertorio de instrucciones, lo cual hace difícil la comparación de los MIPS de computadores con diferentes repertorios de instrucciones;
- Los MIPS varían entre programas en el mismo computador; y lo más importante,
- ¡Los MIPS pueden variar inversamente al rendimiento!

El ejemplo clásico, del último caso, es la variación de los MIPS en una máquina con hardware opcional de punto flotante. Como, generalmente, se emplean más ciclos de reloj por instrucción en punto flotante que por instrucción entera, los programas en punto flotante que utilizan el hardware opcional en lugar de las rutinas software de punto flotante emplean menos tiempo, pero tienen una **menor** frecuencia de MIPS. El software de punto flotante ejecuta

instrucciones más simples, dando como resultado una mayor frecuencia de MIPS, pero se ejecuta tantas veces que el tiempo global de ejecución es mayor.

Podemos incluso ver tales anomalías con compiladores optimizadores.

Ejemplo

Supongamos que construimos un compilador optimizado para la máquina de carga/almacenamiento descrita en el ejemplo anterior. El compilador descarta el 50 por 100 de las instrucciones de la ALU aunque no pueda reducir cargas, almacenamientos, ni saltos. Ignorando las prestaciones del sistema y suponiendo una duración del ciclo de reloj de 20-ns (frecuencia de reloj 50-MHz), ¿cuál es la frecuencia en MIPS para el código optimizado frente al código sin optimizar? ¿Está el «ranking» de los MIPS de acuerdo con el «ranking» del tiempo de ejecución?

Respuesta

A partir del ejemplo anterior $CPI_{\text{sin optimizar}} = 1{,}57$, por tanto

$$MIPS_{\text{sin optimizar}} = \frac{50\ \text{MHz}}{1{,}57 \cdot 10^6} = 31{,}85$$

El rendimiento del código sin optimizar es

$$\begin{aligned}\text{Tiempo CPU}_{\text{sin optimizar}} &= \text{Recuento de instrucciones}_{\text{sin optimizar}} \cdot \\ &\quad \cdot 1{,}57 \cdot (20 \cdot 10^{-9}) \\ &= 31{,}4 \cdot 10^{-9} \cdot \text{Recuento de instrucciones}_{\text{sin optimizar}}\end{aligned}$$

Para el código optimizado

$$CPI_{\text{optimizado}} = \frac{(0{,}43/2) \cdot 1 + 0{,}21 \cdot 2 + 12 \cdot 2 + 0{,}24 \cdot 2}{1 - (0{,}43/2)} =$$

$$= \frac{0{,}215 + 0{,}42 + 0{,}24 + 0{,}48}{0{,}785} = 1{,}73$$

ya que la mitad de las instrucciones de la ALU están descartadas (0,43/2) y la cuenta de instrucciones se reduce por las instrucciones que faltan de la ALU. Por ello,

$$MIPS_{\text{optimizado}} = \frac{50\ \text{MHz}}{1{,}73 \cdot 10^6} = 28{,}90$$

El rendimiento del código optimizado es

$$\begin{aligned}\text{Tiempo CPU}_{\text{optimizada}} &= (0{,}785 \cdot \text{Recuento de instrucciones}_{\text{sin optimizar}}) \cdot \\ &\quad \cdot 1{,}73 \cdot (20 \cdot 10^{-9}) \\ &= 27{,}2 \cdot 10^{-9} \cdot \text{Recuento de instrucciones}_{\text{sin optimizar}}\end{aligned}$$

El código optimizado es el 13 por 100 más rápido, ¡pero su frecuencia en MIPS es inferior!

Ejemplos como éste muestran que los MIPS pueden fallar al dar una visión verdadera del rendimiento, ya que no reflejan el tiempo de ejecución. Para compensar esta carencia, otra alternativa al tiempo de ejecución es utilizar una máquina particular, con una estimación convenida sobre los MIPS, como punto de referencia. Los *MIPS Relativos* —para distinguirlo de la forma original, denominada *MIPS nativos*— se calculan entonces como sigue:

$$\text{MIPS relativos} = \frac{\text{Tiempo}_{\text{referencia}}}{\text{Tiempo}_{\text{no estimado}}} \cdot \text{MIPS}_{\text{referencia}}$$

donde

$Tiempo_{referencia}$ = tiempo de ejecución de un programa en la máquina de referencia
$Tiempo_{no\ estimado}$ = tiempo de ejecución del mismo programa en la máquina que se va a medir
$MIPS_{referencia}$ = estimación convenida sobre los MIPS de la máquina de referencia

Los MIPS relativos sólo descubren el tiempo de ejecución para el programa y entrada dados. Incluso cuando están identificados, tal como la máquina envejece, es más difícil encontrar una máquina de referencia en la que ejecutar los programas. (En los años ochenta, la máquina de referencia dominante era la VAX-11/780, que se denominó máquina de 1-MIPS; ver la Sección 2.7.) La pregunta también surge si en la máquina más antigua deben correr las ediciones más modernas del compilador y sistema operativo, o si el software debe fijarse para que la máquina de referencia no sea más rápida a lo largo del tiempo. Existe también la tentación de generalizar de una estimación de los MIPS relativos utilizando un «benchmark» a tiempo de ejecución relativo, aun cuando pueda haber amplias variaciones en el rendimiento relativo.

En resumen, la ventaja de los MIPS relativos es pequeña, ya que tiempo de ejecución, programa y entrada del programa deben ser conocidos para que tengan información significativa.

MFLOPS y errores de utilización

Otra alternativa popular al tiempo de ejecución son los *millones de operaciones en punto flotante por segundo,* abreviadamente megaFLOPS o MFLOPS, pero siempre pronunciado «megaflops». La fórmula de los MFLOPS es simplemente la definición del acrónimo:

$$\text{MFLOPS} = \frac{\text{Número de operaciones de punto flotante de un programa}}{\text{Tiempo de ejecución} \cdot 10^6}$$

Evidentemente, una estimación en MFLOPS depende de la máquina y del programa. Como los MFLOPS se pensaron para medir el rendimiento en

Operaciones reales PF	Operaciones normalizadas PF
ADD, SUB, COMPARE, MULT	1
DIVIDE, SQRT	4
EXP, SIN, ...	8

FIGURA 2.3 Operaciones en punto flotante reales frente a las normalizadas. Número de operaciones normalizadas en punto flotante por operación real en un programa utilizado por los autores del «Livermore FORTRAN Kernels» o «Livermore Loops», para calcular los MFLOPS. Un núcleo (kernel) con una suma (ADD), una división (DIVIDE), y un seno (SIN) necesitaría 13 operaciones normalizadas en punto flotante. Los MFLOPS nativos no darán los resultados obtenidos para otras máquinas con ese benchmark.

punto flotante, no son aplicables fuera de ese rango. Los compiladores, como ejemplo extremo, tienen una estimación de MFLOPS próxima a cero sin que importe lo rápida que sea la máquina, y a que raramente utilizan aritmética en punto flotante.

El término MFLOPS está basado en las operaciones en lugar de en las instrucciones, y se pensó para que fuera una comparación buena entre diferentes máquinas. La creencia es que el mismo programa corriendo en computadores diferentes debe ejecutar un número diferente de instrucciones, pero el mismo número de operaciones en punto flotante. Desgraciadamente, los MFLOPS no son fiables, porque el conjunto de operaciones en punto flotante no es consistente con las máquinas. Por ejemplo, el CRAY-2 no tiene instrucción de dividir, mientras que el Motorola 68882 tiene división, raíz cuadrada, seno y coseno. Otro problema observado es que la estimación en MFLOPS cambia no sólo en la mezcla de operaciones de enteros y punto flotante sino también en la mezcla de operaciones rápidas y lentas de punto flotante. Por ejemplo, un programa con el 100 por 100 de sumas en punto flotante tendrá una estimación mayor que un programa con el 100 por 100 de divisiones en punto flotante. La solución para ambos problemas es dar un número canónico de operaciones de punto flotante a nivel del programa fuente y, después, dividirlo por el tiempo de ejecución. La Figura 2.3 muestra cómo los autores del benchmark «Livermore Loops» calculan el número de operaciones normalizadas en punto flotante por programas, de acuerdo con las operaciones encontradas realmente en el código fuente. Por tanto, la estimación en *MFLOPS nativos* no es la misma que la de *MFLOPS normalizados* citados en la literatura de supercomputadores, lo cual ha sido una sorpresa para algunos diseñadores de computadores.

Ejemplo

El programa Spice se ejecuta en la DECstation 3100 en 94 segundos (ver Figs. 2.16 a 2.18 para más detalles sobre el programa, entrada, compiladores, máquina, etc.) El número de operaciones de punto flotante ejecutadas en ese programa se indica a continuación:

ADDD	25 999 440
SUBD	18 266 439
MULD	33 880 810
DIVD	15 682 333
COMPARED	9 745 930
NEGD	2 617 846
ABSD	2 195 930
CONVERTD	1 581 450
TOTAL	109 970 178

¿Cuántos son los MFLOPS nativos para ese programa? Usando las conversiones de la Figura 2.3, ¿cuántos son los MFLOPS normalizados?

Respuesta

Los MFLOPS nativos se calculan fácilmente:

$$\text{MFLOPS nativos} = \frac{\text{Número de operaciones de punto flotante de un programa}}{\text{Tiempo de ejecución} \cdot 10^6}$$

$$\approx \frac{110\text{M}}{94 \cdot 10^6} \approx 1{,}2$$

La única operación de la Figura 2.3 que cambia para los MFLOPS normalizados y está en la lista anterior es la de dividir, elevando el total de operaciones en punto flotante (normalizadas), y, por tanto, el número de MFLOPS casi un 50 por 100:

$$\text{MFLOPS normalizados} \approx \frac{157\text{M}}{94 \cdot 10^6} \approx 1{,}7$$

Como cualquier otra medida de rendimiento, la estimación en MFLOPS para un único programa no puede generalizarse para establecer una métrica única de rendimiento para un computador. Como los MFLOPS normalizados representan realmente una constante dividida por el tiempo de ejecución para un programa específico y entrada específica (como los MIPS relativos), los MFLOPS son redundantes con el tiempo de ejecución, nuestra principal medida de rendimiento. Y, de manera distinta al tiempo de ejecución, es tentador caracterizar una máquina con una única estimación en MIPS o MFLOPS sin nombrar un programa. Finalmente, los MFLOPS no son una medida útil para todos los programas.

Elección de programas para evaluar el rendimiento

Dhrystone no utiliza punto flotante. Los programas típicos no...

Richk Richardson, *Aclaración de Dhrystone*, 1988

Este programa es el resultado de una búsqueda exhaustiva para determinar la mezcla de instrucciones de un programa FORTRAN normal. Los resultados de este programa en diferentes máquinas deberían dar una buena indicación de qué máquina funciona mejor bajo una carga típica de programas FORTRAN. Las sentencias están expresamente organizadas para que dificulten las optimizaciones del compilador.

Anónimo, de los comentarios del benchmark Whetstone

Un usuario de computadores que ejecuta los mismos programas día tras día debería ser el candidato perfecto para evaluar un nuevo computador. Para evaluar un nuevo sistema simplemente compararía el tiempo de ejecución de su *carga de trabajo (workload)* —la mezcla de programas y órdenes del sistema operativo que los usuarios corren en una máquina. Sin embargo, pocos están en esta feliz situación. La mayoría debe confiar en otros métodos, para evaluar las máquinas, y, con frecuencia, en otros evaluadores, esperando que estos métodos predigan el rendimiento de la nueva máquina. Hay cuatro niveles de programas utilizados en estas circunstancias, listados a continuación en orden decreciente de precisión de la previsión.

1. *Programas (Reales).*—Aunque el comprador puede no conocer qué fracción de tiempo se emplea en estos programas, sabe que algunos usuarios los ejecutarán para resolver problemas reales. Ejemplos son compiladores de C, software de tratamiento de textos como TeX, y herramientas CAD como Spice. Los programas reales tienen entradas, salidas y opciones que un usuario puede seleccionar cuando está ejecutando el programa.

2. *«Núcleos» (Kernels).*—Se han hecho algunos intentos para extraer pequeñas piezas clave de programas reales y utilizarlas para evaluar el rendimiento. «Livermore Loops» y «Linpack» son los ejemplos mejor conocidos. De forma distinta a los programas reales, ningún usuario puede correr los programas «núcleo»; únicamente se emplean para evaluar el rendimiento. Los «núcleos» son adecuados para aislar el rendimiento de las características individuales de una máquina para explicar las razones de las diferencias en los rendimientos de programas reales.

3. *Bechmarks reducidos (toys).*—Los benchmarks reducidos, normalmente, tienen entre 10 y 100 líneas de código y producen un resultado que el usuario conoce antes de ejecutarlos. Programas como la Criba de Eratóstenes, Puzzle, y Clasificación Rápida *(Quicksort)* son populares porque son pequeños, fáciles de introducir y de ejecutar casi en cualquier computador. El mejor uso de estos programas es empezar asignaciones de programación.

4. *Benchmarks Sintéticos.*—Análogos en filosofía a los «núcleos», los «benchmarks sintéticos» intentan determinar la frecuencia media de operaciones y operandos de un gran conjunto de programas. Whetstone y Dhrystone son benchmarks sintéticos populares. (Las Figs. 2.4 y 2.5 muestran algunas partes de los «benchmarks» antes citados.) Igual que ocurre con los «núcleos», ningún usuario ejecuta los benchmarks sintéticos porque no calculan nada que ningún usuario pueda utilizar. Los benchmarks sintéticos están, en efecto, aún más lejos de la realidad porque el código de los «núcleos» se extrae de programas reales, mientras que el código sintético se crea artificialmente para determinar un perfil medio de ejecución. Los benchmarks sintéticos no son *partes* de programas reales, mientras que los demás pueden serlo.

```
      I = ITER
      ...
      N8 = 899 * I
      ...
      N11 = 93 * I
      ...
      X = 1.0
      Y = 1.0
      Z = 1.0
      IF (N8) 89,89,81
81    DO 88 I = 1, N8, 1
88          CALL P3 (X,Y,Z)
89    CONTINUE
      ...
      X = 0.75
      IF (N11) 119,119,111
111   DO 118 I = 1, N11, 1
118         X = SQRT (EXP (ALOG (X) /T1))
119   CONTINUE
      ...
      SUBROUTINE P3 (X,Y,Z)
      COMMON T, TT1, T2
      X1 = X
      Y1 = Y
      X1 = T * (X1 + Y1)
      Y1 = T * (X1 + Y1)
      Z = (X1 + Y1) / T2
      RETURN
      END
      ...
```

FIGURA 2.4 Dos bucles del benchmark sintético Whetstone. Basándose en la frecuencia de las instrucciones Algol de los programas sometidos a un sistema operativo por lotes *(batch)* universitario a principios de los años setenta, se creó un programa sintético para obtener ese perfil. (Ver Curnow y Wichmann [1976]). Las instrucciones del comienzo (por ejemplo, N8 = 899 · I) controlan el número de iteraciones de cada uno de los 12 bucles (por ejemplo, el bucle DO de la línea 81). El programa se redactó más tarde en FORTRAN y se convirtió en un benchmark popular en la literatura del mercado. (La línea con la etiqueta 118 es objeto de una falacia en la Sección 2.5.)

Si no se está seguro cómo clasificar un programa, compruebe primero si tiene alguna entrada o muchas salidas. Un programa sin entradas calcula el mismo resultado cada vez que se invoca. (Pocos compran computadores para que actúen como máquinas de copiar.) Aunque algunos programas, especialmente de simulación y de aplicaciones de análisis numérico, utilizan entradas despreciables, cualquier programa real tiene alguna entrada.

Debido a que las compañías de computadores florecen o quiebran dependiendo del precio/rendimiento de sus productos con respecto a los demás productos del mercado, se dispone de enormes recursos para mejorar el rendimiento de los programas más utilizados en evaluar rendimientos. Tales presiones pueden solapar los esfuerzos de las ingenierías del hardware y software al añadir optimizaciones que mejoren el rendimiento de los programas sintéticos reducidos, o de los núcleos, pero no de los programas reales.

Un ejemplo extremo de esta ingeniería empleó optimizaciones de compiladores que eran sensibles a los «benchmarks». En lugar de realizar el análisis

```
...
for (Run_Index = 1; Run_Index<=Number_Of_Runs; ++Run_Index)
{
        Proc_5();
        Proc_4();
        Int_1_Loc = 2;
        Int_2_Loc = 3;
        strcpy(Str_2_Loc,"DHRYSTONE PROGRAMS, 2'ND STRING");
        ...
}
...
Proc_4();
}
        Boolean Bool_Loc;

        Bool_Loc = Ch1_1_Glob == 'A';
        Bool_Glob = Bool_Loc | Bool_Glob;
        Ch1_2_Glob = 'B';
} /* Proc_4 */

Proc_5()
{
        Ch1_1_Glob = 'A';
        Bool_Glob = false;
} /* Proc_5 */
...
```

FIGURA 2.5 Una sección del benchmark sintético Dhrystone. Inspirado en Whetstone, este programa fue un intento para caracterizar el rendimiento del compilador y CPU para un programa típico. Estaba basado en la frecuencia de las instrucciones de lenguajes de alto nivel de una diversidad de publicaciones. El programa se escribió originalmente en Ada y más tarde en C y Pascal (ver Weicker [1984]). Observar que el pequeño tamaño y sencilla naturaleza de estos procedimientos lo hacen trivial para que un compilador de optimización evite gastos de llamadas a procedimientos al expandirlos en línea. «`strcpy ()`» en la línea octava es objeto de una falacia en la Sección 2.5.

para que el compilador pudiera decidir adecuadamente si se podía aplicar la optimización, una persona de una compañía recién creada utilizó un preprocesador que exploraba las palabras claves del texto para tratar de identificar los benchmarks examinando el nombre del autor y el nombre de una subrutina esencial. Si la exploración confirmaba que este programa estaba en una lista predefinida, se realizaban las optimizaciones especiales. Esta máquina consiguió un cambio brusco en el rendimiento —al menos de acuerdo con aquellos benchmarks. No obstante estas optimizaciones no eran aplicables a programas que no estuviesen en la lista, sino que eran inútiles para un código idéntico con algunos cambios de nombre.

El pequeño tamaño de los programas en las últimas tres categorías los hacen vulnerables a tales esfuerzos. Por ejemplo, a pesar de las mejores intenciones, la serie inicial de benchmarks SPEC (pág. 84) incluye un pequeño programa. El 99 por 100 del tiempo de ejecución de Matrix300 está en una sola línea (ver SPEC [1989]). Una pequeña mejora del compilador MIPS FORTRAN (que mejoraba la optimización de la eliminación de variables de inducción (ver Sección 3.7 del Cap. 3) daba como resultado un incremento del rendimiento de un 56 por 100 en un M/2000 y de un 117 por 100 en un RC 6280. Esta concentración del tiempo de ejecución hizo caer a Apolo en la tentación: el rendimiento del DN 10000 se referencia cambiando esta línea por una llamada a una rutina de librería codificada a mano. Si la industria adoptase programas reales para comparar rendimientos, entonces al menos, los recursos empleados en mejorar el rendimiento ayudarían a los usuarios reales.

¿Por qué no se ejecutan programas reales para medir el rendimiento? Los núcleos y benchmarks reducidos son atractivos cuando se comienza un diseño ya que son lo suficientemente pequeños para simularlos fácilmente, incluso a mano. Son especialmente tentadores cuando se inventa una nueva máquina, porque los compiladores no están disponibles hasta mucho más tarde. Los pequeños benchmarks también se estandarizan más fácilmente, mientras que los grandes programas son más difíciles de estandarizar; de aquí que haya numerosos resultados publicados para el rendimiento de pequeños benchmarks pero pocos para los grandes.

Aunque haya racionalizaciones para usarlos al principio de un diseño, no hay ningún fundamento válido para utilizar «benchmarks» y «núcleos» para evaluar sistemas de computadores en funcionamiento. En el pasado, los lenguajes de programación eran inconsistentes entre las máquinas, y cada máquina tenía su propio sistema operativo; por ello, los programas reales no podían ser transportados sin pena ni gloria. Había también una falta importante de software cuyo código fuente se dispusiera libremente. Finalmente, los programas tenían que ser pequeños porque el simulador de la arquitectura debía de correr sobre una máquina vieja y lenta.

La popularidad de los sistemas operativos estándares como UNIX y DOS, la existencia de software distribuido libremente, básicamente entre las universidades y los computadores más rápidos, de que se dispone hoy día, eliminan la mayoría de estos obstáculos. Aunque «núcleos», benchmarks reducidos y benchmarks sintéticos fueron un intento de hacer comparaciones entre diferentes máquinas, usar cosas inferiores a los programas reales después de realizar estudios iniciales de diseño, conduce probablemente a resultados erróneos y lleva por mal camino al diseñador.

Información sobre los resultados de rendimiento

El principio a seguir a la hora de informar sobre las medidas de rendimiento deberá ser la *reproductibilidad* —enumerar todo lo que otro experimentador pueda necesitar para duplicar los resultados. Comparemos las descripciones sobre rendimientos de computadores que aparecen en revistas científicas referenciadas con las descripciones sobre rendimientos de automóviles que aparecen en las revistas que se venden en cualquier quiosco. Las revistas de automóviles, además de proporcionar 20 medidas sobre el rendimiento, indican todo el equipamiento opcional del automóvil de prueba, los tipos de neumáticos usados en la prueba de rendimiento y la fecha en que se realizó la prueba. Las revistas de computadores pueden dar solamente los segundos de ejecución, con el nombre del programa y el nombre del modelo de computador —Spice tarda 94 segundos en una DECstation 3100. Se deja a la imaginación del lector las entradas del programa, versión del programa, versión del compilador, nivel de optimización y código compilado, versión del sistema operativo, cantidad de memoria principal, número y tipos de disco, versión de la CPU —todo lo cual contribuye a que existan diferencias en el rendimiento.

Las revistas de automóviles dan suficiente información sobre las medidas para que los lectores puedan duplicar los resultados o cuestionen las opciones seleccionadas para las medidas, pero, con frecuencia, las revistas de computadores no.

Comparación y resumen de rendimientos

La comparación de rendimientos de los computadores raramente es una actividad aburrida, especialmente cuando hay diseñadores implicados. Ataques y contraataques vuelan a través de la red electrónica; uno es acusado de prácticas sucias y otro de afirmaciones erróneas. Como las carreras profesionales, a veces, dependen de los resultados de las comparaciones sobre el rendimiento, es comprensible que la verdad sea amplificada ocasionalmente. Pero las discrepancias más frecuentes pueden deberse a diferentes hipótesis o falta de información.

Nos gustaría pensar que si pudiésemos estar de acuerdo sobre programas, entornos experimentales y la definición de «más rápido», se evitarían muchas faltas de entendimiento, y las redes de computadores estarían libres para cuestiones eruditas. Desgraciadamente, las cosas no son tan fáciles, las batallas se libran entonces sobre cuál es la mejor forma de resumir el rendimiento relativo de una colección de programas. Por ejemplo, dos artículos sobre resumenes de rendimiento en la misma revista tenían puntos de vista opuestos. La Figura 2.6, tomada de uno de los artículos, es un ejemplo de la confusión que puede surgir.

Utilizando nuestra definición del Capítulo 1 (pág. 6), se deducen las siguientes afirmaciones:

A es 900 por 100 más rápido que B para el programa 1

B es 900 por 100 más rápido que A para el programa 2

	Computador A	Computador B	Computador C
Programa 1 (s)	1	10	20
Programa 2 (s)	1000	100	20
Tiempo total (s)	1001	110	40

FIGURA 2.6 Tiempos de ejecución de dos programas en tres máquinas. Tomado de la Figura 1 de Smith [1988].

A es 1900 por 100 más rápido que C para el programa 1

C es 4900 por 100 más rápido que A para el programa 2

B es 100 por 100 más rápido que C para el programa 1

C es 400 por 100 más rápido que B para el programa 2

Tomadas individualmente, cada una de estas afirmaciones puede ser útil. Sin embargo, colectivamente, presentan un cuadro confuso —el rendimiento relativo de los computadores A, B y C no está claro.

Tiempo total de ejecución: una medida resumen consistente

La aproximación más sencilla para resumir el rendimiento relativo es utilizar el tiempo de ejecución total de los dos programas. Así

B es 810 por 100 más rápido que A para los programas 1 y 2

C es 2400 por 100 más rápido que A para los programas 1 y 2

C es 175 por 100 más rápido que B para los programas 1 y 2

Este resumen da pistas sobre los tiempos de ejecución, nuestra medida final del rendimiento. Si la carga de trabajo consistía en correr los programas 1 y 2 un número igual de veces, las afirmaciones anteriores predicen los tiempos de ejecución relativos para la carga de trabajo de cada máquina.

Una media de los tiempos de ejecución basada en el tiempo de ejecución total es la *media aritmética.*

$$\frac{1}{n}\sum_{i=1}^{n} \text{Tiempo}_i$$

en donde Tiempo_i es el tiempo de ejecución del programa *i*ésimo de un total de n de la carga de trabajo. Si el rendimiento se expresa como una frecuencia (como MFLOPS), entonces la media del tiempo de ejecución total es la *media armónica*

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{\text{Velocidad}_i}}$$

donde Velocidad$_i$ es una función de 1/Tiempo$_i$, el tiempo de ejecución para el *i*ésimo de *n* programas de la carga de trabajo.

Tiempo de ejecución ponderado

Se plantea el problema de cual es la mezcla adecuada de programas para la carga de trabajo: ¿están los programas 1 y 2 corriendo efectivamente de igual forma en la carga de trabajo como supone la media aritmética? Si no es así, se puede emplear una de dos aproximaciones para resumir el rendimiento. La primera aproximación consiste en asignar a cada programa un factor de peso w_i que indique la frecuencia relativa del programa en esa carga de trabajo. Si, por ejemplo, el 20 por 100 de las tareas de la carga de trabajo correspondiesen al programa 1 y el 80 por 100 al programa 2, entonces los factores de peso serían 0,2 y 0,8. (La suma de los factores de peso siempre es 1.) Al sumar los productos de los factores de peso por los tiempos de ejecución, se obtiene un cuadro claro de rendimiento de la carga de trabajo. A esto se denomina *media aritmética ponderada:*

$$\sum_{i=1}^{n} \text{Peso}_i \cdot \text{Tiempo}_i$$

donde Peso$_i$ es la frecuencia del programa *i*ésimo de la carga de trabajo y Tiempo$_i$ es el tiempo de ejecución de ese programa. La Figura 2.7 muestra los datos de la Figura 2.6 con tres pesos diferentes, cada uno proporcional al

	A	B	C	W(1)	W(2)	W(3)
Programa 1 (s)	1,00	10,00	20,00	0,50	0,909	0,999
Programa 2 (s)	1000,00	100,00	20,00	0,50	0,091	0,001
Media aritmética: W(1)	500,50	55,00	20,00			
Media aritmética: W(2)	91,82	18,18	20,00			
Media aritmética: W(3)	2,00	10,09	20,00			

FIGURA 2.7 Medias aritméticas ponderadas de los tiempos de ejecución utilizando tres pesos. W(1) pondera de la misma forma los programas, dando como resultado una media (fila 3) que es la misma que la media aritmética no ponderada. W(2) hace la mezcla de programas inversamente proporcional a los tiempos de ejecución en la máquina B; la fila 4 muestra la media aritmética para esa ponderación. W(3) pondera los programas en proporción inversa a los tiempos de ejecución de los dos programas en la máquina A; la media aritmética se da en la última fila. El efecto neto de los pesos segundo y tercero es «normalizar» los pesos de los tiempos de ejecución de los programas que se ejecutan en esa máquina, para que el tiempo de ejecución se distribuya uniformemente entre cada programa para esa máquina. Para un conjunto de *n* programas tales que cada uno emplea un tiempo T_i en un máquina, los pesos de igual tiempo en esa máquina son $w_i = \dfrac{1}{T_i \cdot \sum_{j=1}^{n} \left(\dfrac{1}{T_j}\right)}$.

tiempo de ejecución de una carga de trabajo en una mezcla dada. La *media armónica ponderada* de frecuencias mostrará el mismo rendimiento relativo que la media aritmética ponderada de los tiempos de ejecución. La definición es

$$\frac{1}{\sum_{i=1}^{n} \frac{\text{Peso}_i}{\text{Velocidad}_i}}$$

Tiempo de ejecución normalizado y «pros» y «contras» de las medias geométricas

Una segunda aproximación cuando hay una mezcla desigual de programas en la carga de trabajo consiste en normalizar los tiempos de ejecución para una máquina de referencia y después tomar la media de los tiempos de ejecución normalizados, análogo a las estimaciones en MIPS relativos anteriormente explicadas. Esta medida surgiere que el rendimiento de los nuevos programas puede predecirse multiplicando sencillamente este números por su rendimiento en la máquina de referencia.

El tiempo medio de ejecución normalizado puede expresarse como una media aritmética o *geométrica.* La fórmula para la media geométrica es

$$\sqrt[n]{\prod_{i=1}^{n} \text{Razón del tiempo de ejecución}_i}$$

donde *razón del tiempo de ejecución*$_i$ es el tiempo de ejecución, normalizado para la máquina de referencia, para el *i*ésimo programa de un total de *n* de la carga de trabajo. Las medias geométricas también tienen la siguiente propiedad

$$\frac{\text{Media geométrica } (X_i)}{\text{Media geométrica } (Y_i)} = \text{Media geométrica}\left(\frac{X_i}{Y_i}\right)$$

que significa que el cociente de las medias o la media de los cocientes dan el mismo resultado. En contraste con las medias aritméticas, las medias geométricas de los tiempos de ejecución normalizados son consistentes sin importar cual sea la máquina de referencia. Por consiguiente, la media aritmética *no* se debe utilizar para promediar tiempos de ejecución normalizados. La Figura 2.8 muestra algunas variaciones utilizando medias aritméticas y geométricas de tiempos normalizados.

Debido a que los pesos de las medias aritméticas ponderadas son proporcionales a los tiempos de ejecución en una máquina dada, como en la Figura 2.7, éstos están influenciados no solo por la frecuencia de uso de la carga de trabajo, sino también por las peculiaridades de una máquina particular y por el tamaño de las entradas del programa. La media geométrica de los tiempos de ejecución normalizados, por otro lado, es independiente de los tiempos

	Normalizado para A (%)			Normalizado para B (%)			Normalizado para C (%)		
	A	B	C	A	B	C	A	B	C
Programa 1	100	1000	2000	10	100	200	5	50	100
Programa 2	100	10	2	1000	100	20	5000	500	100
Media aritmética	100	505	1001	505	100	110	2503	275	100
Media geométrica	100	100	63	100	100	63	158	158	100
Tiempo total	100	11	4	910	100	36	2503	275	100

FIGURA 2.8 Tiempos de ejecución de la Figura 2.6 normalizados para cada máquina. El rendimiento de la media aritmética varía dependiendo de cuál sea la máquina de referencia —la columna 2 dice que el tiempo de ejecución de la máquina B es cinco veces mayor que el de la A mientras que la columna 4 dice exactamente lo opuesto; la columna 3 indica que C es más lenta mientras que la columna 9 indica que C es más rápida. Las medias geométricas son consistentes independientemente de la normalización— A y B tienen el mismo rendimiento, y el tiempo de ejecución de C es el 63 por 100 de A o B (100/158 por 100 es 63 por 100). Desgraciadamente el tiempo total de ejecución de A es nueve veces mayor que el de B, y el de B es tres veces mayor que el de C. Como nota interesante, la relación entre las medias del mismo conjunto de número es siempre media armónica $\leqslant$ media geométrica $\leqslant$ media aritmética.

de ejecución de los programas individuales, y no importa la máquina que se utilice para normalizar. Si surgiese una situación en la evaluación comparativa de rendimientos donde los programas fuesen fijos pero las entradas no, entonces los competidores podrían manipular los resultados de las medias aritméticas ponderadas haciendo que sus mejores «benchmarks» de rendimiento tuviesen la entrada más grande dominando por tanto el tiempo de ejecución. En dicha situación la media geométrica sería menos errónea que la media aritmética.

El mayor inconveniente para las medias geométricas de los tiempos de ejecución normalizados es que violan nuestro principio fundamental de medida del rendimiento —no predicen los tiempos de ejecución. Las medias geométricas de la Figura 2.8 sugieren que para los programas 1 y 2 el rendimiento de las máquinas A y B es idéntico, esto sería cierto para una carga de trabajo que ejecutase el programa 1 cien veces por cada ocurrencia del programa 2 (ver Fig. 2.6). El tiempo de ejecución total para dicha carga de trabajo sugiere que las máquinas A y B son aproximadamente el 50 por 100 más rápidas que la máquina C, en contraste con la media geométrica, ¡que indica que la máquina C es más rápida que la A y B! En general **no hay carga de trabajo** para tres o más máquinas que haga coincidir el rendimiento predicho por las medias geométricas de los tiempos de ejecución normalizados. Nuestra razón original para examinar las medias geométricas de los rendimientos normalizados fue para evitar dar el mismo énfasis a los programas de nuestra carga de trabajo, pero ¿es esta solución una mejora?

La solución ideal es medir una carga de trabajo real y ponderar los programas de acuerdo con su frecuencia de ejecución. Si esto no puede hacerse así, entonces normalizar para que se emplee el mismo tiempo en cada programa sobre alguna máquina; al menos esto hace explícitos los pesos relativos y predice el tiempo de ejecución de una carga de trabajo con esa mezcla (ver Fig. 2.7). La mejor solución al problema anterior de entradas inespecificadas

es especificar las entradas cuando se comparan los rendimientos. Si los resultados deben normalizarse para una máquina específica, resumir primero el rendimiento con la medida ponderada apropiada y después normalizar. La Sección 2.4 da un ejemplo.

2.3 Coste

Aunque hay diseños de computadores donde los costes tienden a ignorarse —específicamente en los supercomputadores— los diseños sensibles al coste están creciendo en importancia. Los libros de texto han ignorado la mitad del coste de la relación coste/rendimiento, porque los costes cambian desde que se publican los libros. La comprensión del coste es esencial para que los diseñadores puedan tomar decisiones inteligentes sobre si se debe incluir una determinada característica en los diseños donde el coste es un problema. (Imagine arquitectos que diseñen rascacielos sin tener ninguna información sobre los costes de las vigas de acero y del hormigón.) Por tanto, en esta sección cubriremos los fundamentos de los costes que no cambiarán durante la vida del libro y daremos ejemplos específicos utilizando costes que, aunque no se mantengan a lo largo del tiempo, demuestren los conceptos empleados.

El rápido cambio de los costes de la electrónica es el primero de una serie de temas en los diseños sensibles al coste. Este parámetro cambia tan rápidamente, que los buenos diseñadores basan sus decisiones no en los costes actuales sino en los costes proyectados en el instante en que se fabrique el producto. El principio fundamental que controla los costes es la *curva de evolución (learning curve)* —los costes de manufactura decrecen con el tiempo. La curva de evolución es la mejor medida de la transformación de la *productividad (yield)* —el procentaje de dispositivos fabricados que superan al procedimiento de prueba *(testing)*. Si es un circuito integrado, un circuito impreso o un sistema, los diseños que tengan doble productividad tendrán básicamente la mitad de coste. Comprender la manera en que la curva de evolución mejorará la productividad es clave para proyectar los costes durante la vida del producto.

Costes más bajos, sin embargo, no necesariamente hacen los precios más bajos; ya que pueden incrementar los beneficios. Pero cuando el producto está disponible en múltiples fuentes y la demanda no supera la oferta, la competencia fuerza a que caigan los precios con los costes. Durante el resto de esta discusión supondremos que las fuerzas competitivas normales tienen un equilibrio razonable entre oferta y demanda.

Como ejemplo de la curva de evolución en acción, el coste por megabyte de los DRAM cae a largo plazo un 40 por 100 por año. Una versión más espectacular de la misma información se muestra en la Figura 2.9, donde el coste de un nuevo «chip» (pastilla integrada) de DRAM se dibuja a lo largo de su vida. Entre el comienzo del proyecto y la fabricación de un producto, digamos dos años, el coste de una nueva DRAM cae aproximadamente en un factor de cuatro. Como no todos los costes de los componentes cambian a la misma

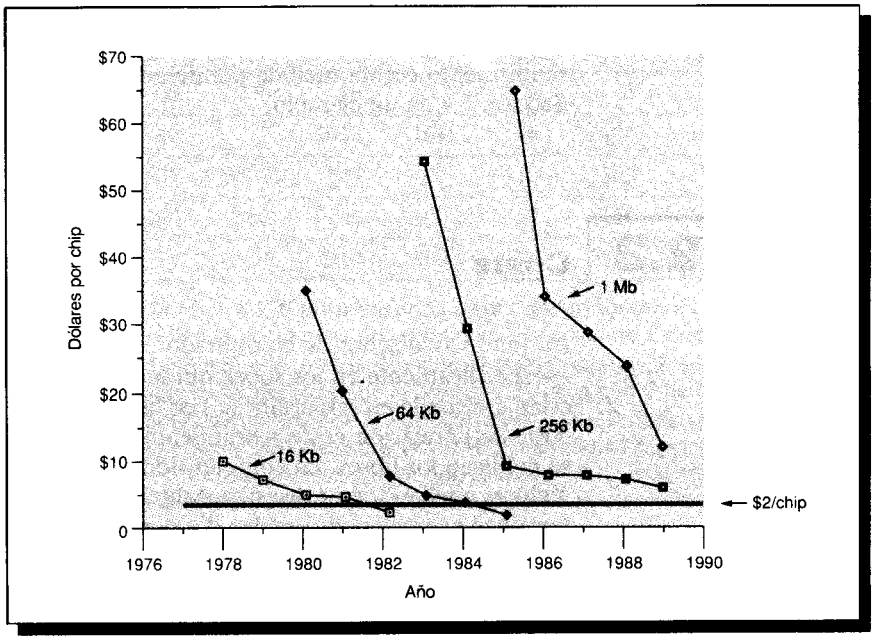


FIGURA 2.9 Evolución de los precios de cuatro generaciones de DRAM, mostrando la curva de evolución. Aunque el promedio mayor es el 40 por 100 de mejora por año, cada generación disminuye el precio aproximadamente en un factor de 10 durante su tiempo de vida. Las DRAM bajan aproximadamente a 1 o 2 dólares por chip (pastilla integrada), independientemente de su capacidad. Los precios **no** están ajustados por la inflacción —si lo estuviesen el gráfico mostraría aun una mayor caída en el coste. Entre 1987 y 1988 los precios de las DRAM de 256 Kb y 1 Mb fueron mayores que los indicados por anteriores curvas de evolción debido a que parecía que iba a haber un exceso de demanda temporalmente con relación al suministro disponible.

velocidad, los diseños basados en los costes proyectados dan una relación coste-rendimiento diferente de la que se obtendría utilizando costes actuales.

Un segundo tema importante en los diseños sensibles al coste es el impacto del encapsulamiento en las decisiones de diseño. Hace pocos años las ventajas de ajustar un diseño en un solo circuito impreso daban como resultado un menor coste e incluso un rendimiento más elevado. Dentro de pocos años será posible integrar todos los componentes de un sistema, excepto la memoria principal, en un único chip. El principal problema será lograr que el sistema esté en un chip, evitando así penalizaciones de velocidad y coste al tener múltiples chips, lo que significa más interfaces, más patillas *(pins)* para interfaces, mayores circuitos impresos, etc. La densidad de circuitos integrados y la tecnología de encapsulamiento determinan los recursos disponibles en cada umbral de coste. El diseñador debe conocer dónde están estos umbrales —o cruzarlos a ciegas.

Coste de un circuito integrado

¿Por qué un libro de arquitectura de computadores tiene una sección dedicada al coste de los circuitos integrados? En un mercado de computadores, crecientemente competitivo, donde las partes estándares —discos, DRAM, etc.— están convirtiéndose en una parte significativa del coste de cualquier sistema, el coste de los circuitos integrados empieza a representar una gran proporción del coste global, que varía entre máquinas, especialmente en la parte del mercado de gran volumen sensible al coste. Por ello, los diseñadores de computadores deben conocer los costes de los chips para comprender los costes de los computadores actuales. Seguimos aquí el enfoque de contabilidad norteamericano para el coste de los chips.

Aunque los costes de los circuitos integrados han caído exponencialmente, el procedimiento básico de la manufactura del sicilio sigue inalterado: una *oblea* todavía se examina *(test)* y divide en *dados (dies)* que se encapsulan (ver Figs. 2.10a, b, y c). Así, el coste de un chip encapsulado es

$$\text{Coste del circuito integrado} = \frac{\text{Coste del dado} + \text{Coste del test del dado} + \text{Coste del encapsulamiento}}{\text{Productividad del test final}}$$

Coste de los dados

Saber cómo predecir el número de «chips» buenos por oblea requiere aprender primero cuántos dados caben en una oblea y cómo predecir el porcentaje de los que funcionarán. A partir de aquí es fácil predecir el coste:

$$\text{Coste del dado} = \frac{\text{Coste de la oblea}}{\text{Dados por oblea} \cdot \text{Productividad del dado}}$$

La característica más interesante de este primer término de la ecuación del coste del chip es su sensibilidad al tamaño del dado, lo que se muestra a continuación.

El número de dados por oblea es básicamente el área de la oblea dividida por el área del dado. Puede estimarse precisamente por

$$\text{Dados por oblea} = \frac{\pi \cdot (\text{Diámetro de la oblea}/2)^2}{\text{Area del dado}} - \frac{\pi \cdot \text{Diámetro de la oblea}}{\sqrt{2 \cdot \text{Area del dado}}} - \text{Dados de test por oblea}$$

El primer término es la relación del área de la oblea (πr^2) al área de dado. El segundo compensa el problema del encuadre —dados rectangulares cerca de la periferia de las obleas circulares. Dividir la circunferencia (πd) por la dia-

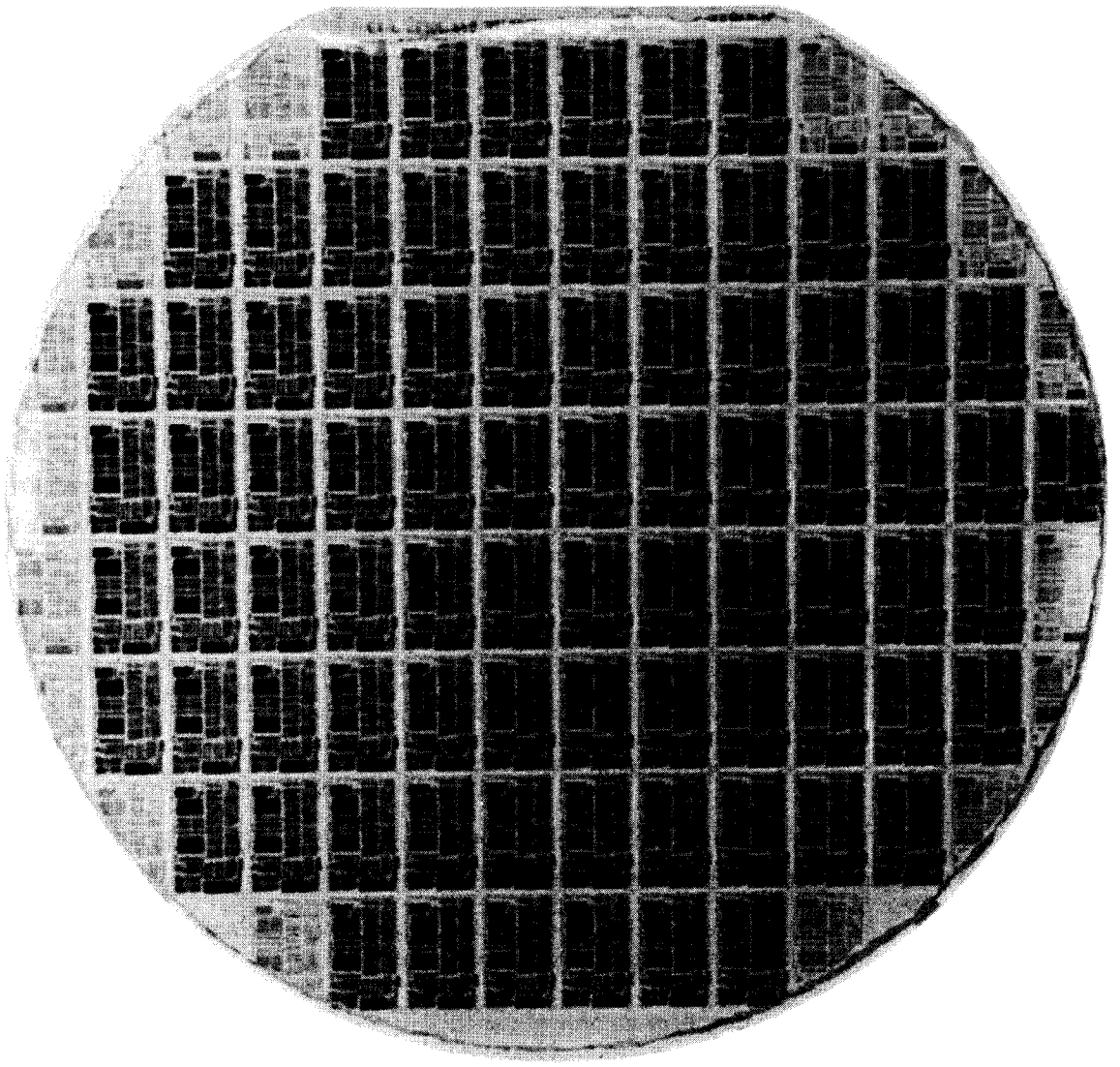


FIGURA 2.10a Fotografía de una oblea de 6 pulgadas que contiene microprocesadores Intel 80486. Hay 80 dados de 1,6 cm x 1,0 cm, aunque cuatro dados están tan próximos al extremo que pueden o no pueden ser completamente funcionales. No hay dados de test separados; en cambio, se colocan **entre** los dados los circuitos paramétricos y eléctricos de test. El 80486 tiene una unidad de punto flotante, una pequeña cache, y una unidad de gestión de memoria además de la unidad entera.

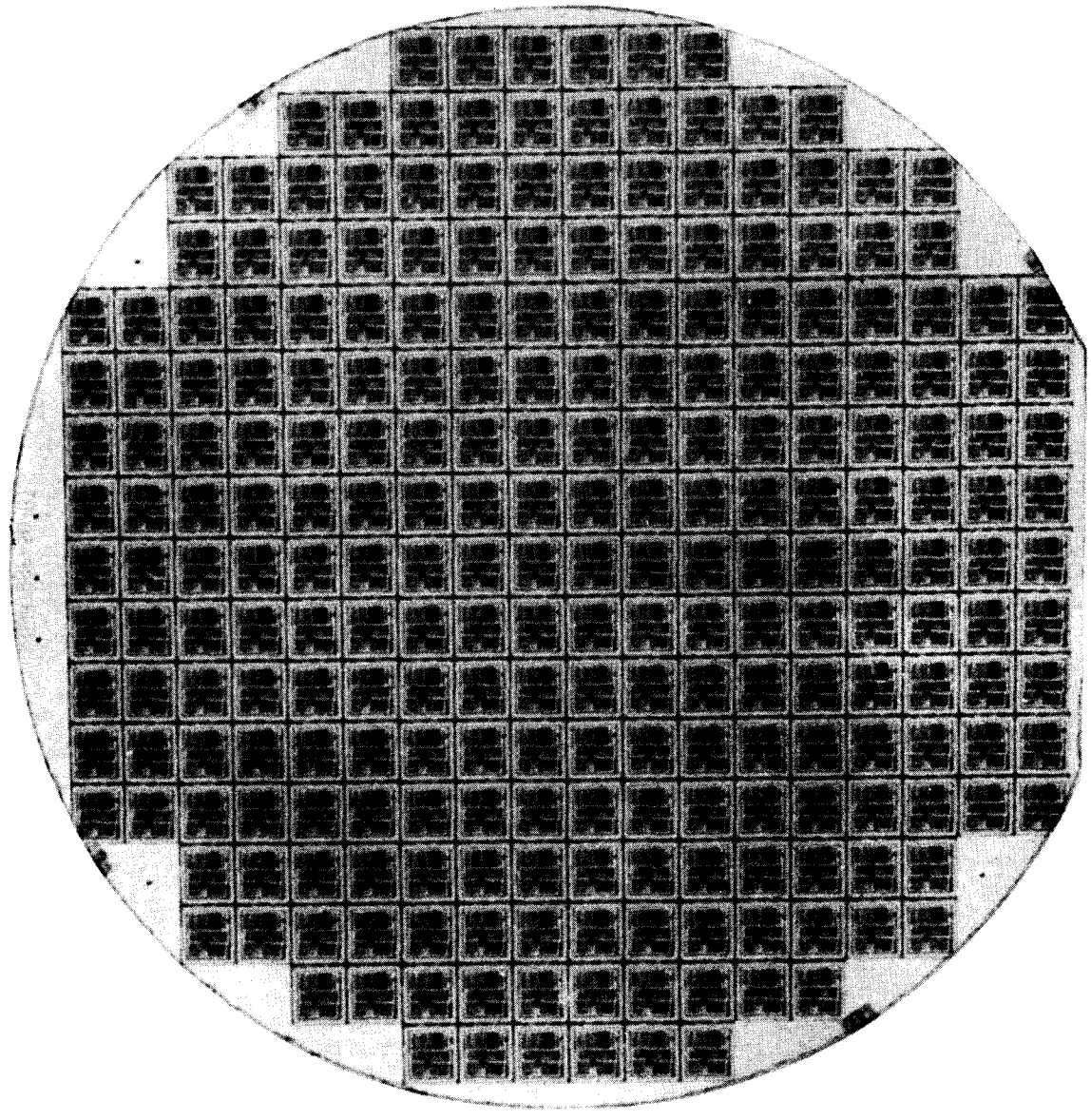


FIGURA 2.10b Fotografía de una oblea de 6 pulgadas que contiene microprocesadores Cypress CY7C601. Hay 246 dados de 0,8 cm × 0,7 cm, aunque, de nuevo, cuatro dados están tan próximos al extremo que es difícil decir si están completos. Igual que Intel, Cypress coloca los circuitos paramétricos y eléctricos de test entre los dados. Estos circuitos de test se eliminan cuando la oblea se descompone en chips. En contraste con el 80486, el CY7C601 contiene sólo la unidad entera.

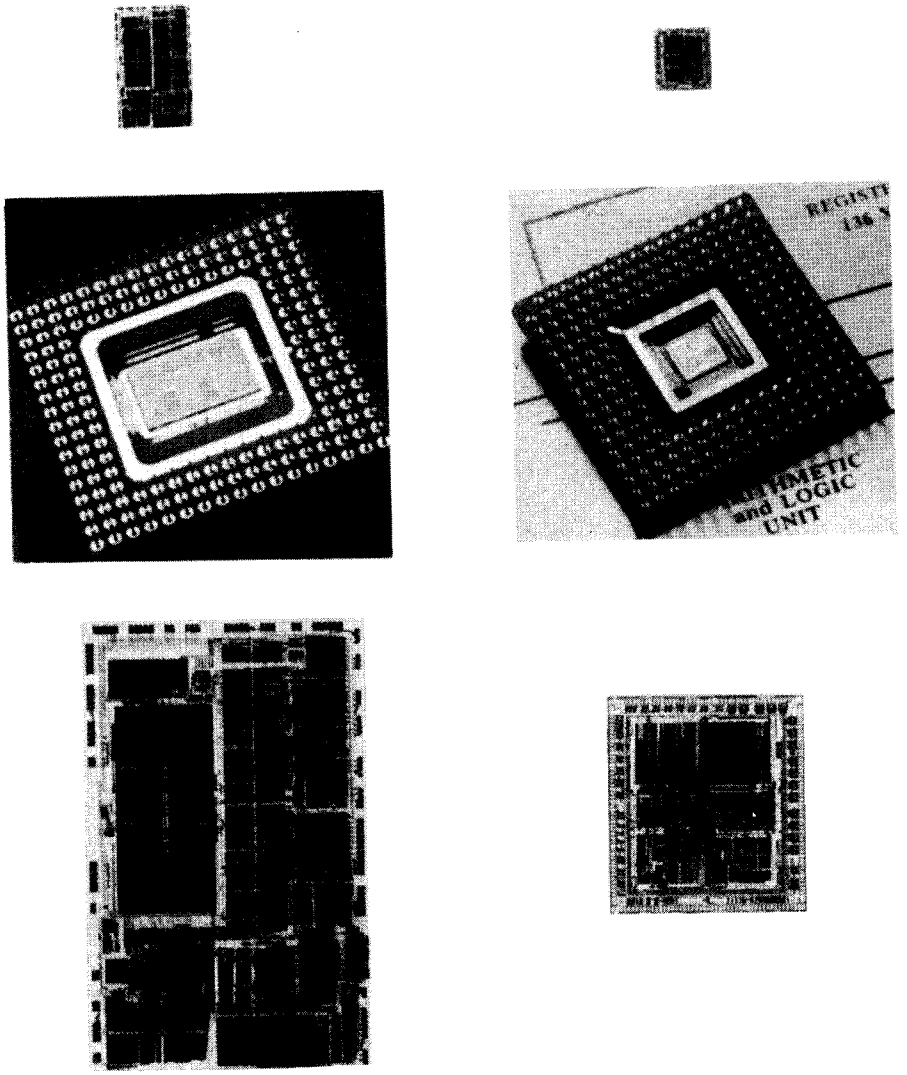


FIGURA 2.10c En la esquina superior izquierda está el dado Intel 80486, y el dado Cypress CY7C601 está a la derecha, ambos se muestran en sus tamaños reales. Debajo de los dados aparecen las versiones encapsuladas de cada microprocesador. Observar que el 80486 tiene tres filas de patillas (168 en total) mientras que el 601 tiene cuatro filas (207 en total). La fila inferior muestra una ampliación de los dos dados, que aparecen conservando sus proporciones relativas.

gonal de un dado cuadrado es, aproximadamente, el número de dados en la periferia. El último término es para los dados de prueba *(test)* que deben colocarse estratégicamente para controlar la fabricación. Por ejemplo, una oblea de 15 cm de diámetro (aproximadamente 6 pulgadas) con 5 dados de test contiene $3{,}14 \cdot 225/4 - 3{,}14 \cdot 15/\sqrt{2} - 5$ o 138 dados de 1 cm cuadrado. Duplicar el área del dado —el parámetro que un diseñador de computadores controla— reduciría el número de dados por oblea a 59.

Pero esto solamente da el número máximo de dados por oblea, y la pregunta crítica es ¿cuál es la fracción o porcentaje de dados buenos en una serie de obleas, o la *productividad del dado? (die yield)*. Un sencillo modelo del rendimiento de un circuito integrado supone que los defectos están distribuidos aleatoriamente en la oblea:

$$\text{Productividad del dado} = \text{Productividad de la oblea} \cdot \left\{1 + \frac{\text{Defectos por unidad de área} \cdot \text{Area del dado}}{\alpha}\right\}^{-\alpha}$$

donde *Productividad de la oblea* contabiliza las obleas que son completamente malas y no necesitan ser examinadas y α es un parámetro que corresponde, aproximadamente, al número de niveles de máscaras críticas para la productividad del dado. α depende también del proceso de manufactura. Generalmente $\alpha = 2{,}0$ para procesos MOS sencillos y tienen valores más elevados para procesos más complejos, como bipolares y BiCMOS. A título de ejemplo, si la productividad de la oblea es del 90 por 100, los *defectos por unidad de área* son 2 por centímetro cuadrado, y el área de dado es 1 centímetro cuadrado. Entonces la producitividad del dado es $90\% \cdot (1 + (2 \cdot 1)/2{,}0)^{-2{,}0}$, o sea 22,5 por 100.

La línea inferior es el número de dados buenos por oblea, y se obtiene al multiplicar los dados por oblea por el rendimiento por dado. Los ejemplos anteriores predicen $138 \cdot 0{,}225$, o sea 31 dados buenos de 1 centímetro cuadrado por oblea de 15 cm. Como mencionamos antes, tanto los dados por oblea como la productividad del dado son sensibles al tamaño del dado —¡duplicar el área del dado hace caer la productividad del dado a un 10 por 100 y los chips buenos por oblea a $59 \cdot 0{,}10$, o sea 6! El tamaño del dado depende de la tecnología y de las puertas que se necesiten para la función que realice el chip, pero está también limitado por el número de patillas que se pueden colocar en el borde de un dado cuadrado.

Una oblea de 15 cm de diámetro procesada en CMOS con dos niveles de metal le costaba a un fabricante de semiconductores aproximadamente 550 dólares en 1990. El coste de un dado de 1 centímetro cuadrado con dos defectos por centímetro cuadrado en una oblea de 15 cm es $\$550/(138 \cdot 0{,}225)$ o 17,74 dólares.

¿Qué debería recordar un diseñador de computadores sobre el coste de un circuito integrado? El proceso de fabricación dicta el coste de la oblea, la productividad de la oblea, α, y los defectos por unidad de área; así el único control del diseñador es el área del dado. Como α es habitualmente mayor o igual

que 2, el coste del dado es proporcional a la tercera potencia (o superior) del área del dado:

$$\text{Coste del dado} = f(\text{Area del dado}^3)$$

Al diseñador de computadores le afecta el tamaño del dado, y, por consiguiente, el coste, tanto por las funciones que están incluidas o excluidas del dado como por el número de patillas de E/S.

Costes del test y encapsulamiento del dado

El test es el segundo término de la ecuación del chip, y la frecuencia de éxitos de los tests (productividad del dado) afecta el coste del test:

$$\text{Coste del test del dado} = \frac{\text{Coste del test por hora} \cdot \text{Tiempo medio del test del dado}}{\text{Productividad del dado}}$$

Como los dados malos se descartan, la productividad del dado está en el denominador de la ecuación —los buenos deben cargar con los costes de test de los que fallan. En 1990 el test costaba aproximadamente 150 dólares por hora y el test del dado necesita una media de 5 a 90 segundos, dependiendo de la simplicidad del dado y de las provisiones para reducir el tiempo de prueba incluidas en el chip. Por ejemplo, a 150 dólares por hora y 5 segundos para el test, el coste del test es 0,21 dólares. Después de factorizar la productividad del dado para un dado de 1 centímetro cuadrado, el coste es de 0,93 dólares por dado bueno. Un segundo ejemplo, supongamos que el test dura 90 segundos. El coste es 3,75 dólares por dado sin test y 16,67 dólares por dado bueno. La factura, hasta el momento, para nuestro dado de 1 centímetro cuadrado, es de 18,67 o de 34,41 dólares dependiendo de lo que dure el test. Estos dos ejemplos ilustran la importancia de reducir el tiempo de test para reducir el coste.

Coste del encapsulamiento y productividad del test final

El coste de un encapsulado depende del material usado, del número de patillas y del área del dado. El coste del material usado en el encapsulado está determinado, en parte, por las posibilidades de disipación de la potencia generada por el dado. Por ejemplo, un encapsulamiento *plano cuadrado de plástico* (PQFP) que disipa menos de un vatio, con 208 patillas como máximo, y que contenga un dado con un centímetro de lado como máximo cuesta 3 dólares en 1990. Un *array cerámico de rejilla de patillas* (PGA) puede tener de 300 a 400 patillas y un dado mayor con más potencia, pero cuesta 50 dólares. Además del coste de la cápsula está el coste del trabajo de colocar el dado en la cápsula y, después, unir los «pads» a las patillas. Podemos suponer que cuesta 2 dólares. Durante el proceso de quemado se somete al dado encapsulado a

Area (cm^2)	Lado (cm)	Dado/ oblea	Productividad dado/oblea (%)	Coste del dado ($)	Coste del test del dado ($)	Costes de encapsulamiento ($)	Coste total después del test final ($)
0,06	0,25	2778	79,72	0,25	0,63	5,25	6,81
0,25	0,50	656	57,60	1,46	0,87	5,25	8,42
0,56	0,75	274	36,86	5,45	1,36	5,25	13,40
1,00	1,00	143	22,50	17,09	2,22	5,25	27,29
1,56	1,25	84	13,71	47,76	3,65	52,25	115,18
2,25	1,50	53	8,52	121,80	5,87	52,25	199,91
3,06	1,75	35	5,45	288,34	9,17	52,25	388,62
4,00	2,00	23	3,60	664,25	13,89	52,25	811,54

FIGURA 2.11 Costes para varios tamaños de dados. Los costes se muestran en las columnas 5 a 7. La columna 8 es la suma de las columnas 5 a 7 dividida por la productividad del test final. La Figura 2.12 presenta gráficamente esta información. Esta figura supone una oblea de 15,24 cm (6 pulgadas) que cuesta 550 dólares, con 5 dados de test por oblea. El rendimiento de la oblea es del 90 por 100, la densidad de defectos es 2,0 por centímetro cuadrado, y α es 2,0. Se necesita un promedio de 12 segundos en «testear» un dado, el examinador (tester) cuesta 150 dólares por hora, y el rendimiento del test final es del 90 por 100. (Los números difieren un poco del texto para un dado de un centímetro cuadrado porque el tamaño de la oblea se calcula en 15,24 cm en lugar de redondearlo a 15 cm y a causa de la diferencia del tiempo de test.)

una potencia elevada durante un tiempo muy pequeño con el fin de desechar los circuitos integrados que fallarían muy pronto. Estos costes suponían unos 0,25 dólares en 1990.

El capítulo de costes no finaliza hasta que no se tenga noticia de los fallos de algunos chips durante los procesos de ensamblamiento y quemado. Utilizando una estimación del 90 por 100 para el rendimiento del test final, los dados buenos deben pagar de nuevo el coste de los que fallan, por tanto los costes son de 26,58 dólares a 96,29 dólares por un dado de 1 centímetro cuadrado.

Aunque estas estimaciones de costes específicos pueden no mantenerse, los fundamentos sí se mantinen. La Figura 2.11 muestra los dados por oblea, rendimiento del dado, y su producto frente al área del dado para una línea de fabricación típica, esta vez utilizando programas que predicen de forma más precisa los dados por oblea y la productividad del dado. La Figura 2.12 representa los cambios de área y coste cuando cambia una dimensión del dado cuadrado. Los cambios para dados pequeños suponen poca diferencia en el coste aunque incrementos del 30 por 100 en dados grandes pueden duplicar los costes. El diseñador de silicio prudente minimizará el área del dado, el tiempo del test, y las patillas por chip y estudiará las opciones de los costes proyectados del encapsulamiento cuando considere utilizar más potencia, patillas, o área para conseguir mayores prestaciones.

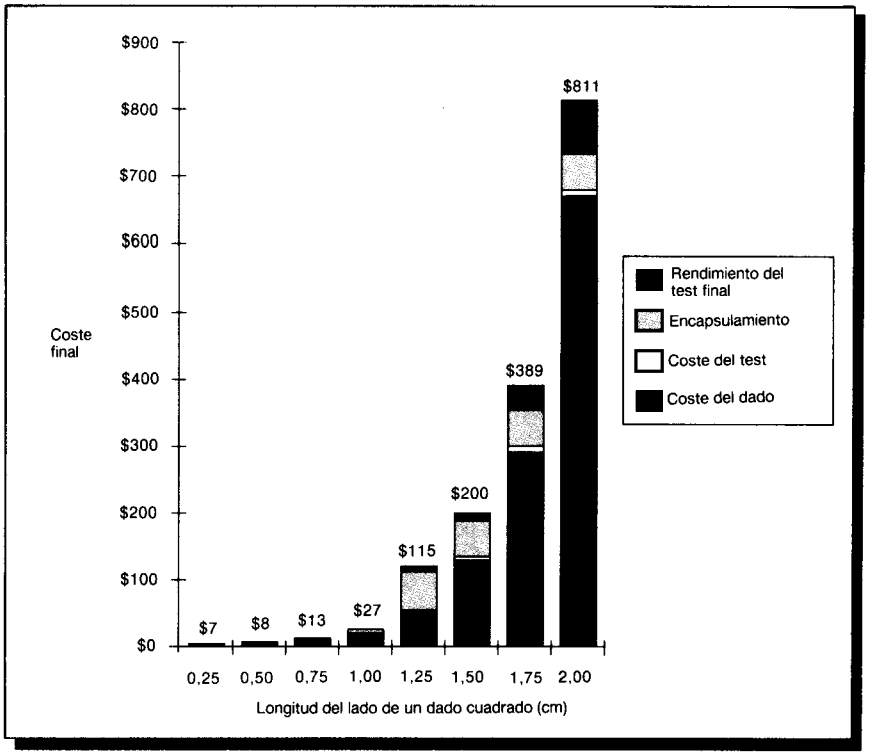


FIGURA 2.12 Los costes del chip de la Figura 2.11 presentados gráficamente. Utilizando los parámetros dados en el texto, el encapsulamiento supone un porcentaje importante del coste de los dados cuyo tamaño máximo es 1,25 centímetros cuadrados, para dados mayores el coste del dado domina los costes finales.

Coste de una estación de trabajo

Para poner los costes del silicio en perspectiva, la Figura 2.13 muestra los costes aproximados de los componentes de una estación de trabajo en 1990. Los costes de un componente pueden dividirse por dos al pasar de bajo volumen a alto volumen; aquí supondremos alto volumen de compras: 100 000 unidades. Aunque los costes para unidades como las DRAM seguramente bajarán a lo largo del tiempo con respecto a los de la Figura 2.13, las unidades cuyos precios ya han caído, como pantallas y carcasas, cambiarán muy poco.

El procesador, unidad de punto flotante, unidad de gestión de memoria, y cache, representan sólo del 12 al 21 por 100 del coste de la CPU de la Figura 2.13. Dependiendo de las opciones incluidas en el sistema —número de discos, monitor en color, etc.— los componentes del procesador caen del 9 al 16 por 100 del coste de un sistema, como ilustra la Figura 2.14. En el futuro

			Regla empírica	Coste bajo ($)	Estación mono (%)	Coste alto ($)	Estación color (%)
Carcasa CPU	Láminas metal, plástico			50	2	50	1
	Alimentación y disipación de potencia		0,80$/vatios	55	3	55	1
	Cables, tornillos, tuercas			30	1	30	1
	Cajas de embalaje, manuales			10	0	10	0
	Subtotal			145	7	145	3
Tablero CPU	IU, FPU, MMU, cache			200	9	800	16
	DRAM		150$/MB	1200	56	2400	48
	Lógica vídeo (buffer, DAC, mono/color)	Mono		100	5		
		Color				500	10
	Interfaces de E/S (SCSI, Ethernet, flotante, PROM, reloj de fecha			100	5	100	2
	Tablero de circuito impreso		8 capas 1,00$/pulgada^2				
			6 capas 0,50$/pulgada^2	50	2	50	1
			4 capas 0,25$/pulgada^2				
	Subtotal			1650	77	3850	76
Dispositivos de E/S	Teclado, ratón			50	2	50	1
	Pantalla monitor	Mono		300	14		
		Color				1,000	20
	Disco duro		100 MB	400			
	Unidad de cinta		150 MB	400			
Estación de trabajo mono	(8 MB, lógica mono & pantalla, teclado, ratón, sin disco)			2 145	100	2 745	
Estación de trabajo color	(16 MB, lógica color & pantalla, teclado, ratón, sin disco)			4 445		5 045	100
Servidor de ficheros	(16 MB, 6 discos + unidad de cinta			5 595		6 195	

FIGURA 2.13 Coste estimado de los componentes de una estación de trabajo en 1990 suponiendo 100 000 unidades. IU se refiere a la unidad entera del procesador, FPU a la unidad de punto flotante, y MMU a la unidad de gestión de memoria. La columna de coste más bajo se refiere a las opciones menos caras, como por ejemplo la estación de trabajo monocolor que aparece en la tercera fila empezando por el fondo. La columna de coste más alto se refiere a las opciones más caras, como por ejemplo la estación de trabajo en color que aparece en la segunda fila empezando por el fondo. Observar que aproximadamente la mitad del coste de los sistemas está en las DRAM. Cortesía de Andy Bechtolsheim de Sun Microsystems, Inc.

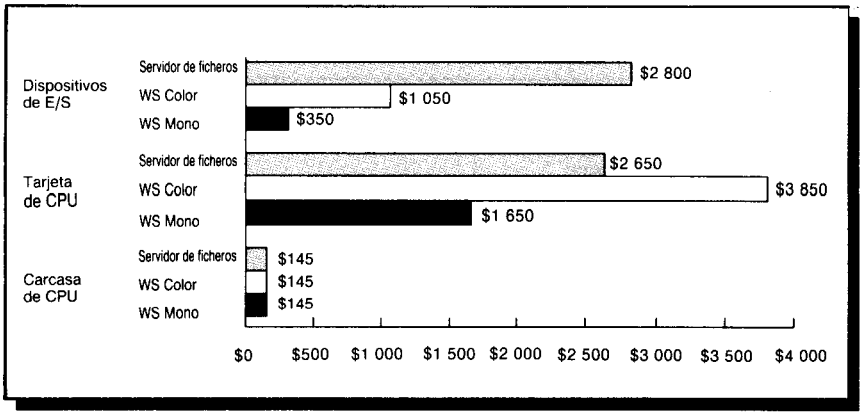


FIGURA 2.14 Los costes de cada máquina de la Figura 2.13 divididos en las tres categorías principales, suponiendo la estimación inferior de coste. Observar que los dispositivos de E/S y la cantidad de memoria contabilizan las principales diferencias en coste.

será interesante considerar dos preguntas: ¿Qué costes puede controlar un ingeniero? Y ¿qué costes puede controlar un ingeniero de computadores?

Coste frente a precio. Por qué y en cuánto se diferencian

Los costes de los componentes pueden limitar los deseos del diseñador, pero están muy lejos de representar lo que el cliente debe pagar. Pero ¿por qué un libro sobre arquitectura de computadores debe contener información sobre precios? Los costes sufren una serie de cambios antes de convertirse en precio, y el diseñador de computadores debe conocerlos para determinar su impacto en las elecciones de diseño. Por ejemplo, cambiar el coste en 1 000 dólares puede cambiar el precio entre 4 000 y 5 000 dólares. Sin comprender la relación entre coste y precio, el diseñador de computadores puede no comprender el impacto sobre el precio al añadir, suprimir, o sustituir componentes.

Las categorías que hacen subir los precios pueden mostrarse como una tasa sobre el coste o como un porcentaje del precio. Examinaremos esta información de ambas formas. La Figura 2.15 muestra el incremento del precio de un producto de izquierda a derecha cuando se añade un nuevo tipo de coste.

Los costes directos se refieren a los costes directamente relacionados con la fabricación de un producto. Estos incluyen costes de mano de obra, compra de componentes, desperdicios (restos de la producción) y garantía, la cual cubre los costes de los sistemas que fallan en el domicilio del cliente durante el período de garantía. Los costes directos, normalmente, suponen un incremento del 25 al 40 por 100 de los costes de los componentes. Los costes de servicio o mantenimiento no están incluidos porque, normalmente, los paga el cliente.

El siguiente incremento se denomina *margen bruto;* los gastos de la com-

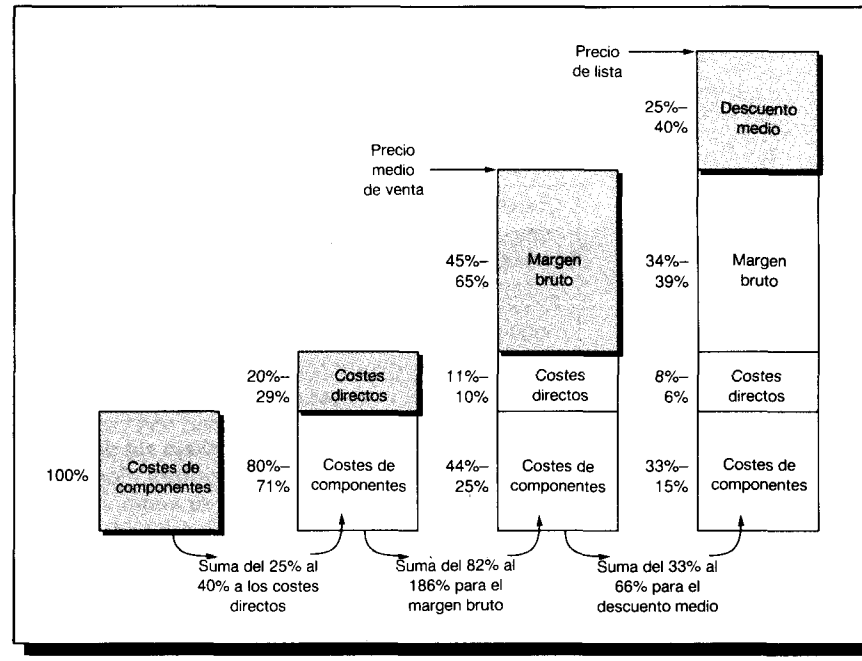


FIGURA 2.15 Comenzando con los costes de los componentes, el precio incrementa cuando se toman en consideración los costes directos, margen bruto, y descuento medio hasta que llegamos al precio de lista. Cada incremento se muestra en la parte inferior como una tasa sobre el precio anterior. A la izquierda de cada columna se muestran los porcentajes del nuevo precio para todos los elementos.

pañía que no pueden ser facturados directamente a un producto. Y puede considerarse como un coste indirecto. Incluye los gastos de investigación y desarrollo de la compañía (I + D), marketing, ventas, mantenimiento del equipo de fabricación, alquiler de edificios, coste de financiación, beneficios, e impuestos. Cuando los costes de los componentes se multiplican por el coste directo y el margen bruto obtenemos el *precio de venta medio (average selling price)* —ASP en el lenguaje de MBA— el dinero que entra direcctamente en la compañía por cada producto vendido. El margen bruto normalmente es del 45 al 65 por 100 del precio de venta medio.

Precio de lista y precio de venta medio no son lo mismo. La razón es que las compañías ofrecen descuentos en volumen, bajando el precio de venta medio. Además, si el producto se va a vender en almacenes al «detall», como ocurre con los computadores personales, los almacenes quieren retener para ellos el 40 por 100 del precio de lista. Por tanto, dependiendo del sistema de distribución, el precio medio de venta, normalmente, es del 60 al 75 por 100 del precio de lista. La fórmula siguiente liga estos cuatro términos:

$$\text{Precio de lista} = \frac{\text{coste} \cdot (1 + \text{costes directos})}{(1 - \text{descuento medio}) \cdot (1 - \text{margen bruto})}$$

	Modelo A ($)	Como % de costes	Como % de precios de lista	Modelo B ($)	Como % de costes	Como % de precios de lista
Costes de los componentes	2 145	100	27	2 145	100	21
Costes de los componentes + costes directos	2 681	125	33	3 003	140	30
Precio medio de venta (suma margen bruto)	5 363	250	67	7 508	350	75
Precio de lista	8 044	375	100	10 010	467	100

FIGURA 2.16 La estación de trabajo sin discos de la Figura 2.13 presupuestada utilizando dos modelos diferentes. Por cada dólar de incremento del coste de componentes, el precio medio de venta sube entre 2,50 y 3,50 dólares, y el precio de lista aumenta entre 3,75 y 4,67 dólares.

La Figura 2.16 muestra los conceptos abstractos de la Figura 2.15 utilizando dólares y centavos al convertir los costes de la Figura 2.13 en precios. Esto se hace utilizando dos modelos comerciales. El modelo A supone el 25 por 100 (del coste) de costes directos, el 50 por 100 (de ASP) de margen bruto, y un 33 por 100 (del precio de lista) de descuento medio. El modelo B supone el 40 por 100 de costes directos, el 60 por 100 de margen bruto, y el descuento medio cae al 25 por 100.

El precio es sensible a la competencia. Una compañía que se esfuerce por ganar mercado puede, por tanto, ajustar el descuento medio o los beneficios, pero debe vivir con sus costes de componentes y directos, más los costes del margen bruto.

Muchos ingenieros se sorprenden al enterarse que la mayoría de las compañías gastan solamente del 8 al 15 por 100 de sus rentas en I+D, donde se incluye toda la ingeniería (excepto la manufactura e ingeniería de campo). Este es un porcentaje bien establecido que aparece en los informes anuales de la compañía y está tabulado en las revistas nacionales, por tanto este porcentaje es improbable que cambie en el tiempo.

La información anterior sugiere que una compañía aplica uniformemente porcentajes de gastos fijos para convertir el coste en precio, lo cual es cierto para muchas compañías. Pero otro punto de vista es que los gastos de I + D deben ser considerados como una inversión, y así una inversión entre el 8 y el 15 por 100 de renta significa que cada dólar gastado en I + D puede generar de 7 a 13 dólares en ventas. Este punto de vista alternativo sugiere entonces un margen bruto diferente para cada producto, dependiendo del número de ventas y de la magnitud de la inversión. Las grandes máquinas generalmente tienen más costes de desarrollo —una máquina que cueste diez veces lo que vale su manufactura puede tener un coste de desarrollo muchas veces mayor. Como las máquinas grandes generalmente no se venden tan bien como

las pequeñas, el margen bruto debe ser mayor en aquellas para que la compañía obtenga un rendimiento rentable de su inversión. Este modelo de inversión hace que las máquinas grandes se encuentren sometidas a un riesgo doble —porque se venden menos y requieren mayores costes de I + D—, y justifican una mayor relación de precio/coste frente a las máquinas más pequeñas.

2.4 Juntando todo: precio/rendimiento de tres máquinas

Una vez estudiados rendimientos y costes, el paso siguiente es medir el rendimiento de programas reales en máquinas reales e indicar los costes de estas máquinas. Por desgracia, los costes son difíciles de calcular por lo que, en su lugar, utilizaremos los precios. Comenzamos con la controvertida relación de precio/rendimiento.

Nombre del programa	Compilador Gnu C para 68000	Common TeX	Spice
Versión	1,26	2,9	2G6
Líneas	79 409	23 037	18 307
Opciones	-O	'&latex/lplain'	análisis transitorio, pasos de 200 ps, para 40 ns
Entrada	i*.c	bit-set.tex, compilador, tex,...	digital - registro de desplazamiento
Líneas/bytes de entrada	28 009/373 668	10 992/698 914	233/1294
Líneas/bytes de salida	47 553/664 479	758/524 728	656/4172
% de operaciones en punto flotante (on the DECstation 3100)	0,01%	0,05%	13,58%
Lenguaje de programación	C	C	FORTRAN 66
Propósito	Licencia pública, portable, compilador C de optimización	Formatear documentos	Análisis de circuitos ayudado por computador

FIGURA 2.17 Programas utilizados en este libro para medidas de rendimiento. El compilador Gnu C es un producto de «Free Software Foundation» y, por razones no limitadas a su precio, es preferido por algunos usuarios a los compiladores suministrados por el fabricante. Sólo 9 540 de las 79 409 líneas son específicas para el 68000, y existen versiones para VAX, SPARC, 88000, MIPS, y algunos otros repertorios de instrucciones. La entrada para GCC son los ficheros fuente del compilador que comienzan con la letra «i». Common TeX es una versión C del programa de tratamiento de documentos escrito originalmente por el profesor Donald Knuth de Stanford. La entrada es un conjunto de páginas de manual para el compilador SUIF de Stanford. Spice es un paquete de análisis de circuitos ayudado por computador distribuido por la Universidad de California en Berkeley. (Estos programas y sus entradas están disponibles como parte del paquete software asociado a este libro. El Prefacio menciona la forma de obtener una copia.)

La Figura 2.17 indica los programas elegidos por los autores para medir rendimientos en este libro. Dos de los programas casi no tienen operaciones en punto flotante, y el otro tiene una cantidad moderada de operaciones en punto flotante. Los tres programas tienen entradas, salidas, y opciones —que es lo que se puede esperar de los programas reales. Cada programa tiene, en

	VAXstation 2000	VAXstation 3100	DECstation 3100
Año de introducción	1987	1989	1989
Versión de CPU/FPU	μVAXII	CVAX	MIPS R2000A/R2010
Frecuencia de reloj	5 MHz	11,11 MHz	16,67 MHz
Tamaño de memoria	4 MB	8 MB	8 MB
Tamaño de cache	ninguno	1 KB en chip, 64-KB segundo nivel	128 KB (dividir 64-KB instrución y 64-KB datos)
Tamaño TLB	8 entradas completamente asociativa	28 entradas completamente asociativa	64 entradas completamente asociativa
Precio base de lista	4 825$	7 950$	11 950$
Equipamiento opcional	monitor 19", extra 10 MB	(modelo 40) extra 8 MB	monitor 19", extra 8 MB
Precio de lista	15 425$	14 480$	17 950$
Rendimiento según marketing	0,9 MIPS	3,0 MIPS	12 MIPS
Sistema operativo	Ultrix 3.0	Ultrix 3.0	Ultrix 3.0
Versión compilador C	Ultrix y VMS	Ultrix y VMX	1.31
Opciones para compilador C	-O	-O	-O2 -Olimit 1060
Biblioteca C	libc	libc	libc
Versión compilador FORTRAN 77	fort (VMS)	fort (VMS)	1.31
Opciones para compilador FORTRAN 77	-O	-O	-O2 -Olimit 1060
FORTRAN 77	lib*77	lib*77	lib*77

FIGURA 2.18 Las tres máquinas y el software utilizado para medir el rendimiento de la Figura 2.19. Estas máquinas las vende Digital Equipment —en efecto, la DECstation 3100 y la VAXstation 3100 fueron anunciadas el mismo día. Las tres son estaciones de trabajo sin disco y corren las mismas versiones del sistema operativo UNIX, llamado Ultrix. Los compiladores VMS transportados a Ultrix fueron utilizados para TeX y Spice en las VAXstations. Utilizamos el compilador nativo Ultrix C para GCC, debido a que GCC no corría utilizando el compilador C VMS. Los compiladores para la DECstation 3100 son suministrados por MIPS Computer Systems. (La opción «-Olimit 1060» para la DECstation 3100 indica al compilador que no trate de optimizar procedimientos mayores de 1 060 líneas.)

efecto, una gran comunidad de usuarios interesados en la rapidez con que se ejecutan. (Al medir el rendimiento de las máquinas nos habría gustado presentar una muestra mayor, pero nos hemos limitado a tres en este libro para hacer comprensibles las tablas y los gráficos.)

La Figura 2.18 muestra las características de tres máquinas donde realizamos las medidas, incluyendo la lista de precios y el rendimiento relativo calculado por marketing.

La Figura 2.19 muestra los tiempos de CPU y transcurrido medidos para estos programas. Se incluyen tiempos totales y algunas medias ponderadas, con los pesos entre paréntesis. La primera media aritmética ponderada supone una carga de trabajo sólo de programas que operan con enteros (GCC y TeX). La segunda supone pesos para una carga de trabajo de punto flotante (Spice). Las tres siguientes medias ponderadas son para tres cargas de trabajo que emplean el mismo tiempo para cada programa en una de las máquinas (ver Fig. 2.7). Las únicas medias significativamente diferentes son la entera y la de punto

	VAXstation 2000		VAXstation 3100		DECstation 3100	
	Tiempo CPU	Tiempo transcurrido	Tiempo CPU	Tiempo transcurrido	Tiempo CPU	Tiempo transcurrido
Compilador C Gnu 68000	985	1108	291	327	90	159
Common TeX	1264	1304	449	479	95	137
Spice	958	973	352	395	94	132
Media aritmética	1069	1128	364	400	93	143
MA ponderada - sólo entero (50% GCC, 0% TeX, 100% Spice)	1125	1206	370	403	93	148
MA ponderada - sólo punto flotante (0% GCC, 0% TeX, 100% Spice)	958	973	352	395	94	132
MA ponderada - igual tiempo de CPU en V2000 (35,6% GCC, 27,8% TeX, 36,6% Spice)	1053	1113	357	394	93	143
MA ponderada AM - igual tiempo de CPU en V3100 (40,4% GCC, 26,2% TeX, 33,4% Spice)	1049	1114	353	390	93	144
MA ponderada AM - igual tiempo de CPU en D3100 (34,4% GCC, 32,6% TeX, 33,0% Spice)	1067	1127	363	399	93	143

FIGURA 2.19 Rendimiento de los programas de la Figura 2.17 en las máquinas de la Figura 2.18. Los pesos corresponden sólo a programas enteros, y después igual tiempo de CPU corriendo en cada una de las tres máquinas. Por ejemplo, si la mezcla de los tres programas fuese proporcionada a los pesos en la fila «igual tiempo de CPU en D3100», la DECstation 3100 emplearía un tercio de su tiempo de CPU corriendo el Compilador Gnu C, otro tercio corriendo TeX, y el otro tercio corriendo Spice. Los pesos utilizados están entre paréntesis, calculados como se muestra en la Figura 2.7.

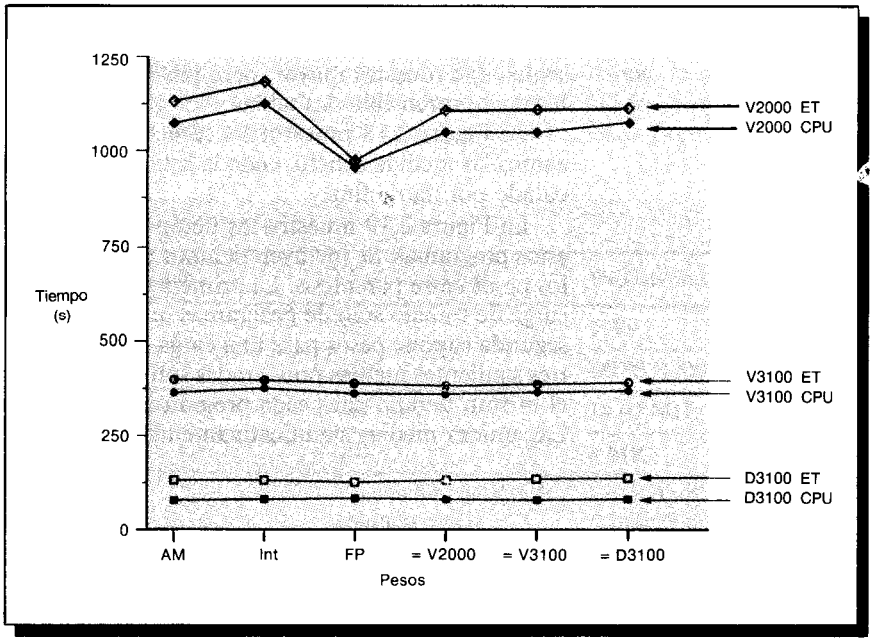


FIGURA 2.20 Dibujo de medias de tiempo de CPU y de tiempo transcurrido de la Figura 2.19.

flotante para las VAXstation 2000 y 3100. El resto de las medias para cada máquina está en el 8 por 100 de las otras, como puede verse en la Figura 2.20, donde se representan las medias ponderadas.

El punto definitivo para muchos compradores de computadores es el precio que pagan por el rendimiento. Esto se representa gráficamente en la Figura 2.21, donde aparecen las medias aritméticas del tiempo de CPU frente al precio de cada máquina.

2.5 Falacias y pifias

Las falacias y pifias sobre el coste/rendimiento han hecho caer en trampas a muchos arquitectos de computadores, entre los que nos incluimos. Por esta razón, se dedica más espacio a la sección de advertencias en este capítulo que en los demás capítulos de este texto.

Falacia: Las métricas independientes del hardware predicen el rendimiento.

Debido a que, predecir con precisión el rendimiento, es muy difícil, el folklore del diseño de computadores está lleno de atajos sugestivos. Estos se emplean

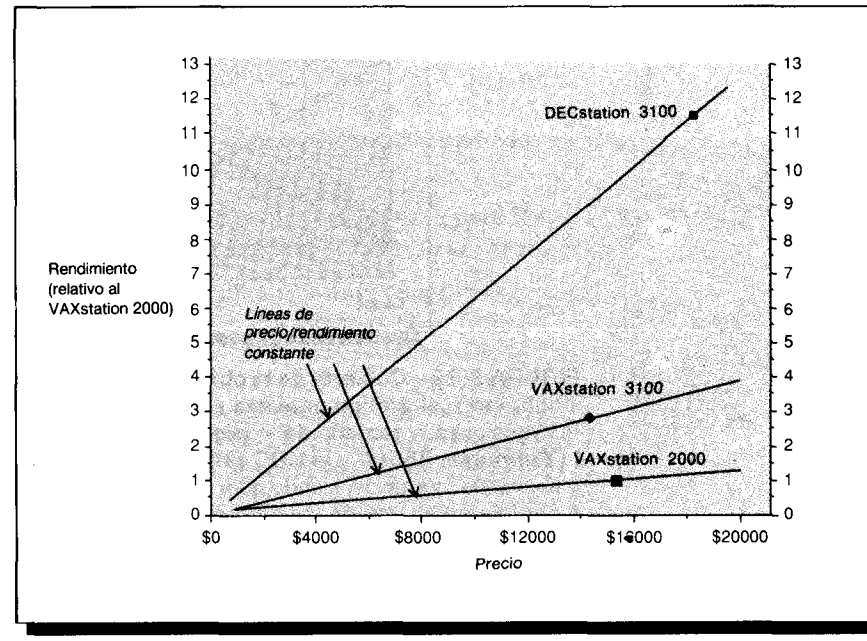


FIGURA 2.21 Precio frente a rendimiento de las VAXstation 2000, VAXstation 3100, y DECstation 3100 para el compilador Gnu C, TeX, y Spice. Basándose en las Figuras 2.18-2.19, en esta figura se dibuja el precio de lista de una máquina frente al rendimiento, donde el rendimiento es la inversa de la razón a la media aritmética del tiempo de CPU en una VAXstation 2100. Las tres máquinas muestran líneas de precio/rendimiento constante. Por ejemplo, una máquina en el extremo derecho de la línea de la VAXstation 3100 cuesta 20 000 dólares. Si costase el 30 por 100 más, debería tener el 30 por 100 más de rendimiento que la VAXstation 3100 para tener el mismo rendimiento de precios.

frecuentemente cuando se comparan diferentes repertorios de instrucciones, especialmente los que son diseños sobre papel.

Uno de dichos atajos es «Tamaño de Código = Velocidad», o la arquitectura con el programa más pequeño es la más rápida. El tamaño del código estático es importante cuando el espacio de memoria está muy solicitado, pero no es lo mismo que rendimiento. Como veremos en el Capítulo 6, los programas más grandes, compuestos de instrucciones que se buscan, decodifican, y ejecutan fácilmente pueden correr con más rapidez que máquinas con instrucciones extremadamente compactas que son difíciles de decodificar. «Tamaño de Código = Velocidad» es especialmente popular entre los escritores de compiladores, ya que mientras puede ser difícil decidir si una secuencia de código es más rápida que otra, es fácil ver que es más corta.

La evidencia de la falacia «Tamaño de Código = Velocidad» puede encontrarse en la cubierta del libro *Assessing the Speed of Algol 60*, en la Figura 2.22. Los programas del CDC 6600 son unas dos veces más grandes; sin embargo, la máquina CDC ejecuta los porgramas Algol 60, casi seis veces

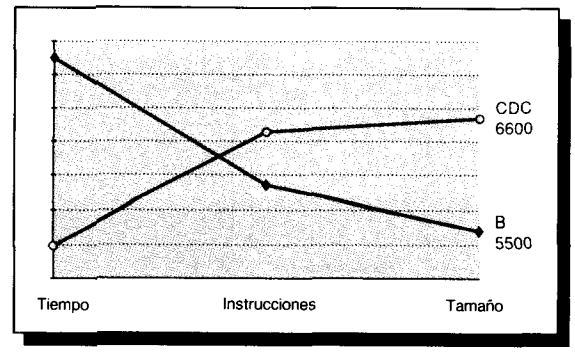


FIGURA 2.22 Cubierta de la obra *Assessing the Speed of Algol 60* por B. A. Wichmann, el gráfico muestra el tiempo relativo de ejecución, recuento de instrucciones, y tamaño del código de los programas escritos en Algol 60 para la Burroughs B5500 y la CDC 6600. Los resultados están normalizados para una máquina de referencia, siendo peor con un número más alto. Este libro tuvo un profundo efecto en uno de los autores (DP). Seymour Cray, el diseñador de la CDC 6600, puede no haber conocido la existencia de este lenguaje de programación, aunque Robert Barton, arquitecto de la B5500, diseñó específicamente el repertorio de instrucciones para Algol 60. Mientras la CDC 6600 ejecuta el 50 por 100 más de instrucciones y tiene un código el 220 por 100 mayor, la CDC 6600 es el 550 por 100 más rápida que la B5500.

con más **rapidez** que la Burroughs B5500, una máquina diseñada para Algol 60.

Pifia: Comparar computadores utilizando solamente una o dos de las tres métricas de rendimiento: frecuencia de reloj, CPI, y recuento de instrucciones.

La ecuación del rendimiento de la CPU muestra por qué esto puede ser erróneo. Un ejemplo es él de la Figura 2.22: el CDC 6600 ejecuta casi un 50 por 100 más instrucciones que la Burroughs B5500, aunque es el 550 por 100 más rápida. Otro ejemplo proviene de incrementar la frecuencia de reloj para que algunas instrucciones se ejecuten con más rapidez —a veces denominado *rendimiento de pico (o rendimiento máximo)*— pero tomando decisiones de diseño que producen un CPI global alto que contrarresta la ventaja de la frecuencia de reloj. La Intergraph Clipper C100 tiene una frecuencia de reloj de 33 MHz y un rendimiento máximo de 33 MIPS nativos. Sin embargo, Sun 4/280, con la mitad de la frecuencia de reloj y la mitad de los MIPS nativos estimados, corre los programas más rápidamente [Hollingsworth, Sachs, y Smith 1989, 215]. Como el recuento de instrucciones de la Clipper es aproximadamente la misma que la de la Sun, el CPI de la primera máquina debe ser más del doble que él de la última.

Falacia: Cuando se calculan MIPS relativos, las versiones del compilador y sistema operativo de la máquina de referencia tienen poca importancia.

La Figura 2.19 muestra que la VAXstation 2000 emplea 958 segundos de tiempo de CPU cuando se corre Spice con una entrada estándar. En lugar de

Ultrix 3.0 con el compilador VMS F77, muchos sistemas utilizan Ultrix 3.0 con el compilador estándar UNIX F77. Este compilador incrementa el tiempo de CPU de Spice a 1604 segundos. Utilizando la evaluación estándar de 0.9 MIPS relativos para la VAXstation 2000, la DECstation 3100 tiene 11 ó 19 MIPS relativos para Spice, dependiendo del compilador de la máquina de referencia.

Falacia: El CPI puede calcularse a partir de la mezcla de instrucciones y de los tiempos de ejecución de las instrucciones que aparecen en el manual.

Las máquinas actuales son muy complicadas para estimar su rendimiento a partir de un manual. Por ejemplo, en la Figura 2.19 Spice emplea 94 segundos de tiempo de CPU en la DECstation 3100. Si calculamos el CPI a partir del manual de la DECstation 3100 —ignorando la jerarquía de memoria y las ineficiencias de la segmentación para esta mezcla de instrucciones de Spice— obtenemos un CPI de 1.41. Cuando se multiplica por la cuenta de instrucciones y la frecuencia de reloj, solamente se obtienen 73 segundos. El 25 por 100 del tiempo de CPU que falta se debe a la estimación del CPI basada solamente en el manual. El valor real medido, incluyendo todas las ineficiencias del sistema de memoria es un CPI de 1.87.

Pifia: Resumir el rendimiento convirtiendo la productividad en tiempo de ejecución.

Los benchmarks de SPEC informan sobre el rendimiento midiendo el tiempo transcurrido para cada uno de los 10 benchmarks. La única estación de trabajo de doble procesador en el informe inicial de los benchmarks no los ejecutaba más rápidamente, ya que los compiladores no paralelizaban automáticamente el código en los dos procesadores. La solución fue correr una copia de cada benchmark en cada procesador y registrar el tiempo transcurrido para las dos copias. Esto no habría servido de ayuda si la versión de SPEC solamente hubiese resumido el rendimiento utilizando tiempos transcurridos, ya que los tiempos eran más lentos debido a la interferencia de los procesadores en los accesos a memoria. El pretexto fue que la versión inicial de SPEC daba medias geométricas del rendimiento relativo para una VAX-11/780 además de los tiempos transcurridos, y estas medias se utilizaban para representar los resultados. Esta innovación interpretaba la relación de rendimientos en una VAX-11/780 como una medida de la **productividad,** ¡así duplicaba sus relaciones medidas para la VAX! La Figura 2.23 muestra las representaciones encontradas en el informe para el uniprocesador y el multiprocesador. Esta técnica casi duplica las medias geométricas de las relaciones, sugiriendo la conclusión errónea de que un computador que ejecuta simultáneamente dos copias de un programa tiene el mismo tiempo de respuesta para un usuario que un computador que ejecuta un solo programa en la mitad de tiempo.

Falacia: Los benchmarks sintéticos predicen el rendimiento.

Los ejemplos mejor conocidos de estos benchmarks son Whetstone y Dhrystone, que no son programas reales y, como tales, no pueden reflejar el comportamiento de un programa para factores no medidos. Las optimizaciones de los compiladores y el hardware pueden inflar artificialmente el rendimiento de estos benchmarks pero no el de los programas reales. La otra cara

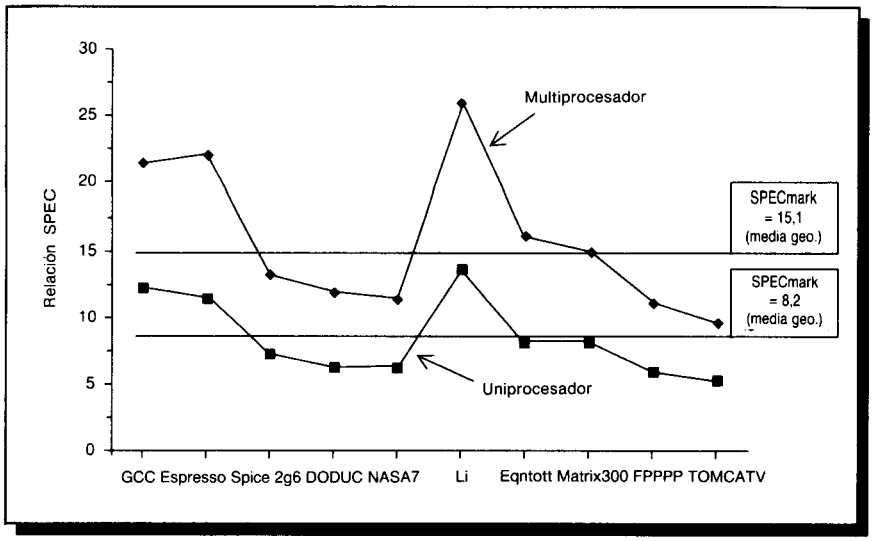


FIGURA 2.23 Rendimiento del uniprocesador y multiprocesador según la versión de prensa del benchkmark de SPEC. El rendimiento se dibuja con relación a una VAX-11/780. El cociente para el multiprocesador es realmente la razón del tiempo transcurrido multiplicado por el número de procesadores.

de la moneda es que como estos benchmarks no son programas naturales, no recompensan las optimizaciones de comportamiento que se presentan en los programas reales. Aquí se dan algunos ejemplos:

- Los compiladores optimizadores pueden descargar el 25 por 100 del código de Dhrystone; los ejemplos incluyen bucles que sólo se ejecutan una vez, haciendo innecesarias las instrucciones extras debidas al bucle. Para controlar estos problemas, los autores del benchmark «requieren» informes tanto sobre código optimizado como no optimizado. Además, «prohíben» la práctica de la optimización de expansión de procedimientos en línea. (La estructura de sencillos procedimientos de Dhrystone permite la eliminación de todas las llamadas a procedimientos casi sin incrementar el tamaño del código; ver Figura 2.5.)
- Todos los bucles de punto flotante de Whetstone hacen esencialmente inútiles las optimizaciones vía vectorización. (El programa se escribió antes que fuesen populares los computadores con instrucciones vectoriales; ver Capítulo 7.)
- Dhrystone tiene una larga historia de optimizaciones que modifican su rendimiento. La más reciente procede de un compilador C creado para que incluyese optimizaciones para Dhrystone (Fig. 2.5). Si el señalizador de opción adecuado se inicializa en tiempo de compilación, el compilador toma la parte de la versión C de este benchmark que copia una cadena de bytes de longitud variable (terminada con un símbolo fin de cadena) en un bucle

que transfiere un número fijo de palabras suponiendo que la fuente y destino de la cadena es una palabra alineada de memoria. Aunque se estima que entre el 99,70 y el 99,98 por 100 de copias de cadenas **no** podrían utilizar esta optimización, este simple cambio puede suponer de un 20 a un 30 por 100 de mejora en el rendimiento global —si Dhrystone es su medida.

- Los compiladores pueden optimizar una pieza clave del bucle de Whetstone observando la relación entre la raíz cuadrada y la exponencial, aun que esto es muy improbable que ocurra en los programas reales. Por ejemplo, un bucle clave contiene el siguiente código FORTRAN (ver Fig. 2.4):

```
X = SQRT( EXP (ALOG(X)/T1) )
```

Esto podría ser compilado como si fuese

```
X = EXP( ALOG(X)/2*T1) )
```

puesto que

$$\text{SQRT(EXP(X))} = \sqrt[2]{e^x} = e^{x/2} = \text{EXP (X/2)}$$

Sería sorprendente que se invocasen siempre estas optimizaciones excepto en este benchmark sintético. (¡Sin embargo, uno de los revisores de este libro encontró algunos compiladores que realizaban esta optimización!) Este sencillo cambio convierte todas las llamadas a la función raíz cuadrada en Whetstone en multiplicaciones por 2, mejorando seguramente el rendimiento —si Whetstone es su medida.

Falacia: El rendimiento máximo se asemeja al rendimiento observado.

Una definición del rendimiento máximo es el rendimiento que una máquina tiene «garantizado no sobrepasar». El salto entre rendimiento máximo y rendimiento observado, normalmente, es de un factor mayor o igual que 10 en los supercomputadores. (Ver Cap. 7 sobre vectores para una explicación.) Como el salto es tan grande, el rendimiento máximo no es útil para predecir rendimientos observados a menos que la carga de trabajo conste de pequeños programas que, normalmente, operen próximos al pico (máximo).

Como ejemplo de esta falacia, un pequeño segmento de código que tenía grandes vectores se ejecutó en el Hitachi S810/20 a 236 MFLOPS y en el CRAY X-MP a 115 MFLOPS. Aunque esto sugiere que el S810 es el 105 por 100 más rápido que el X-MP, el X-MP ejecuta un programa con longitudes más normales de vectores el 97 por 100 más rápido que el S810. Estos datos se muestran en la Figura 2.24.

Otro buen ejemplo procede de un conjunto de benchmarks denominado Perfect Club (ver pág. 85). La Figura 2.25 muestra la estimación máxima de MFLOPS, la media armónica de los MFLOPS conseguidos para 12 programas reales, y el tanto por ciento del rendimiento máximo para tres grandes computadores. Estos computadores consiguen solamente el 1 por 100 del rendimiento máximo.

	CRAY X-MP	Hitachi S810/20	Rendimiento
A(i)=B(i)*C(i)+D(i)*E(i) (longitud de vector 1 000 realizado 100 000 veces)	2,6 s	1,3 s	Hitachi 105% más rápida
FFT vectorizado (longitud de vector 64, 32,..., 2)	3,9 s	7,7 s	CRAY 97% más rápida

FIGURA 2.24 Medidas del rendimiento máximo y del rendimiento real para el Hitachi S810/20 y el Cray X-MP. De Lubeck, Moore, y Mendez [1985, 18-20]. Ver también la pifia en la sección de Falacias y pifias del Capítulo 7.

Máquina	Estimación máxima de MFLOPS	Media armónica MFLOPS de los benchmarks Perfect	Porcentaje de MFLOPS máximos (de pico)
CRAY X-MP/416	940	14,8	1 %
IBM 3090-600S	800	8,3	1 %
NEC SX/2	1300	16,6	1 %

FIGURA 2.25 Rendimiento máximo y media armónica del rendimiento real para los benchmarks de Perfect. Estos resultados son para los programas ejecutados sin modificaciones. Cuando se ajustan a mano el rendimiento de las tres máquinas se desplaza a 24.4, 11.3 y 18.3 MFLOPS, respectivamente. Esto es todavía el 2 por 100 o menos del rendimiento de pico.

Aunque el rendimiento máximo se ha utilizado mucho en los supercomputadores, recientemente se ha extendido esta métrica a los fabricantes de microprocesadores. Por ejemplo, en 1989 se anunció un microprocesador que tenía un rendimiento de 150 millones de «operaciones» por segundo («MOPS»). La única forma que esta máquina podía lograr este rendimiento era ejecutando una instrucción entera y una instrucción en punto flotante cada ciclo de reloj de forma que la instrucción de punto flotante realizara una operación de multiplicación y otra de suma. Para que este rendimiento máximo prediga el rendimiento observado, un programa real debería tener un 66 por 100 de sus operaciones en punto flotante y ninguna pérdida en el sistema de memoria o de segmentación. En contraste con las afirmaciones, el rendimiento típico medido de este microprocesador está por debajo de 30 «MOPS».

Los autores esperan que el rendimiento máximo pueda ponerse en cuarentena en la industria de supercomputadores y eventualmente erradicarlo de ese campo; pero, en cualquier caso, aproximarse al rendimiento de los supercomputadores no es una excusa para adoptar hábitos dudosos en el marketing de supercomputadores.

2.6 Observaciones Finales

Tener un estándar sobre informes de rendimientos en las revistas de informática tan alto como el de las revistas de automóviles sería una mejora en la práctica actual. Siendo optimistas, eso ocurrirá cuando la industria realice evaluaciones del rendimiento basándose en programas reales. Quizá entonces los argumentos sobre el rendimiento, sean incluso menos violentos.

Los diseños de computadores se medirán siempre por el coste y el rendimiento, y encontrar el mejor equilibrio siempre será el arte del diseño de computadores. Si la tecnología continúa mejorando rápidamente, las alternativas serán como las curvas de la Figura 2.26. Una vez que un diseñador seleccione una tecnología, no se pueden conseguir determinados niveles de rendimiento por mucho que se pague e, inversamente, por mucho que disminuya el rendimiento hay un límite de cuánto puede bajar el coste. En estos casos sería mejor cambiar las tecnologías.

Como nota final, el número de máquinas vendidas no es siempre la mejor medida de la relación coste/rendimiento de los computadores, ni la relación coste/rendimiento predice siempre el número de ventas. El marketing es muy importante para las ventas. Sin embargo, es más fácil vender una máquina con mejor relación coste/rendimiento. Incluso los negocios con elevados márgenes brutos necesitan ser sensibles al coste/rendimiento, ya que si no la compañía no podría bajar los precios cuando se enfrente con competidores inflexibles. ¡A menos que el lector se dedique al marketing, su trabajo consistirá en mejorar la relación coste/rendimiento!

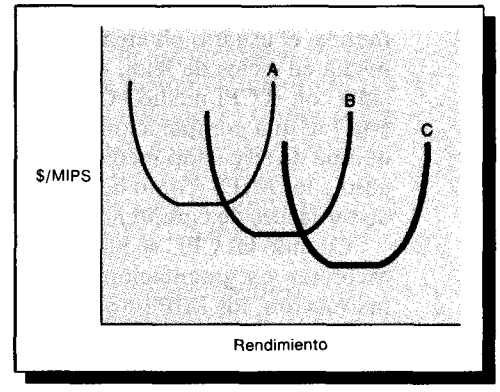


FIGURA 2.26 El coste por MIPS se representa en el eje y, y el rendimiento del sistema aumenta sobre el eje x. A, B y C son tres tecnologías, digamos tres tecnologías de semiconductores diferentes, para construir un procesador. Los diseños de la parte plana de las curvas pueden ofrecer diversidad de rendimientos para la misma relación coste/rendimiento. Si los objetivos del rendimiento son muy altos para una tecnología, llegan a ser muy caros, y un diseño muy barato hace que el rendimiento sea muy bajo (coste por MIPS caro para MIPS bajos). En casos extremos es mejor cambiar de tecnología.

2.7 Perspectiva histórica y referencias

El anticipado grado de solapamiento, buffering y colas en el Modelo 85 [IBM 360] [primer computador con cache] apareció para invalidar en gran parte las medidas conveniconales de rendimiento basadas en mezclas de instrucciones y núcleos (kernels) de programas.

Conti, Gibson y Pitkowsky [1968]

En los primeros días de la computación, los diseñadores perseguían objetivos de rendimiento —la ENIAC era mil veces más rápida que la Mark I de Hardvard, y la IBM Stretch (7030) era cien veces más rápida que la máquina más rápida que existiese. Lo que no estaba claro, sin embargo, era cómo se medía este rendimiento. Al echar una mirada retrospectiva, se observa que cada generación de computadores deja obsoletas las técnicas de evaluación del rendimiento de la generación anterior.

La medida original de rendimiento fue el tiempo que se tardaba en realizar una operación individual, como por ejemplo la suma. Como la mayor parte de las instrucciones tenían el mismo tiempo de ejecución, el tiempo de una daba idea de los otros. Sin embargo, cuando los tiempos de ejecución de las instrucciones de una máquina fueron menos uniformes, el tiempo de una operación no era muy útil para realizar comparaciones. Para tener en cuenta estas diferencias, se calculó una *mezcla de instrucciones* midiendo la frecuencia relativa de las instrucciones en un computador a través de muchos programas. La mezcla de Gibson [1970] fue una primera mezcla de instrucciones muy popular. Al multiplicar el tiempo de cada instrucción por su peso en la mezcla, el usuario obtenía el *tiempo de ejecución medio por instrucción.* (Si se medía en ciclos de reloj, el tiempo de ejecución medio por instrucción coincidía con el CPI medio.) Como los repertorios de instrucciones eran similares, ésta fue una comparación más precisa que la de tiempos de suma. Desde el tiempo de ejecución medio por instrucción, por tanto, había sólo un pequeño paso a los MIPS (como hemos visto, son inversos). Los MIPS tienen la virtud de ser fáciles de comprender para el profano, y de ahí su popularidad.

Cuando las CPU se hicieron más sofisticadas y contaron con jerarquías de memoria y segmentación, dejó de haber un único tiempo de ejecución por instrucción; los MIPS no se podían calcular a partir de la mezcla y del manual. El siguiente paso fue la utilización de *núcleos (kernels)* y programas sintéticos. Curnow y Wichmann [1976] crearon el programa sintético Whetstone para medir programas científicos escritos en Algol 60. Este programa se redactó en FORTRAN y se utilizó ampliamente para caracterizar el rendimiento de programas científicos. Un esfuerzo para lograr objetivos similares a los de Whetstone, «Livermore FORTRAN Kernels», fue realizado por McMahon [1986] y los investigadores del Lawrence Livermore Laboratory en un intento de establecer un benchmark para supercomputadores. Sin embargo, estos *kernels* estaban formados por bucles de programas reales.

La noción de MIPS relativos se presentó como una forma de resucitar la

estimación fácilmente comprensible de los MIPS. Cuando el VAX-11/780 estaba listo para su presentación en 1977, DEC ejecutó benchmarks que corrían también en un IBM 370/158. El marketing de IBM referenció el 370/158 como un computador de 1-MIPS, y como los programas corrían a la misma velocidad, el marketing de DEC denominó al VAX-11/780 un computador de 1-MIPS. (Obsérvese que esta estimación incluía la eficacia de los compiladores en ambas máquinas en el momento en que se realizó la comparación.) La popularidad del VAX-11/780 lo convirtió en una máquina popular de referencia para los MIPS relativos, especialmente porque los MIPS relativos o un computador de 1-MIPS son fáciles de calcular: si una máquina era cinco veces más rápida que el VAX-11/780, para ese benchmark su estimación sería de 5 MIPS relativos. La estimación de 1-MIPS fue incuestionada durante cuatro años hasta que Joel Emer de DEC midió el VAX-11/780 bajo una carga de tiempo compartido. Encontró que la estimación en MIPS nativos del VAX-11/780 era de 0,5. Los VAX posteriores que corrían a 3 MIPS nativos para algunos benchmarks fueron denominadas por tanto máquinas de 6 MIPS porque eran seis veces más rápidos que el VAX-11/780.

Aunque otras compañías seguían esta práctica confusa, los eruditos han redefinido los MIPS como «Indicación sin sentido de la velocidad del procesador» *(Meaningless Indication of Processor Speed)* o «Adoctrinamiento sin sentido para vendedores agresivos» *(Meaningless Indoctrination by Pushy Salerpersons)*. En el instante actual, el significado más común de MIPS en la literatura de marketing no es el de los MIPS nativos sino «número de veces más rápido que el VAX-11/780» y también incluye frecuentemente programas en punto flotante. La excepción es IBM, que define los MIPS relativos a la «capacidad de procesamiento» de un IBM 370/158, presumiblemente corriendo benchmarks de grandes sistemas (ver Henly y McNutt, [1989, 5]). A finales de los ochenta, DEC comenzó a utilizar *unidades VAX de rendimiento* (VUP), significando la relación al VAX-11/780, así 6 MIPS relativos significan 6 VUP.

Los años setenta y ochenta marcaron el crecimiento de la industria de los supercomputadores, que fueron definidos por el alto rendimiento en programas intensivos de punto flotante. El tiempo medio de instrucción y los MIPS eran, claramente, métricas inapropiadas para esta industria, de ahí la invención de los MFLOPS. Desgraciadamente, los compradores olvidaron rápidamente el programa usado para la estimación, y los grupos de marketing decidieron empezar a citar MFLOPS máximos en las guerras sobre el rendimiento de los supercomputadores.

Se han propuesto diversas medidas para promediar el rendimiento. McMahon [1986] recomienda la media armónica para promediar los MFLOPS. Flemming y Wallace [1986] defienden, en general, los méritos de la media geométrica. La réplica de Smith [1988] a su artículo da argumentos convincentes para las medias aritméticas de tiempo y las medias armónicas de frecuencia. (Los argumentos de Smith son los seguidos en el epígrafe «Comparación y resumen de rendimientos» de la Sección 2.2 anterior).

Como por la comunidad de la computación se extendía la distinción entre arquitectura e implementación (ver Cap. 1), surgió la pregunta de si podría evaluarse el rendimiento de una arquitectura, en contraposición a una implementación de la arquitectura. Un estudio de esta pregunta realizado en la

Universidad Carnegie-Mellon aparece resumido en Fuller y Burr [1977]. Se inventaron tres medidas cuantitativas para examinar arquitecturas:

S Número de bytes del código del programa

M Número de bytes transferidos entre memoria y CPU durante la ejecución de un programa para código y datos (S mide el tamaño de código en tiempo de compilación, mientras que M es el tráfico de memoria durante la ejecución del programa)

R Número de bytes transferidos entre registros en un modelo canónico de CPU

Una vez tomadas estas medidas, se les aplicó un factor de peso para determinar qué arquitectura era «mejor». Todavía no ha habido esfuerzos formales para ver si estas medidas realmente importan —¿las implementaciones de una arquitectura con medidas superiores de S, M y R son mejores que arquitecturas inferiores? La arquitectura VAX se diseñó en la cima de la popularidad del estudio de Carnegie-Mellon, y para aquellas medidas va muy bien. Sin embargo, las arquitecturas creadas desde 1985 tienen medidas más pobres que el VAX, y a pesar de todo sus implementaciones van bien frente a las implementaciones VAX. Por ejemplo, la Figura 2.27 compara S, M y el tiempo de CPU para la VAXstation 3100, que utiliza el repertorio de instrucciones VAX, y la DECstation 3100 que no lo utiliza. La DECstation 3100 es del 300 1 casi 500 por 100 más rápida, aún cuando su medida S sea del 35 por 100 al 70 por 100 peor y su medida M sea del 5 al 15 por 100 peor. El esfuerzo para evaluar arquitecturas independientemente de la implementación fue valeroso, parece, aunque no próspero.

Un desarrollo esperanzador en la evaluación del rendimiento es la formación del grupo «Cooperativa de Evaluación del Rendimiento de Sistemas», o *(System Performance Evaluation Cooperative)* o SPEC, en 1988. SPEC tiene representantes de muchas compañías de computadores —los fundadores fue-

	S (tamaño de código en bytes)		M (código en megabytes + datos transferidos)		Tiempo de CPU (en segundos)	
	VAX 3100	DEC 3100	VAX 3100	DEC 3100	VAX 3100	DEC 3100
Compilador Gnu C	409 600	688 128	18	21	291	90
Common TeX	158 720	217 088	67	78	449	95
Spice	223 232	372 736	99	106	352	94

FIGURA 2.27 Tamaños de código y tiempos de CPU de la VAXstation 3100 y de la DECstation 3100 para el compilador Gnu C, TeX y Spice. Los programas y máquinas se describen en las Figuras 2.17 y 2.18. Ambas máquinas corren el mismo sistema operativo y fueron anunciadas el mismo día por la misma compañía. La diferencia está en los repertorios de instrucciones, compiladores, duración del ciclo de reloj, y organización. La medida *M* proviene de la Figura 3.33 para entradas más pequeñas que las de la Figura 2.17, pero el rendimiento relativo permanece inalterado. El tamaño de código incluye bibliotecas.

ron Apollo/Hewlett-Packard, DEC, MIPS, y Sun— que llegaron al acuerdo de ejecutar todos un conjunto de programas y entradas reales. Es digno de notar que SPEC no podría haberse creado con anterioridad a los sistemas operativos portables y la popularidad de los lenguajes de alto nivel. Ahora los compiladores, también, son aceptados como parte propia del rendimiento de los sistemas de computadores y deben ser medidos en cualquier evaluación. (Ver Ejercicios 2.8-2.10, para saber más sobre benchmarks de SPEC.)

La historia nos enseña que, aunque el esfuerzo de SPEC es útil con los computadores actuales, no podrá enfrentarse a las necesidades de la siguiente generación. Un esfuerzo similar a SPEC, llamado Perfect Club, vincula a las universidades y compañías interesadas en el cálculo paralelo [Berry y cols. 1988]. En lugar de forzar a correr el código de programas secuenciales existentes, Perfect Club incluye programas y algoritmos, y permite a los miembros escribir nuevos programas en nuevos lenguajes, que pueden ser necesarios para las nuevas arquitecturas. Los miembros del Perfect Club también pueden sugerir nuevos algoritmos para resolver problemas importantes.

Aunque los artículos sobre rendimiento son abundantes, hay pocos artículos disponibles sobre el coste de los computadores. Fuller [1976] escribió el primer artículo comparando precio y rendimiento para el «Annual International Symposium on Computer Architecture». En esta conferencia también fue éste el último artículo sobre precio/rendimiento. El libro de Phister [1979] sobre costes de computadores es exhaustivo, y Bell, Mudge y McNamara [1978] describen el proceso de construcción de computadores desde la perspectiva de DEC. En contraste, hay una gran cantidad de información sobre productividad *(yield)* de los dados. Strapper [1989] da una visión general de la historia de la modelación de dados, aunque los detalles técnicos sobre el modelo de productividad de dados utilizado en este capítulo se encuentran en Strapper, Armstrong y Saji [1983].

Referencias

BELL, C. G., J. C. MUDGE, AND J. E. MCNAMARA [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass.

BERRY, M., D. CHEN, P. KOSS, AND KUCK [1988]. «The Perfect Club benchmarks: Effective performance evaluation of supercomputers», CSRD Report No. 827 (November), Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

CONTI, C. J., D. H. GIBSON, AND S. H. PITKOWSLY [1968]. «Structural aspects of the System/360 Model 85:I general organization», *IBM Systems J.* 7:1, 2-11.

CURNOW, H. J., AND B. A. WICHMANN [1976]. «A synthetic benchmark», *The Computer J.* 19:1.

FLEMMING, P. J., AND J. J. WALLACE [1986]. «How not to lie with statistics: The correct way to summarize benchmarks results», *Comm. ACM* 29:3 (March) 218-221.

FULLER, S. H. [1976]. «Price/performance comparison of C.mmp and the PDP-11», *Proc. Third Annual Symposium on Computer Architecture* (Texas, January 19-21), 197-202.

FULLER, S. H., AND W. E. BURR [1977]. «Measurement and evaluation of alternative computer architectures», *Computer* 10:10 (October) 24-35.

GIBSON, J. C. [1970]. «The Gibson mix», Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (Research done in 1959.)

HENLY, M., AND B. MCNUTT [1989]. «DASD I/O characteristics: A comparison of MVS to VM», Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, California.

HOLLINGSWORTH, W., H. SACHS, AND A. J. SMITH [1989]. «The Clipper processor: Instruction set architecture and implementation», *Comm. ACM* 32:2 (February), 200-219.

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. «A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2», *Computer* 18:12 (December) 10-24.

MCMAHON, F. M. [1986]. «The Livermore FORTRAN kernels: A computer test of numerical performance range», Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, California (December).

PHISTER, M. JR. [1979]. *Data Processing Technology and Economics*, 2nd ed., Digital Press and Santa Monica Publishing Company.

SMITH, J. E. [1988]. «Characterizing computer performance with a single number», *Comm. ACM* 31:10 (October) 1202-1206.

SPEC [1989]. «SPEC Benchmark Suite Release 1.0», October 2, 1989.

STRAPPER, C. H. [1989]. «Fact and fiction in yield modelling», Special Issue of the *Microelectronics Journal* entitled *Microelectronics into the Nineties*, Oxford, UK; Elsevier (May).

STRAPPER, C. H., F. H. ARMSTRONG, AND K. SAJI [1983]. «Integrated circuit yield statistics», *Proc. IEEE* 71:4 (April) 453-470.

WEICKER, R. P. [1984]. «Dhrystone: A synthetic systems programming benchmark», *Comm. ACM* 27:10 (October) 1013-1030.

WICHMANN, B. A. [1973]. *Algol 60 Compilation and Assessment*, Academic Press, New York.

EJERCICIOS

2.1 [20] <2.2> Una vez graduado, el lector se preguntará cómo llegar a ser un líder en el diseño de computadores. Su estudio sobre la utilización de construcciones de los lenguajes de alto nivel sugiere que las llamadas a los procedimientos son una de las operaciones más caras. Suponga que ha inventado un esquema que reduce las operaciones de carga y almacenamiento normalmente asociadas con las llamadas y vueltas de procedimientos. Lo primero que hace es ejecutar algunos experimentos con y sin esta optimización. Sus experimentos utilizan el mismo compilador optimizador en ambas versiones del computador.

Los experimentos realizados revelan lo siguiente:

- La duración del ciclo de reloj de la versión no optimizada es el 5 por 100 más rápido.
- El 30 por 100 de las instrucciones de la versión no optimizada son operaciones de carga o almacenamiento.
- La versión optimizada ejecuta 1/3 menos de operaciones de carga y almacenamiento que la versión no optimizada. Para las demás instrucciones, el recuento de ejecución dinámica son inalterables.
- Todas las instrucciones (incluyendo las de carga y almacenamiento) emplean un ciclo de reloj.

¿Qué es más rápido? Justificar cuantitativamente la decisión.

2.2 [15/15/10] <2.2> Supongamos que cada uno de los dos programas de la Figura 2.6 ejecuta 100 000 000 operaciones de punto flotante.

a) [15] Calcular la estimación en MFLOPS (nativos) de cada programa.

b) [15] Calcular las medias aritmética, geométrica, y armónica en MFLOPS nativos para cada máquina.

c) [10] ¿Cuál de las tres medias coincide con el rendimiento relativo del tiempo total de ejecución?

Las Cuestiones 2.3-2.7 requieren la siguiente información:

El benchmark Whetstone contiene 79 550 operaciones en punto flotante, sin incluir las operaciones en punto flotante realizadas en cada llamada a las siguientes funciones:

- arco tangente, invocado 640 veces
- seno, invocado 640 veces
- coseno, invocado 1920 veces
- raíz cuadrada, invocada 930 veces
- exponencial, invocada 930 veces
- y logaritmo, invocado 930 veces

Las operaciones básicas para una iteración (sin incluir operaciones en punto flotante para realizar las funciones anteriores) se desomponen como sigue:

Suma	37 530
Resta	3 520
Multiplicación	22 900
División	11 400
Conversión de entero a punto flotante	4 200
TOTAL	79 550

El número total de operaciones en punto flotante para una iteración también puede calcularse incluyendo las operaciones en punto flotante necesarias para realizar las funciones arco tangente, seno, coseno, raíz cuadrada, exponencial, y logaritmo:

Suma	82 014
Resta	8 229
Multiplicación	73 220
División	21 399
Conversión de entero a punto flotante	6 006
Comparación	4 710
TOTAL	195 578

Wheststone se ejecutó en una Sun 3/75 utilizando el compilador F77 con la optimización activada. La Sun 3/75 está basada en un Motorola 68020 que corre a 16,67 MHz, e incluye un coprocesador de punto flotante. (Suponer que el coprocesador no incluye arco tangente, seno, coseno, raíz cuadrada, exponencial y logaritmo como instrucciones.) El compilador Sun permite que el punto flotante se calcule con el coprocesador o utilizando rutinas software, dependiendo de los señalizadores del compilador. Una iteración de Whetstone necesita 1,08 segundos utilizando el coprocesador y 13,6 segundos utilizando el software. Suponer que al medir el CPI utilizando el coprocesador se obtuvo una medida de 10 mientras que utilizando el software la medida fue de 6.

2.3 [15] <2.2> ¿Cuál es la estimación en MIPS (nativos) para ambas ejecuciones?

2.4 [15] <2.2> ¿Cuál es el número **total** de instrucciones ejecutadas en ambas ejecuciones?

2.5 [8] <2.2> En promedio, ¿cuántas instrucciones enteras se necesitan para realizar cada operación de punto flotante en software?

2.6 [18] <2.2> ¿Cuáles son los MFLOPS nativos y normalizados para la Sun 3/75 cuando se ejecuta Whetstone con el coprocesador de punto flotante? (Suponer que la conversión cuenta como una única operación de punto flotante y utilizar la Figura 2.3 para las operaciones normalizadas.)

2.7 [20] <2.2> La Figura 2.3 sugiere el número de operaciones en punto flotante que pueden realizar las seis funciones anteriores (arco tangente, seno, etc.). A partir de los datos anteriores se puede calcular el número medio de operaciones en punto flotante por función. ¿Cuál es la relación entre las estimaciones de la Figura 2.3 y las medidas de operaciones en punto flotante para la Sun 3? Suponer que el coprocesador implementa solamente Suma, Resta, Multiplicación, División y Conversión *(Convert)*.

Las Cuestiones 2.8-2.10 requieren la información de la Figura 2.28.

El resumen de la versión 1.0 del Benchmark SPEC [SPEC 89] da los rendimientos que se indican en la Figura 2.28.

2.8 [12/15] <2.2> Comparar el rendimiento relativo obtenido utilizando los tiempos totales de ejecución para 10 programas con el obtenido al utilizar las medias geométricas de la relación de velocidad con el VAX-11/780.

a) [12] ¿Cómo difieren los resultados?

Nombre de programa	VAX-11/780	DECstation 3100		Delta Series 8608		SPARCstation 1	
	Tiempo	Tiempo	Relación	Tiempo	Relación	Tiempo	Relación
GCC	1482	145	10,2	193	7,7	138,9	10,7
Espresso	2266	194	11,7	197	11,5	254,0	8,9
Spice 2g6	23951	2500	9,6	3350	7,1	2875,5	8,3
DODUC	1863	208	9,0	295	6,3	374,1	5,0
NASA7	20093	1646	12,2	3187	6,3	2308,2	8,7
Li	6206	480	12,9	458	13,6	689,5	9,0
Eqntott	1101	99	11,1	129	8,5	113,5	9,7
Matrix300	4525	749	6,0	520	8,7	409,3	11,1
FPPP	3038	292	10,4	488	6,2	387,2	7,8
TOMCATV	2649	260	10.2	509	5,2	469,8	5,6
Media geométrica	3867,7	381,4	10,1	496,5	7,8	468,5	8,3

FIGURA 2.28 Resumen del rendimiento de SPEC 1.0. Los cuatro programas enteros son GCC, Espresso, Li, y Eqntott, los restantes cuentan con hardware de punto flotante. El informe de SPEC no describe la versión de los compiladores ni sistema operativo utilizados para el VAX-11/780. La DECstation 3100 se describe en la Figura 2.18. La Serie 8608 Motorola Delta utiliza un MC88100 de 20 MHz, cache de instrucciones de 16 Kb, y cache de datos de 16 Kb utilizando dos M88200 (ver Ejercicio 8.6 del Cap. 8), el sistema operativo Morotola Sys. V/88 R32V1, el compilador C 1.8.4m14 del C88000, y el compilador FORTRAN SysV88 2.0a4 de Absoft. La SPARCstation 1 utiliza una unidad entera MB8909 de 20 MHz y una unidad de punto flotante WTL3170 de 20 MHz, una cache unificada de 64 Kb, sistema operativo SunOS 4.0.3c y compilador C, y compilador FORTRAN de Sun 1.2. El tamaño de la memoria principal de estas tres máquinas es de 16 MB.

b) [15] Comparar la media geométrica de las relaciones de los cuatro programas enteros (GCC, Espresso, Li, y Eqntott) frente al tiempo total de ejecución para estos cuatro programas. ¿Cómo difieren los resultados entre sí y de los resúmenes de los diez programas?

2.9 [15/20/12/10] <2.2> Ahora comparamos rendimientos utilizando medias aritméticas ponderadas.

a) [15] Calcular los pesos para una carga de trabajo para que los tiempos de ejecución en el VAX-11/780 sean iguales para cada uno de los diez programas (ver Fig. 2.7).

b) [20] Utilizando esos pesos, calcular las medias aritméticas ponderadas de los tiempos de ejecución de los diez programas.

c) [12] Calcular la relación de las medias ponderadas de los tiempos de ejecución del VAX a las medias ponderadas para las otras máquinas.

d) [10] ¿En qué se diferencian las medias geométricas de los cocientes y los cocientes de las medias aritméticas ponderadas de los tiempos de ejecución, al resumir el rendimiento relativo?

2.10 [Discusión] <2.2> ¿Qué es una interpretación de las medias geométricas de los tiempos de ejecución? ¿Qué piensa sobre las ventajas y desventajas de utilizar los tiempos totales de ejecución frente a las medias ponderadas de los tiempos de ejecución utilizando igual tiempo de ejecución en el VAX-11/780 frente a las medias geométricas de los cocientes de velocidad respecto al VAX-11/780?

Las Cuestiones 2.11-2.12 requieren la información de la Figura 2.29.

2.11 [15] <2.3> Escoger los microprocesadores mayor y más pequeño de la Figura 2.29, y utilizar los valores de la Figura 2.11 como parámetros de productividad *(yield)*. ¿Cuántos chips buenos se obtienen por oblea?

2.12 [15/10/10/15/15] <2.3> Calcular los costes y precios de los microprocesadores mayor y menor de la Figura 2.29. Usar las hipótesis sobre fabricación de la Figura 2.11 a menos que, específicamente, se mencione otra cosa.

Microprocesador	Tamaño (cm)	Patillas	Encapsulado	Frecuencia de reloj	Precio de lista ($)	Año disponible
Cypress CY7C601	0,8 × 0,7	207	Cerámico PGA	33 MHz	500	1988
Intel 80486	1,6 × 1,0	168	Cerámico PGA	33 MHz	950	1989
Intel 860	1,2 × 1,2	168	Cerámico PGA	33 MHz	750	1989
MIPS R3000	0,8 × 0,9	144	Cerámico PGA	25 MHz	300	1988
Motorola 88100	0,9 × 0,9	169	Cerámico PGA	25 MHz	695	1989

FIGURA 2.29 Características de los microprocesadores. Los precios de lista se tomaron el 15 de julio de 1989 para una cantidad de 1 000 compras.

a) [15] Hay grandes diferencias en las densidades de defectos entre los fabricantes de semiconductores. ¿Cuáles son los costes de los dados no probados suponiendo: (1) 2 defectos por centímetro cuadrado; y (2) 1 defecto por centímetro cuadrado?

b) [10] Suponer que el test cuesta 150 dólares por hora y el circuito integrado más pequeño necesita 10 segundos para ser probado y el mayor 15 segundos, ¿cuál es el coste del test de cada dado?

c) [10] Haciendo las hipótesis de encapsulamiento de la Sección 2.3, ¿cuales son los costes de encapsulamiento y quemado?

d) [15] ¿Cuál es el coste final?

e) [15] Dada la lista de precios y el coste calculado en las preguntas anteriores, calcular el margen bruto. Suponer que el coste directo es el 40 por 100 y el descuento de venta medio es el 33 por 100. ¿Qué porcentaje del precio de venta medio es el margen bruto para ambos circuitos integrados?

2.13-2.14 Algunas compañías afirman que lo hacen tan bien que la densidad de defectos está dejando de ser la razón de los fallos de los dados, haciendo a la productividad de la oblea responsable de la mayoría de los fallos. Por ejemplo, Gordon Moore de Intel dijo en una conferencia en el MIT, en 1989, que la densidad de defectos está mejorando hasta el punto de que algunas compañías han obtenido un rendimiento del 100 por 100 sobre la tirada completa. En efecto, él tiene una oblea con un rendimiento del 100 por 100 en su despacho.

2.13 [20] <2.3> Para comprender el impacto de tales afirmaciones, indicar los costes de los dados mayor y más pequeño de la Figura 2.29 para densidades por defecto por centímetro cuadrado de 3, 2, 1, y 0. Para los demás parámetros utilizar los valores de la Figura 2.11. Ignorar los costes de los tiempos de test, encapsulamiento y test final.

2.14 [Discusión] <2.3> Si la sentencia anterior fuese cierta para la mayoría de los fabricantes de semiconductores, ¿cómo cambiarían las opciones para el diseñador de computadores?

2.15 [10/15] <2.3,2.4> La Figura 2.18 muestra el precio de lista de la estación de trabajo DECstation 3100. Comenzar con los costes del modelo «coste más elevado» de la Figura 2.13, (suponer de color), pero cambiar el coste de DRAM a 100 dólares/MB para los 16 MB completos de la 3100.

a) [10] Utilizando el descuento medio y los porcentajes de gastos del Modelo B de la Figura 2.16, ¿cuál es el margen bruto en la DECstation 3100?

b) [15] Suponer que se sustituye la CPU R2000 de la DECstation 3100 por la R3000, y que este cambio hace a la máquina un 50 por 100 más rápida. Utilizar los costes de la Figura 2.29 para la R3000, y suponer que el coste de la R2000 es de la tercera parte. Como la R3000 no requiere mucha más potencia, suponer que tanto la fuente de alimentación como el sistema de refrigeración de la DECstation 3100 sigue siendo satisfactorio. ¿Cuál es la relación coste/rendimiento de una estación de trabajo (monocolor) sin disco con una R2000 frente a otra con una R3000? Utilizando el modelo comercial de la respuesta de la parte a, ¿en cuánto hay que incrementar el precio de la máquina basada en la R3000?

2.16 [30] <2.2,2.4> Escoger dos computadores y correr el benchmark Dhrystone y el compilador Gnu C. Intentar ejecutar los programas sin optimización y utilizando la

máxima optimización. (Nota. GCC es un benchmark, usar el compilador C apropiado para compilar ambos programas. ¡No tratar de compilar GCC y de utilizarlo como compilador!) Después calcular las siguientes relaciones de rendimiento:

1. Dhrystone no optimizado en la máquina A frente a Dhrystone no optimizado en la máquina B
2. GCC no optimizado en A frente a GCC no optimizado en B
3. Dhrystone optimizado en A frente a Dhrystone optimizado en B
4. GCC optimizado en A frente a GCC optimizado en B
5. Dhrystone no optimizado frente a Dhrystone optimizado en la máquina A
6. GCC no optimizado frente a GCC optimizado en A
7. Dhrystone no optimizado frente a Dhrystone optimizado en B
8. GCC no optimizado frente a GCC optimizado en B

La pregunta sobre los benchmarks es cómo predicen el rendimiento de los programas reales.

Si los benchmarks predicen el rendimiento, entonces las siguientes ecuaciones deben ser ciertas con respecto a las relaciones:

(1) = (2) y (3) = (4)

Si las optimizaciones del compilador funcionan igual de bien en los programas reales que en los benchmarks, entonces

(5) = (6) y (7) = (8)

¿Son estas ecuaciones ciertas? Si no es así, intentar encontrar la explicación. ¿Son las máquinas, las optimizaciones del compilador, o los programas quienes explican la respuesta?

2.17 [30] <2.2,2.4> Realizar el mismo experimento que en la cuestión 2.16, pero sustituyendo Dhrystone por Whetstone y GCC por Spice.

2.18 [Discusión] <2.2> ¿Cuáles son los pros y contras de los benchmarks sintéticos? Encontrar una evidencia cuantitativa —como por ejemplo los datos suministrados para responder a las cuestiones 2.16 y 2.17— e indicar las ventajas y desventajas cualitativas.

2.19 [30] <2.2,2.4> Realizar un programa en C o Pascal que obtenga la estimación de MIPS máxima para un computador. Ejecutarlo en dos máquinas para calcular los MIPS máximos. Ahora ejecutar GCC y TeX en ambas máquinas. ¿Cómo predicen los MIPS máximos el rendimiento de GCC y TeX?

2.20 [30] <2.2,2.4> Redactar un programa en C o en FORTRAN que obtenga la estimación máxima de MFLOPS para un computador. Ejecutarlo en dos máquinas para calcular los MFLOPS máximos. Correr ahora Spice en ambas máquinas. ¿Cómo predicen los MFLOPS máximos el rendimiento de Spice?

2.21 [Discusión] <2.3> Utilizar la información sobre costes de la Sección 2.3 como base para las ventajas de un gran computador a tiempo compartido frente a una red de estaciones de trabajo. (Para determinar el valor potencial de las estaciones de trabajo frente al tiempo compartido, ver la Sección 9.2 del Capítulo 9 sobre productividad del usuario.)

A n *Sumar el número de la posición de memoria n al acumulador.*

H n *Transferir el número de la posición de memoria n al registro multiplicador.*

E n *Si el número del acumulador es mayor o igual que cero, ejecutar la siguiente orden que está en la posición de memoria n; en cualquier otro caso proceder en serie.*

I n *Leer la fila siguiente de agujeros de la cinta y poner los 5 dígitos resultantes en los lugares menos significativos de la posición de memoria n.*

Z *Detener la máquina y hacer sonar la campana de aviso.*

Selección de la lista de las 18 instrucciones máquina de la EDSAC de Wilkes y Renwick (1949)

3.1 Introducción

3.2 Clasificación de las arquitecturas a nivel lenguaje máquina

3.3 Almacenamiento de operandos en memoria: clasificación de las máquinas de registros de propósito general

3.4 Direccionamiento de memoria

3.5 Operaciones del repertorio de instrucciones

3.6 Tipo y tamaño de los operandos

3.7 El papel de los lenguajes de alto nivel y compiladores

3.8 Juntando todo: cómo los programas utilizan los repertorios de instrucciones

3.9 Falacias y pifias

3.10 Observaciones finales

3.11 Perspectiva histórica y referencias

Ejercicios

3 Diseño de repertorios de instrucciones: Alternativas y principios

3.1 Introducción

En este capítulo y en el siguiente nos centraremos en las arquitecturas a nivel lenguaje máquina —la parte de la máquina visible al programador o escritor de compiladores. Este capítulo introduce la gran cantidad de alternativas de diseño con las que se encuentran los arquitectos de los repertorios de instrucciones. En particular, este capítulo hace énfasis en tres aspectos. Primero, presentamos una taxonomía de alternativas de repertorios de instrucciones y hacemos una valoración cualitativa de las ventajas y desventajas de los diversos enfoques. Segundo, presentamos y analizamos algunas medidas de repertorios de instrucciones que son suficientemente independientes de un repertorio específico. Finalmente, tratamos el aspecto de los lenguajes y compiladores y su influencia sobre la arquitectura a nivel lenguaje máquina. Antes de explicar cómo clasificar arquitecturas, necesitamos decir algo sobre medidas de los repertorios de instrucciones.

A lo largo de este capítulo y del siguiente, examinaremos una amplia variedad de medidas arquitectónicas. Estas medidas dependen de los programas medidos y de los compiladores utilizados al hacer las medidas. Los resultados no deben interpretarse de forma absoluta, ya que se pueden ver datos diferentes si se hacen medidas con compiladores diferentes o con un conjunto diferente de programas. Los autores piensan que las medidas mostradas en estos capítulos son razonablemente indicativas de una clase de aplicaciones típicas. Las medidas se presentan utilizando un pequeño conjunto de «benchmarks» para que los datos se puedan visualizar razonablemente, y así, puedan verse las diferencias entre programas. A un arquitecto, para una nueva máquina, le

gustaría analizar una colección **mucho mayor** de programas para tomar sus decisiones arquitectónicas. Todas las medidas mostradas son *dinámicas* —es decir, la frecuencia de un evento medido está ponderada por el número de veces que ocurre el evento durante la ejecución del programa medido.

Ahora, comenzaremos a explorar cómo se pueden clasificar y analizar las arquitecturas a nivel lenguaje máquina.

3.2 Clasificación de las arquitecturas a nivel lenguaje máquina

Los repertorios de instrucciones se pueden clasificar en general siguiendo las cinco dimensiones descritas en la Figura 3.1, que están ordenadas aproximadamente por el papel que juegan en diferenciar los repertorios de instrucciones.

Almacenamiento de operandos en la CPU	Además de en memoria ¿dónde se encuentran los operandos?
Número de operandos explícitos por instrucción	¿Cuántos operandos son designados explícitamente en una instrucción típica?
Posición del operando	¿Puede cualquier operando de una instrucción de la ALU estar localizado en memoria o deben algunos o todos los operandos estar en la memoria interna de la CPU? Si un operando está localizado en memoria, ¿cómo se especifica la posición de memoria?
Operaciones	¿Qué operaciones se proporcionan en el repertorio de instrucciones?
Tipo y tamaño de operandos	¿Cuál es el tipo y tamaño de cada operando y cómo se especifica?

FIGURA 3.1 Un conjunto de ejes para elecciones de alternativas de diseño de repertorios de instrucciones. El tipo de almacenamiento proporcionado para que los operandos estén en la CPU, en contraposición a que estén en memoria, es el factor distintivo más importante entre las arquitecturas a nivel lenguaje máquina. (Todas las arquitecturas conocidas por los autores tienen algún almacenamiento temporal en la CPU.) El tipo de almacenamiento de operandos de la CPU a veces dicta el número de operandos explícitamente nombrados en una instrucción. En una clase de máquinas, el número de operandos explícitos puede variar. Entre los repertorios de instrucciones recientes, el número de operandos de memoria por instrucción es otro factor diferencial significativo. La elección de las operaciones que soportarán las instrucciones tiene poca interacción con otros aspectos de la arquitectura. Finalmente, especificar el tipo de datos y el tamaño de un operando es enormemente independiente de otras elecciones de los repertorios de instrucciones.

Almacenamiento temporal proporcionado	Ejemplos	Operandos explícitos por instrucción ALU	Destino para resultados	Procedimiento para acceder a operandos explícitos
Pila	B5500, HP 3000/70	0	Pila	Introducir y sacar de la pila
Acumulador	PDP-8 Motorola 6809	1	Acumulador	Cargar/ almacenar acumulador
Conjunto de registros	IBM 360, DEC VAX	2 ó 3	Registros o memoria	Cargar/ almacenar registros o memoria

FIGURA 3.2 Algunas alternativas para almacenar operandos en la CPU. Cada alternativa implica que se necesita un número diferente de operandos explícito para una instrucción de dos operandos fuente y un operando resultado. Los repertorios de instrucciones se clasifican habitualmente por este estado interno como máquina de pila, máquina de acumulador o máquina de registros de propósito general. Aunque muchas arquitecturas están incluidas claramente en una clase u otra, algunas son híbridos de los diferentes enfoques. Por ejemplo, el Intel 8086 está a mitad de camino entre una máquina de registros de propósito general y una máquina de acumulador.

La diferencia básica está en el tipo de almacenamiento interno de la CPU; por consiguiente, en esta sección, nos centraremos en las alternativas para esta parte de la arquitectura. Como muestra la Figura 3.2, las elecciones principales son una pila, un acumulador o un conjunto de registros. Los operandos pueden ser nombrados explícita o implícitamente: Los operandos en una *arquitectura de pila* están implícitamente en el tope de la pila; en una *arquitectura de acumulador* un operando está implícitamente en el acumulador. Las *arquitecturas de registros de propósito general* tienen solamente operandos explícitos —en registros o en posiciones de memoria. Dependiendo de la arquitectura, los operandos explícitos para una operación pueden ser accedidos directamente desde memoria o puede ser necesario cargarlos primero en el almacenamiento temporal, dependiendo de la clase de instrucción y elección de la instrucción específica.

La Figura 3.3 muestra cómo sería la secuencia de código C = A + B en estas tres clases de repertorios de instrucciones. Las ventajas y desventajas principales de cada uno de estos enfoques aparecen en la Figura 3.4.

Aunque las máquinas más primitivas utilizaban arquitecturas estilo pila o acumulador, las máquinas diseñadas en los últimos diez años y que sobreviven todavía, utilizan una arquitectura de registros de propósito general. Las razones principales para la aparición de las máquinas de registros de propósito general son dobles. Primero, los registros —como otras formas de almacenamiento interno para la CPU— son más rápidos que la memoria. Segundo, los registros son más fáciles de utilizar por un compilador y se pueden usar de manera más efectiva que otras formas de almacenamiento interno. Debido a

Pila	Acumulador	Registro
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE C, R1
POP C		

FIGURA 3.3 Secuencia de código de C = A + B para tres repertorios de instrucciones diferentes. Se supone que A, B y C están en memoria y que los valores de A y B no pueden ser destruidos.

Tipo de máquina	Ventajas	Desventajas
Pila	Modelo sencillo para evaluación de expresiones (polaca inversa). Instrucciones cortas pueden dar una buena densidad de código.	A una pila no se puede acceder aleatoriamente. Esta limitación hace difícil generar código eficiente. También dificulta una implementación eficiente, ya que la pila llega a ser un cuello de botella.
Acumulador	Minimiza los estados internos de la máquina. Instrucciones cortas.	Como el acumulador es solamente almacenamiento temporal, el tráfico de memoria es el más alto en esta aproximación.
Registro	Modelo más general para generación de código.	Todos los operandos deben ser nombrados, conduciendo a instrucciones más largas.

FIGURA 3.4 Principales ventajas y desventajas de cada clase de máquina. Estas ventajas y desventajas están relacionadas con tres aspectos: la forma en que la estructura se adapta a las necesidades de un compilador; lo eficiente que es el enfoque desde el punto de vista de la implementación, y cómo es el tamaño efectivo de código con respecto a otros enfoques.

que las máquinas de registros de propósito general dominan hoy día las arquitecturas a nivel lenguaje máquina —y parece improbable que cambie en el futuro— consideraremos solamente estas arquitecturas a partir de este punto. Incluso con esta limitación, todavía hay un gran número de alternativas de diseño a tener en cuenta. Algunos diseñadores han propuesto la extensión del concepto de registro, para lograr una memoria intermedia adicional de múltiples conjuntos de registros, de forma análoga a una pila. Este nivel adicional de jerarquía de memoria se examina en el Capítulo 8.

3.3 Almacenamiento de operandos en memoria: clasificación de las máquinas de registros de propósito general

Las ventajas claves de las máquinas de registros de propósito general surgen del uso efectivo de los registros por el compilador, al calcular valores de ex-

presiones y, más globalmente, al utilizar los registros para que contengan variables. Los registros permiten una ordenación más flexible que las pilas o acumuladores, a la hora de evaluar las expresiones. Por ejemplo, en una máquina de registros, la expresión (A · B) – (C · D) – (E · F) puede evaluarse haciendo las multiplicaciones en cualquier orden, lo que puede ser más eficiente debido a la posición de los operandos o a causa de las posibilidades de la segmentación (ver Cap. 6). Pero en una máquina de pilas la expresión debe evaluarse de izquierda a derecha, a menos que se realicen operaciones especiales o intercambios de las posiciones de la pila.

Más importante, los registros pueden utilizarse para que contengan variables. Cuando las variables están ubicadas en registros, se reduce el tráfico de memoria, se acelera el programa (ya que los registros son más rápidos que la memoria), y mejora la densidad de código (ya que un registro se puede nombrar con menos bits que una posición de memoria). Los escritores de compiladores prefieren que todos los registros sean equivalentes y no reservados. Muchas máquinas comprometen este deseo —especialmente las máquinas antiguas con muchos registros dedicados, disminuyendo de manera efectiva el número de registros de propósito general. Si este número realmente es muy pequeño, tratar de ubicar las variables en los registros no será rentable. En lugar de ello, el compilador reservará todos los registros sin cometido para utilizarlos en la evaluación de expresiones.

¿Cuántos registros son suficientes? La respuesta, por supuesto, depende de cómo los utilice el compilador. La mayoría de los compiladores reservan algunos registros para la evaluación de expresiones, utilizan otros para paso de parámetros, y dejan los restantes para ubicar variables. Para comprender cuántos registros son suficientes, realmente necesitamos examinar las variables que pueden ubicarse en los registros y el algoritmo de ubicación utilizado. Trataremos esto en nuestra discusión sobre compiladores, en la Sección 3.7 y, posteriormente, examinaremos las medidas de utilización de registros.

Hay dos características importantes de los repertorios de instrucciones que dividen las arquitecturas de registros de propósito general o *GPR*. Ambas características están relacionadas con la naturaleza de los operandos para una instrucción lógica o aritmética, o instrucción de la ALU. La primera se refiere al número de operandos (dos o tres) que pueden tener las instrucciones de la ALU. En el formato de tres operandos, la instrucción contiene un resultado y dos operandos fuente. En el formato de dos operandos, uno de los operandos es fuente y destino para la operación. La segunda característica de las arquitecturas GPR está relacionada con el número de operandos que se pueden direccionar en memoria en las instrucciones de la ALU. Este número puede variar de cero a tres. Todas las combinaciones posibles de estos dos atributos se muestran en la Figura 3.5, con ejemplos de máquinas. Aunque hay siete posibles combinaciones, tres sirven para clasificar aproximadamente todas las máquinas existentes: *registro-registro* (también llamadas *de carga/almacenamiento), registro-memoria* y *memoria-memoria.*

Las ventajas y desventajas de cada una de estas alternativas se muestran en la Figura 3.6. Por supuesto, éstas no son absolutas. Una máquina GPR con operaciones memoria-memoria puede ser transformada fácilmente por el compilador y utilizada como una máquina registro-registro. Las ventajas y desventajas que aparecen en la figura se refieren principalmente al impacto

Número de direcciones de memoria por instrucción típica de la ALU	Máximo número de operandos permitidos por instrucción típica de la ALU	Ejemplos
0	2 3	IBM RT-PC SPARC, MIPS, HP Precision Architecture
1	2 3	PDP-10, Motorola 68000, IBM 360 Parte de IBM 360 (instrucciones RS)
2	2 3	PDP-11, National 32×32, parte de IBM 360 (instrucciones SS)
3	3	VAX (también tiene formatos de dos operandos)

FIGURA 3.5 Posibles combinaciones de operandos de memoria y operandos totales por instrucción ALU con ejemplos de máquinas. Las máquinas sin referencia a memoria por instrucción ALU son denominadas máquinas de carga/almacenamiento o máquinas registro-registro. Las instrucciones con múltiples operandos de memoria por instrucción ALU típica se denominan memoria-registro o memoria-memoria, según tengan uno o más de un operando de memoria.

Tipo	Ventajas	Desventajas
Registro-registro (0,3)	Codificación simple de instrucciones de longitud fija. Modelo simple de generación de código. Las instrucciones emplean números de ciclos similares para ejecutarse (ver Capítulo 6).	Recuento más alto de instrucciones que las arquitecturas con referencias a memoria en las instrucciones. Algunas instrucciones son cortas y la codificación de bits puede ser excesiva.
Registro-memoria (1,2)	Los datos pueden ser accedidos sin cargarlos primero. El formato de instrucción tiende a ser fácil para codificar y obtener buena densidad.	Los operandos no son equivalentes, ya que en una operación binaria se destruye un operando fuente. Codificar un número de registro y una dirección de memoria en cada instrucción puede restringir el número de registros. Los ciclos por instrucción varían por la posición de operando.
Memoria-memoria (3,3)	Más compacta. No emplean registros para temporales.	Gran variación en el tamaño de las instrucciones, especialmente en instrucciones de tres operandos. Además gran variación en el trabajo por instrucción. Los accesos a memoria crean cuellos de botella en memoria.

FIGURA 3.6 Ventajas y desventajas de los tres tipos más comunes de máquinas de registros de propósito general. La notación *(m, n)* significa *m* operandos de memoria y *n* operandos totales. En general, las máquinas con menos alternativas hacen la tarea del compilador más simple ya que el compilador tiene que tomar menos decisiones. Las máquinas con gran variedad de formatos de instrucción flexibles reducen el número de bits necesarios para codificar el programa. Una máquina que use un pequeño número de bits para codificar el programa se dice que tiene una buena *densidad de instrucciones* —un número más pequeño de bits hace el mismo trabajo que un número mayor en una arquitectura diferente—. El número de registros también afecta al tamaño de la instrucción.

sobre el compilador y sobre la implementación. Estas ventajas y desventajas son cualitativas y su impacto real depende de la estrategia de implementación y del compilador. Uno de los impactos más penetrantes está en la codificación de las instrucciones y en el número de instrucciones necesarias para realizar una tarea. En otros capítulos, veremos el impacto de estas alternativas arquitectónicas en distintas aproximaciones de implementaciones.

3.4 Direccionamiento de memoria

Independientemente que una arquitectura sea registro-registro (también llamada de *carga/almacenamiento)* o permita que cualquier operando sea una referencia a memoria, debe definir cuántas direcciones de memoria son interpretadas y cómo se especifican. En esta sección trataremos estas dos cuestiones. Las medidas presentadas aquí son muy, pero no completamente, independientes de la máquina. En algunos casos, las medidas están afectadas significativamente por la tecnología del compilador. Estas medidas se han realizado utilizando un compilador optimizador, ya que la tecnología de compiladores está jugando un papel creciente. Las medidas probablemente reflejarán lo que veremos en el futuro antes que lo que ha sido el pasado.

Interpretación de las direcciones de memoria

¿Cómo se interpreta una dirección de memoria? Es decir, ¿qué objeto es accedido como una función de la dirección y la longitud? Todas las máquinas explicadas en este capítulo y en el siguiente están direccionadas por bytes y proporcionan accesos a bytes (8 bits), medias palabras (16 bits) y palabras (32 bits). La mayoría de las máquinas también proporcionan accesos a dobles palabras (64 bits).

Como muestra la Figura 3.7, hay dos convenios diferentes para clasificar los bytes de una palabra. El orden de bytes *«Little Endian» (Pequeño Endian)* coloca el byte cuya dirección es «x...x00» en la posición menos significativa de la palabra (little end —extremo pequeño—). El orden de bytes *«Big Endian» (Gran Endian)* coloca el byte cuya dirección es «x...x00» en la posición más significativa de la palabra (big end —extremo grande—). En el direccionamiento «Big Endian», la dirección de un dato es la dirección del byte más significativo; mientras que en el «Little Endian», es la del byte menos significativo. Cuando se opera con una máquina, el orden de los bytes, con frecuencia, no es importante —solamente los programas que acceden a las mismas posiciones como palabras y bytes pueden observar la diferencia—. Sin embargo, el orden de los bytes es un problema cuando se intercambian datos entre máquinas con diferentes ordenaciones. (La ordenación de bytes utilizada por una serie de máquinas diferentes se indican en la cubierta anterior.)

En algunas máquinas, los accesos a los objetos mayores de un byte deben estar alineados. Un acceso a un objeto de tamaño s bytes en el byte de dirección A se alinea si A **mod** $s = 0$. La Figura 3.8 muestra las direcciones en las cuales un acceso está alineado o desalineado.

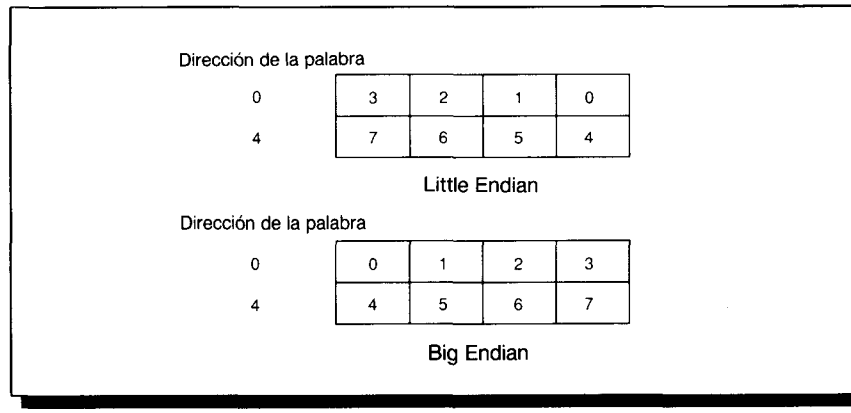


FIGURA 3.7 Los dos diferentes convenios para ordenar bytes en una palabra. Los nombres «Big Endian» y «Little Endian» provienen de un famoso artículo de Cohen [1981]. El artículo establece una analogía entre la discusión sobre por qué extremo del byte comenzar y la discusión de los Viajes de Gulliver sobre qué extremo del huevo abrir. El DEC PDP-11/VAX y los Intel 80x86 siguen el modelo «Little Endian», mientras que el IBM 360/370 y los Motorola 680x0, y otras siguen el modelo «Big Endian». Esta numeración se aplica también a las posiciones de los bits, aunque sólo algunas arquitecturas suministren instrucciones para acceder a los bits por su número de posición.

Objeto direccionado	Alineado en desplazamiento del byte	Mal alineado en desplazamiento del byte
byte	0,1,2,3,4,5,6,7	(nunca)
media palabra	0,2,4,6	1,3,5,7
palabra	0,4	1,2,3,5,6,7
doble palabra	0	1,2,3,4,5,6,7

FIGURA 3.8 Accesos de objetos alineados y mal alineados. Los desplazamientos de los bytes los especifican los tres bits de orden inferior de la dirección.

¿Por qué se diseña una máquina con restricciones de alineación? La no alineación causa complicaciones hardware, ya que la memoria, normalmente, está alineada sobre una frontera de palabras. Un acceso no alineado en memoria, por tanto, tendrá múltiples referencias a una memoria alineada. La Figura 3.9 muestra qué ocurre cuando se realiza un acceso a una palabra no alineada en un sistema con un bus de 32 bits a memoria: se requieren dos accesos para obtener la palabra. Por tanto, incluso en máquinas que permiten accesos no alineados, los programas con accesos alineados se ejecutan más rápidamente.

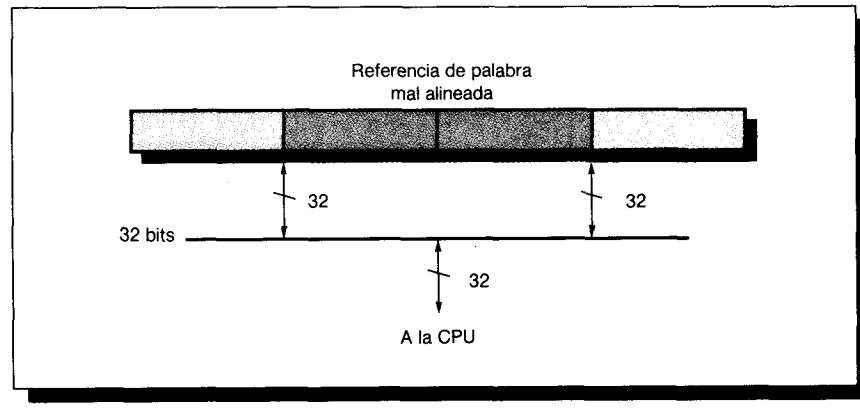


FIGURA 3.9 Una referencia de palabra se hace en un extremo de media palabra (16 bits) en un sistema de memoria que tenga un camino de acceso de 32 bits. La CPU o sistema de memoria tiene que realizar dos accesos separados para obtener la mitad de palabra superior y la inferior. Las dos mitades de la palabra se mezclan entonces para obtener la palabra completa. Con la memoria organizada en módulos de bytes independientes es posible acceder solamente al dato necesario, pero esto requiere un control más complejo para suministrar una dirección diferente a cada módulo para seleccionar el byte adecuado.

Incluso si el dato está alineado, soportar accesos a bytes y medias palabras requiere una red de alineamiento para alinear los bytes y medias palabras en los registros. Dependiendo de la instrucción, la máquina también puede necesitar extender el signo. En algunas máquinas, un byte o media palabra no afecta a la parte superior de un registro. Para los almacenamientos, solamente pueden ser alterados los bytes de memoria afectados. La Figura 3.10 muestra la red de alineamiento para cargar o almacenar un byte desde una palabra de memoria en un registro. Aunque todas las máquinas explicadas en este capítulo y el siguiente permiten accesos a memoria de bytes y medias palabras, solamente el VAX y el Intel 8086 soportan operaciones de la ALU sobre operandos en registros cuyo tamaño es menor de una palabra.

Modos de direccionamiento

Ahora sabemos a qué bytes de memoria acceder, dada una dirección. En esta sección examinaremos los modos de direccionamiento —la forma en que las arquitecturas especifican la dirección de un objeto al que accederán. En las máquinas GPR, un modo de direccionamiento puede especificar una constante, un registro o una posición de memoria. Cuando se utiliza una posición de memoria, la dirección real de memoria especificada por el modo de direccionamiento se denomina *dirección efectiva.*

La Figura 3.11 muestra todos los modos de direccionamiento de datos que se utilizan en las máquinas explicadas en el siguiente capítulo. Los inmediatos o literales se consideran habitualmente un modo de direccionamiento de me-

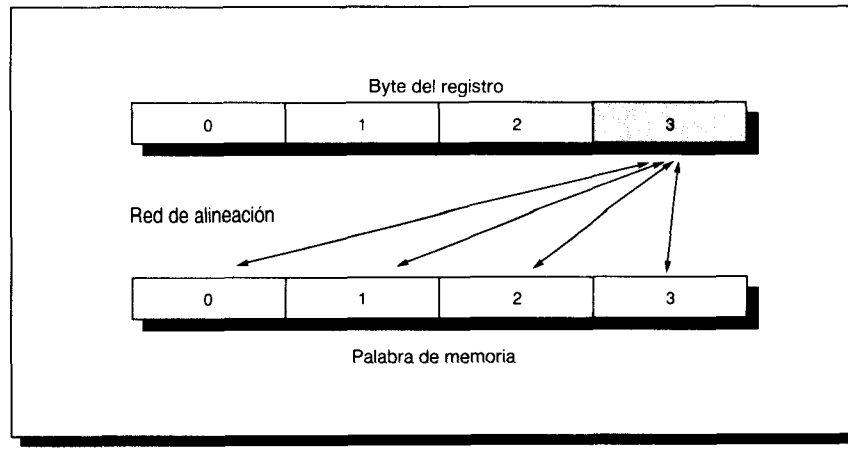


FIGURA 3.10 Red de alineación para cargar o almacenar un byte. El sistema de memoria supone que es de 32 bits, y se necesitan cuatro caminos de alineación para los bytes. Acceder a medias palabras alineadas requeriría dos caminos adicionales para desplazar bien el byto 0 o el byte 2 de memoria al byte 2 del registro. Un sistema de memoria de 64 bits requeriría el doble de caminos de alineación para los bytes y medias palabras, así como dos caminos de alineación de 32 bits para accesos a palabras. La red de alineación sólo posiciona los bytes para almacenarlos —las señales adicionales de control se utilizan para asegurar que sólo las posiciones correctas del byte son escritas en memoria—. En lugar de una red de alineación, algunas máquinas utilizan un desplazador y desplazan el dato sólo en aquellos casos donde se necesita alineación. Esto hace considerablemente más lento el acceso de un objeto no palabra, pero eliminando la red de alineación se acelera el caso más común de acceder a una palabra.

moria (aun cuando el valor que acceden esté en el flujo de instrucciones), aunque los registros están, con frecuencia, separados. Hemos mantenido aparte los modos de direccionamiento que dependen del contador de programa, denominados *direccionamientos relativos al PC*. Estos modos de direccionamiento se utilizan principalmente para especificar direcciones de código en instrucciones de transferencia de control. El uso del direccionamiento relativo al PC en instrucciones de control se explica en la Sección 3.5.

La Figura 3.11 muestra los nombres más comunes de los modos de direccionamiento, aunque difieren entre arquitecturas. En esta figura y a lo largo del libro, utilizaremos una extensión del lenguaje de programación C como notación de descripción hardware. En esta figura, solamente se utilizan dos características no-C. Primero, la flecha izquierda (←) se utiliza para asignaciones. Segundo, el array `M` se utiliza como nombre de la memoria. Por tanto, `M[R1]` referencia el contenido de la posición de memoria cuya dirección está dada por el contenido de R1. Más tarde, introduciremos extensiones para acceder y transferir datos menores de una palabra.

Los modos de direccionamiento tienen la posibilidad de reducir significativamente el recuento de instrucciones; también se añaden a la complejidad de construir una máquina. Por tanto, la utilización de varios modos de direc-

Modo de direcciona-miento	Ejemplo instrucción	Significado	Cuándo se usa
Registro	`Add R4, R3`	`R4←R4+R3`	Cuando un valor está en un registro.
Inmediato o literal	`Add R4,#3`	`R4←R4+3`	Para constantes. En algunas máquinas, literal e inmediato son dos modos diferentes de direccionamiento.
Desplaza-miento	`Add R4, 100(R1)`	`R4←R4+M[100+R1]`	Acceso a variables locales.
Registro diferido o indirecto	`Add R4, (R1)`	`R4←R4+M[R1]`	Acceso utilizando un puntero o una dirección calculada.
Indexado	`Add R3, (R1+R2)`	`R3←R3+M[R1+R2]`	A veces útil en direccionamiento de arrays—`R1`=base del array; `R2`=cantidad de índices
Directo o absoluto	`Add R1, (1001)`	`R1←R1+M[1001]`	A veces útil para acceder a datos estáticos; la constante que especifica la dirección puede necesitar ser grande
Indirecto o diferido de memoria	`Add R1,@(R3)`	`R1←R1+M[M[R3]]`	Si `R3` es la dirección de un puntero *p*, entonces el modo obtiene **p*
Auto-incremento	`Add R1,(R2)+`	`R1←R1+M[R2]` `R2←R2+`*d*	Util para recorridos de arrays en un bucle. `R2` apunta al principio del array; cada referencia incrementa `R2` en el tamaño de un elemento, *d*.
Auto-decremento	`Add R1,-(R2)`	`R2←R2-`*d* `R1←R1+M[R2]`	El mismo uso que autoincremento. Autoincremento/decremento también puede utilizarse para realizar una pila mediante introducir y sacar (push y pop)
Escalado o Indice	`Add R1,100(R2)[R3]`	`R1←` `R1+M[100+R2+R3*`*d*`]`	Usado para acceder arrays por índice. Puede aplicarse a cualquier modo de direccionamiento básico en algunas máquinas

FIGURA 3.11 Selección de modos de direccionamiento con ejemplos, significado y uso. Se definen las extensiones a C utilizadas en las descripciones hardware. En los modos de direccionamiento de autoincremento/decremento y escalado o índice, la variable *d* designa el tamaño del elemento del dato al que se está accediendo (es decir, si la instrucción está accediendo a 1, 2, 4 u 8 bytes); esto significa que estos modos de direccionamiento solamente son útiles cuando los elementos a los que se está accediendo son adyacentes en memoria. En nuestras medidas, utilizamos el primer nombre mostrado para cada modo. Algunas máquinas, como el VAX, codifican algunos de estos modos de direccionamiento como relativos al PC.

cionamiento es bastante importante para ayudar al arquitecto a escoger cual incluir. Aunque muchas medidas de la utilización de los modos de direccionamiento son dependientes de la máquina, otras son prácticamente independientes de la arquitectura de la máquina. En este capítulo, se examinarán algunas de las medidas más importantes, independientes de la máquina. Pero, antes de que examinemos este tipo de medidas, veamos la frecuencia con que se utilizan estos modos de direccionamiento de memoria.

La Figura 3.12 muestra los resultados de medir patrones de utilización de los modos de direccionamiento en nuestros benchmarks —Compilador Gnu C (GCC), Spice y TeX— en el VAX, que soporta todos los modos mostrados en la Figura 3.11. En el siguiente capítulo examinaremos medidas adicionales de utilización de los modos de direccionamiento en el VAX.

Como muestra la Figura 3.12, el direccionamiento inmediato y de desplazamiento dominan la utilización de los modos de direccionamiento. Veamos algunas propiedades de estos dos modos intensamente utilizados.

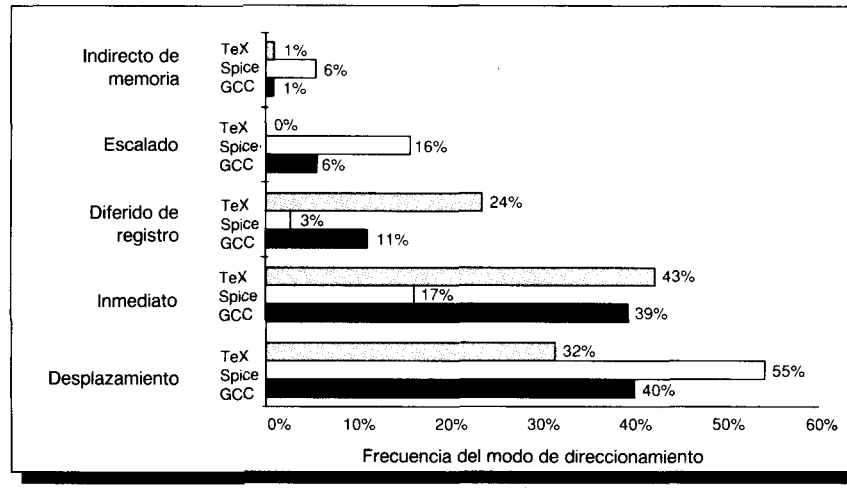


FIGURA 3.12 Resumen del uso de modos de direccionamiento de memoria (incluyendo los inmediatos). Los datos se tomaron en un VAX utilizando nuestros tres programas de benchmark. Sólo se muestran los modos de direccionamiento con una frecuencia media superior al 1 por 100. Los modos de direccionamiento relativos al PC, que se usan casi exclusivamente para los saltos, no se incluyen. El modo de desplazamiento incluye todas las longitudes de desplazamiento (8, 16 y 32 bits). Los modos de registro, que no se cuentan, contabilizan la mitad de las referencias de los operandos, mientras que los modos de direccionamiento de memoria (incluyendo el inmediato) contabilizan la otra mitad. El modo indirecto de memoria en el VAX puede utilizar desplazamiento, autoincremento o autodecremento para formar la dirección inicial de memoria; en estos programas, casi todas las referencias indirectas a memoria utilizan como base el modo de desplazamiento. Por supuesto, el compilador afecta los modos de direccionamiento que se estén usando; explicamos esto con más detalle en la Sección 3.7. Estos modos principales de direccionamiento contabilizan casi todos excepto un pequeño porcentaje (0 al 3 por 100) de los accesos a memoria.

Modo de direccionamiento desplazamiento

La pregunta principal que surge para un modo de direccionamiento de estilo desplazamiento es la del rango de los desplazamientos utilizados. Basándose en el uso de diversos tamaños de desplazamiento, puede tomarse una decisión

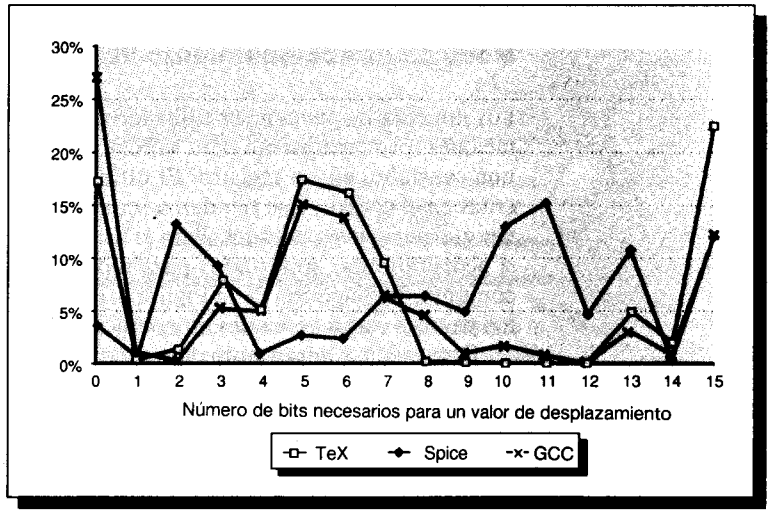


FIGURA 3.13 Los valores de los desplazamientos están ampliamente distribuidos. Aunque hay un gran número de valores pequeños, también hay un número razonable de valores grandes. La amplia distribución de los valores de desplazamiento se debe a múltiples áreas de almacenamiento para variables y diferentes desplazamientos utilizados para accederlas. Las diferentes áreas de almacenamiento y sus patrones de acceso se explican con más detalle en la Sección 3.7. El diagrama muestra solamente la magnitud de los desplazamientos y no los signos, que están afectados por la organización del almacenamiento. La entrada correspondiente a 0 en el eje x muestra el porcentaje de los desplazamientos de valor 0. La gran mayoría de los desplazamientos son positivos, pero una gran parte de los mayores desplazamientos (14 + bits) son negativos. De nuevo, esto se debe al esquema de direccionamiento global utilizado por el compilador y puede cambiar con un esquema de compilación diferente. Como estos datos se tomaron en una máquina con desplazamientos de 16 bits, no pueden decirnos nada sobre los accesos que utilicen un desplazamiento mayor. Estos accesos se descomponen en dos instrucciones separadas —la primera de las cuales carga los 16 bits superiores de un registro base—. Al contar la frecuencia de estas instrucciones de «carga inmediata», que tienen un uso limitado para otros propósitos, podemos acotar el número de accesos con desplazamientos potencialmente mayores de 16 bits. Tal análisis indica que GCC, Spice y TeX realmente pueden necesitar un desplazamiento mayor de 16 bits hasta para el 5, 13 y 27 por 100 de referencias a memoria, respectivamente. Además, si el desplazamiento es mayor de 15 bits, es probable que sea un poco mayor ya que muchas constantes que se cargan son grandes, como muestra la Figura 3.15. Para evaluar la elección de la longitud del desplazamiento, podríamos examinar una distribución acumulativa, como se muestra en el Ejercicio 3.3 (ver Fig. 3.35).

sobre qué tamaños soportar. Escoger los tamaños del campo de desplazamiento es importante porque afectan directamente a la longitud de la instrucción. Las medidas realizadas sobre los accesos a los datos en una arquitectura de carga/almacenamiento utilizando nuestros tres programas de benchmark se muestran en la Figura 3.13. En la siguiente sección veremos los desplazamientos de los saltos —los saltos y patrones de acceso a los datos son tan diferentes, que poco se gana combinándolos.

Modo de direccionamiento literal o inmediato

Los inmediatos pueden ser utilizados en operaciones aritméticas, en comparaciones (principalmente para saltos) y en transferencias en las que se quiera una constante en un registro. El último caso se presenta para constantes escritas en el código, que tienden a ser pequeñas, y para constantes de direcciones que pueden ser grandes. Para el uso de los inmediatos, es importante saber si necesitan estar soportados para todas las operaciones o sólo para un subconjunto. El diagrama de la Figura 3.14 muestra la frecuencia de los inmediatos para las clases generales de operaciones de un repertorio de instrucciones.

Otra medida importante del repertorio de instrucciones es el rango de va-

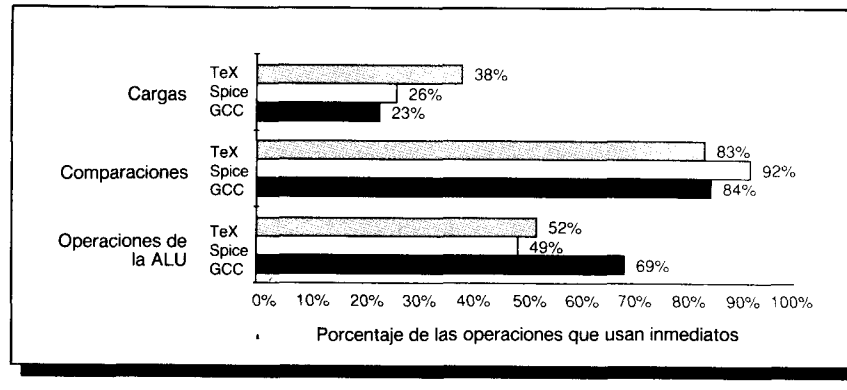


FIGURA 3.14 Vemos que, aproximadamente, la mitad de las operaciones de la ALU tienen un operando inmediato, mientras que más del 85 por 100 de las ocurrencias de las comparaciones utilizan un operando inmediato. (Para las operaciones de la ALU, los desplazamientos en una cantidad constante se incluyen como operaciones con operando inmediato.) Para las cargas, las instrucciones de carga inmediata cargan 16 bits en una de las mitades de un registro de 32 bits. Estas cargas inmediatas no son cargas en sentido estricto ya que no referencian a memoria. En algunos casos, un par de cargas inmediatas pueden utilizarse para cargar una constante de 32 bits, pero esto es raro. Las comparaciones incluyen comparaciones con cero que se utilizan en saltos condicionales. Estas medidas se tomaron en una arquitectura MIPS R2000 con optimización completa del compilador. El compilador intenta utilizar sencillas comparaciones con cero para los saltos siempre que sea posible, ya que estos saltos están eficientemente soportados en la arquitectura.

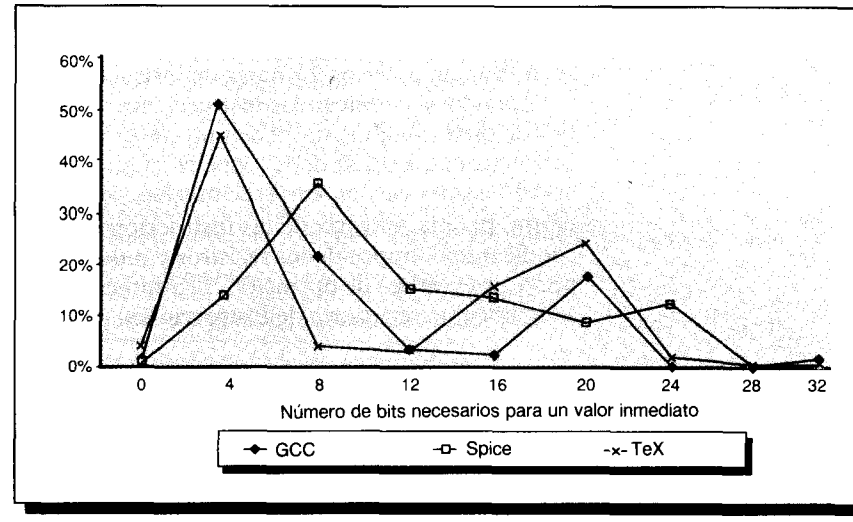


FIGURA 3.15 Distribución de valores inmediatos. *El eje x muestra el número* de bits necesarios para representar la magnitud de un valor inmediato —0 significa que el valor inmediato del campo era 0—. La inmensa mayoría de los valores inmediatos son positivos: globalmente, menos del 6 por 100 de los inmediatos son negativos. Estas medidas se tomaron en un VAX, que soporta un rango completo de tamaño e inmediatos como operandos para cualquier instrucción. Los programas medidos son el conjunto estándar —GCC, Spice y TeX.

lores de los inmediatos. Igual que los valores de los desplazamientos, los tamaños de los valores inmediatos afectan la longitud de la instrucción. Como indica la Figura 3.15, los valores inmediatos pequeños son los más intensamente usados. Sin embargo, a veces, se usan inmediatos grandes en el cálculo de direcciones. Los datos de la Figura 3.15 se tomaron en un VAX, que dispone de muchas instrucciones que tienen cero como operando implícito. Entre éstas hay instrucciones de comparación con cero y de almacenar cero en una palabra. Debido a estas instrucciones, las medidas muestran un uso relativamente infrecuente del cero.

Codificación de los modos de direccionamiento

La forma en que se codifiquen los modos de direccionamiento de los operandos depende del rango de modos de direccionamiento y del grado de independencia entre modos y códigos de operación. Para un pequeño número de modos de direccionamiento o combinaciones de modo de direccionamiento/código de operación, el modo de direccionamiento puede codificarse en el código de operación. Esto funciona para el IBM 360 en el que, con sólo cinco modos de direccionamiento, la mayoría de las operaciones utilizan en sólo uno o dos modos. Para un gran número de combinaciones, normalmente se necesita un *especificador de direcciones* separado para cada operando. El espe-

cificador de direcciones indica el modo de direccionamiento que está usando el operando. En el Capítulo 4, veremos cómo estos dos tipos de codificación se utilizan en diversos formatos de instrucciones reales.

Cuando se codifican las instrucciones, el número de registros y el de modos de direccionamiento tienen un impacto significativo en el tamaño de las instrucciones. Esto se debe a que el campo del modo de direccionamiento y el del registro pueden aparecer muchas veces en una simple instrucción. En efecto, para la mayoría de las instrucciones se emplean muchos más bits en codificar los campos de los registros y modos de direccionamiento que en especificar el código de operación. El arquitecto debe equilibrar diversas fuerzas al codificar el repertorio de instrucciones:

1. El deseo de tener tantos registros y modos de direccionamiento como sea posible.

2. El impacto del tamaño de los campos de los registros y de los modos de direccionamiento en el tamaño medio de la instrucción y, por consiguiente, en el tamaño medio del programa.

3. Un deseo de tener instrucciones codificadas en longitudes que sean fáciles de manejar en la implementación. Como mínimo, el arquitecto quiere instrucciones que sean múltiplos de bytes, mejor que una longitud arbitraria. Muchos arquitectos han escogido el utilizar instrucciones de longitud fija para obtener beneficios en la implementación, aunque se sacrifique el tamaño medio del código.

Como los campos de los modos de direccionamiento y de los registros constituyen un porcentaje elevado de los bits de las instrucciones, su codificación afectará, significativamente, a lo fácil que sea para una implementación decodificar las instrucciones. La importancia de tener instrucciones fácilmente decodificables se explica en los Capítulos 5 y 6.

3.5 Operaciones del repertorio de instrucciones

Los operadores soportados por la mayoría de las arquitecturas, de los repertorios de instrucciones, se pueden clasificar como en la Figura 3.16. En la Sección 3.8, examinaremos el uso de las operaciones de una forma general (por ejemplo, referencias a memoria, operaciones de la ALU y saltos). En el Capítulo 4 examinaremos, con detalle, el uso de las diversas operaciones de las instrucciones para cuatro arquitecturas diferentes. Debido a que las instrucciones utilizadas para implementar el flujo de control son muy independientes de otras elecciones del repertorio de instrucciones y, a que las medidas del comportamiento de los saltos y bifurcaciones también son bastante independientes de otras medidas, examinaremos, a continuación, el uso de las instrucciones del flujo de control.

Tipo de operador	Ejemplos
Aritmético y lógico	Operaciones lógicas y aritméticas enteras: suma, and, resta, or
Transferencia de datos	Cargas/almacenamientos (instrucciones de transferencia en máquinas con direccionamiento de memoria)
Control	Salto, bifurcación, llamada y retorno de procedimiento, traps
Sistema	Llamada al sistema operativo, instrucciones de gestión de memoria virtual
Punto flotante	Operaciones de punto flotante: suma, multiplicación
Decimal	Suma decimal, multiplicación decimal, conversiones de decimal a caracteres
Cadenas	Transferencia de cadenas, comparación de cadenas, búsqueda de cadenas

FIGURA 3.16 Categorías de los operandos de las instrucciones y ejemplos de cada uno. Todas las máquinas generalmente proporcionan un repertorio completo de operaciones para las tres primeras categorías. El soporte para las funciones del sistema en el repertorio de instrucciones varía ampliamente entre arquitecturas, pero todas las máquinas deben tener algún soporte de instrucciones para las funciones básicas del sistema. El soporte que suministre el repertorio de instrucciones para las tres últimas categorías puede variar desde ninguna hasta un amplio conjunto de instrucciones especiales. Las instrucciones de punto flotante estarán disponibles en cualquier máquina diseñada para utilizarlas en aplicaciones que hagan mucho uso del punto flotante. Estas instrucciones a veces forman parte de un repertorio de instrucciones opcional. Las instrucciones decimales y de cadena a veces son primitivas, como en el VAX o en el IBM 360, o el compilador puede sintetizarlas a partir de instrucciones más simples. En el Apéndice B se dan ejemplos de repertorios de instrucciones, mientras que el Apéndice C contiene medidas típicas de utilización. Examinaremos con detalle cuatro repertorios de instrucciones diferentes y su utilización en el Capítulo 4.

Instrucciones para el flujo de control

Como muestra la Figura 3.17, no hay terminología consistente para las instrucciones que cambian el flujo de control. Hasta el IBM 7030, las instrucciones de flujo de control normalmente se denominaban *transferencias*. Comenzando con el 7030, se empezó a utilizar el nombre de *salto (branch)*. Más tarde, las máquinas introdujeron nombres adicionales. A lo largo de este libro nosotros utilizaremos *bifurcación (jump)* cuando el cambio en el control sea incondicional y *salto (branch)* cuando sea condicional.

Podemos distinguir cuatro tipos de cambios del flujo de control:

1. Saltos condicionales
2. Bifurcaciones
3. Llamadas a procedimientos
4. Retornos de procedimientos

Queremos conocer la frecuencia relativa de estos eventos; como cada evento es diferente, podemos usar diferentes instrucciones, y podemos tener diferentes comportamientos. Las frecuencias de estas instrucciones de flujo de con-

Máquina	Año	"Salto" (Branch)	"Bifurcación" (Jump)
IBM 7030	1960	Todas las transferencias de control—direccionamiento es relativo al PC	
IBM 360/370	1965	Todas transferencias de control—no relativo al PC	
DEC PDP-11	1970	Condicional e incondicional, sólo relativo al PC	Todos los modos de direccionamiento; sólo incondicional
Intel 8086	1978		Todas las transferencias son bifurcaciones; las bifurcaciones condicionales sólo son relativas al PC
DEC VAX	1978	Igual que PDP-11	Igual que PDP-11
MIPS R2000	1986	Transferencias de control condicionales, siempre relativas al PC	Instrucciones de llamada y bifurcaciones incondicionales

FIGURA 3.17 Máquinas, fechas y nombres asociados a las transferencias de control de sus arquitecturas. Estos nombres varían ampliamente según que la transferencia sea condicional o incondicional y si es o no relativa al PC. Las arquitecturas VAX, PDP-11 y MIPS R2000 para los saltos sólo permiten direccionamiento relativo al PC.

trol para una máquina de carga/almacenamiento corriendo nuestros benchmarks se muestra en la Figura 3.18.

La dirección de destino de un salto debe especificarse siempre. Este destino, en la mayoría de los casos, se especifica explícitamente en la instrucción

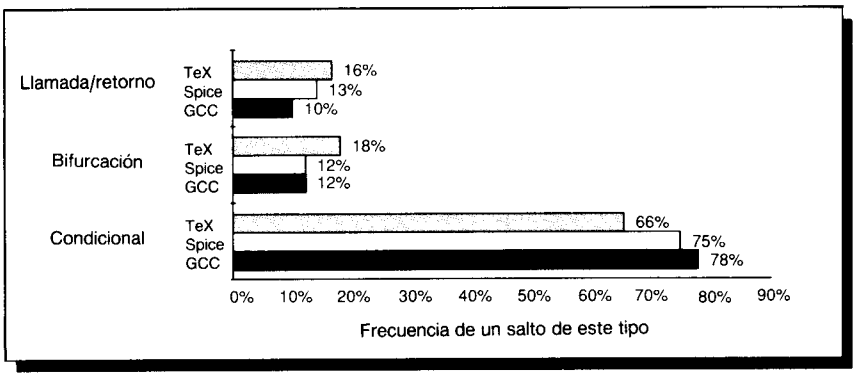


FIGURA 3.18 Descomposición de los saltos en tres clases. Cada salto se contabiliza en una de las tres barras. Los saltos condicionales dominan claramente. Una media del 90 por 100 de las bifurcaciones son relativas al PC.

—siendo el retorno de procedimiento la excepción más importante—, ya que para el retorno, el destino no se conoce en tiempo de compilación. La forma más común de especificar el destino es suministrar un desplazamiento que se sume al *contador de programa,* o PC. Los saltos de este tipo se denominan saltos *relativos al PC.* Los saltos relativos al PC son ventajosos porque el destino del salto, con frecuencia, está cerca de la instrucción actual, y especificar la posición relativa al PC actual requiere pocos bits. La utilización de direccionamiento relativo al PC también permite que el código se ejecute independientemente de dónde esté cargado. Esta propiedad, llamada *independencia de posición,* puede eliminar parte de trabajo cuando se enlace el programa y también es para programas enlazadas durante ejecución.

Para implementar retornos y saltos indirectos en los que el destino no se conoce en tiempo de compilación, se requiere otro método distinto al direccionamiento relativo. Aquí, debe haber una forma de especificar dinámicamente el destino, ya que puede cambiar en tiempo de ejecución. Esto puede ser tan fácil como nombrar un registro que contenga la dirección del destino.

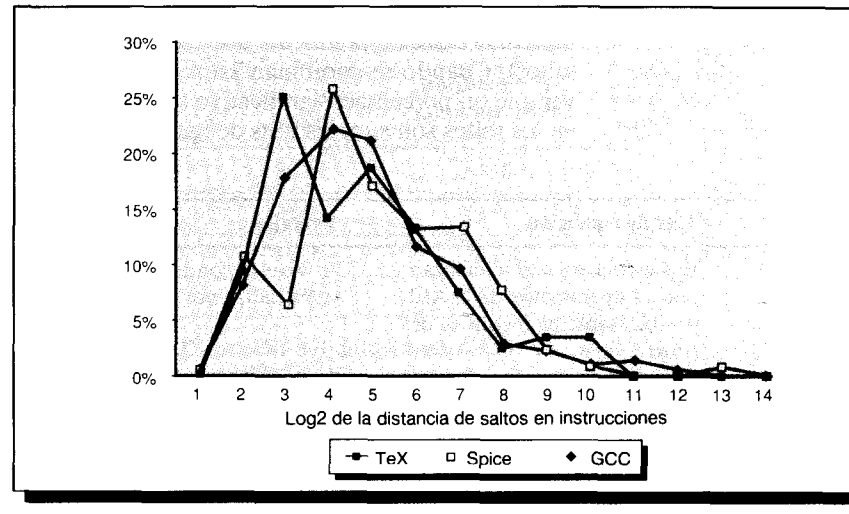


FIGURA 3.19 Distancias de los saltos en función del número de instrucciones entre el destino y la instrucción de salto. Los saltos más frecuentes en Spice son a destinos cuya distancia es de 8 a 15 instrucciones (2^4). La distancia (media aritmética ponderada) del destino del salto es de 86 instrucciones (2^7). Esto nos dice que campos cortos de desplazamiento, con frecuencia, bastan para los saltos y que el diseñador puede ganar en densidad de codificación con una instrucción más corta con un desplazamiento de salto menor. Estas medidas se tomaron en una máquina de carga/almacenamiento (arquitectura MIPS R2000). Una arquitectura que requiere menos instrucciones para el mismo programa, tal como el VAX, podría tener distancias de salto más cortas. Análogamente, el número de bits necesarios para el desplazamiento puede cambiar si la máquina permite que las instrucciones se alineen arbitrariamente. Una distancia acumulativa de estos datos de desplazamientos de saltos se muestra en el Ejercicio 3.3 (ver Fig. 3.35).

Alternativamente, el salto puede permitir que se utilice cualquier modo de direccionamiento para obtener la dirección del destino.

Una pregunta clave se refiere a la distancia que los destinos de los saltos están de los saltos. Conocer la distribución de estos desplazamientos ayudará a elegir qué desplazamientos de salto soportar y, por tanto, cómo afectarán a la longitud y codificación de las instrucciones. La Figura 3.19 muestra la distribución, en instrucciones, de los desplazamientos para saltos relativos al PC. Aproximadamente el 75 por 100 de los saltos son en dirección adelante.

Como la mayoría de los cambios en el flujo de control son saltos, es importante decidir cómo especificar la condición de salto. Las tres técnicas principales que se utilizan así como ventajas y desventajas se muestran en la Figura 3.20.

Una de las propiedades más notables de los saltos es que la mayor parte de las comparaciones son sencillamente tests de igualdad o desigualdad, y un gran número son comparaciones con cero. Así, algunas arquitecturas deciden tratar estas comparaciones como casos especiales, fundamentalmente si se utiliza una instrucción de *comparación y salto.* La Figura 3.21 muestra la frecuencia de las diferentes comparaciones utilizadas para saltos condicionales. Los datos de la Figura 3.14 indican que un gran porcentaje de saltos tenía un operando inmediato (86 por 100), y aunque no se mostraba, era 0 el operando inmediato más ampliamente utilizado (el 83 por 100 de los inmediatos en los saltos). Cuando se combinan estos datos con los de la Figura 3.21 podemos ver que un porcentaje significativo (sobre el 50 por 100) de las comparaciones en los saltos son simples tests de igualdad con cero.

Nombre	Test de condición	Ventajas	Desventajas
Código de condición (CC)	Bits especiales son inicializados por las operaciones de la ALU, posiblemente bajo control del programa.	A veces la condición se inicializa por libre.	CC es un estado extra. Los códigos de condición restringen la ordenación de las instrucciones ya que pasan información de una instrucción a un salto.
Registro de condición	Examina un registro arbitrario con el resultado de una comparación	Simple.	Utiliza un registro.
Comparación y salto	La comparación es parte del salto. Con frecuencia la comparación está limitada a subconjuntos.	Una instrucción en lugar de dos para un salto.	Puede ser demasiado trabajo por instrucción.

FIGURA 3.20 Métodos principales para evaluar condiciones de salto, ventajas y desventajas. Aunque los códigos de condición pueden ser utilizados por operaciones de la ALU que son necesarias para otros propósitos, medidas en los programas muestran que raramente ocurre esto. Los principales problemas de implementación con los códigos de condición surgen cuando el código de condición es inicializado por un subconjunto de instrucciones grande o escogido al azar, en lugar de estar controlado por un bit de la instrucción. Las máquinas con comparaciones y saltos limitan, con frecuencia, el conjunto de comparaciones y utilizan un registro de condición para comparaciones más complejas. Con frecuencia, se utilizan técnicas diferentes para los saltos basados en comparaciones de punto flotante frente a otros basados en comparaciones enteras. Esto es razonable ya que el número de saltos que dependen de las comparaciones en punto flotante es mucho menor que el número que depende de las comparaciones enteras.

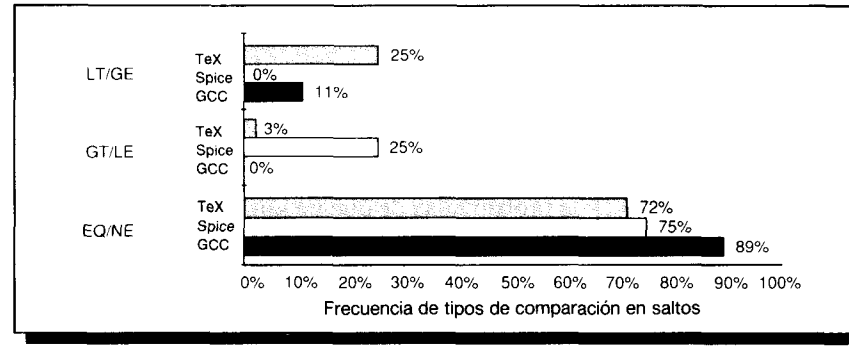


FIGURA 3.21 Frecuencia de diferentes tipos de comparaciones en saltos condicionales. Incluye tanto las comparaciones enteras como las de punto flotante en los saltos. Las comparaciones de punto flotante constituyen el 13 por 100 de las comparaciones de saltos en Spice. Recordar que los primeros datos de la Figura 3.14 indican que muchas comparaciones son con un operando inmediato. Este valor inmediato habitualmente es 0 (83 por 100 de las veces).

Programa	Porcentaje de saltos hacia atrás	Porcentaje de saltos efectivos	Porcentaje de todas las instrucciones de control que realmente saltan
GCC	26 %	54 %	63 %
Spice	31 %	51 %	63 %
TeX	17 %	54 %	70 %
Media	25 %	53 %	65 %

FIGURA 3.22 Dirección del salto, frecuencia de saltos efectivos y frecuencia con la que se cambia el PC. La primera columna muestra qué porcentaje de todos los saltos (efectivos y no efectivos) se realizan hacia atrás. La segunda muestra el porcentaje de saltos efectivos (recordar que un salto siempre es condicional). La columna final muestra el porcentaje de todas las instrucciones de flujo de control que realmente provocan una transferencia no secuencial en el flujo. Esta última columna se calcula al combinar los datos de la segunda columna y los datos de la Figura 3.18.

Diremos que un *salto* se realiza si la condición probada por el salto es verdadera y la siguiente instrucción que se va a ejecutar es el destino del salto. Todas las bifurcaciones, por tanto, son efectivos. La Figura 3.22 muestra la distribución de la dirección de los saltos, la frecuencia de los saltos (condicionales) efectivos, y el porcentaje de instrucciones de flujo de control que cambian el PC. La mayoría de los saltos hacia atrás son saltos de bucle, y normalmente son efectivos con una probabilidad del 90 por 100.

Muchos programas tienen un porcentaje más elevado de saltos de bucle, incrementando así la frecuencia de los saltos efectivos por encima del 60 por

100. En general, el comportamiento de los saltos es dependiente de la aplicación y, a veces, del compilador. Las dependencias del compilador surgen debido a los cambios de flujo de control realizados por los compiladores optimizadores para mejorar el tiempo de ejecución de los bucles.

Ejemplo

Suponiendo que el 90 por 100 de los saltos hacia atrás son efectivos, calcular la probabilidad de que un salto hacia adelante sea efectivo utilizando los datos promediados de la Figura 3.22.

Respuesta

La frecuencia media de los saltos efectivos es la suma de las frecuencias de los efectivos hacia atrás y de los efectivos hacia adelante:

$$\begin{aligned}\%\text{ saltos efectivos} &= (\%\text{ efectivos hacia atrás} \cdot \%\text{ hacia atrás}) + \\ &\quad (\%\text{ efectivos hacia adelante} \cdot \%\text{ hacia adelante})\end{aligned}$$

$$53\,\% = (90\,\% \cdot 25\,\%) + (\%\text{ efectivos hacia adelante} \cdot 75\,\%)$$

$$\%\text{ efectivos hacia adelante} = \frac{53\,\% - 22{,}5\,\%}{75\,\%}$$

$$\%\text{ efectivos hacia adelante} = 40{,}7\,\%$$

No es inusual ver que la mayoría de los saltos hacia adelante son no efectivos. El comportamiento de los saltos hacia atrás varía, con frecuencia, entre programas.

Las llamadas y retornos de procedimiento incluyen transferencia de control y posiblemente guardar algún estado; como mínimo, la dirección de retorno debe guardarse en algún sitio. Algunas arquitecturas proporcionan un mecanismo para guardar los registros, mientras otras requieren que el compilador genere instrucciones. Hay dos convenios básicos que se usan para guardar registros. *Guarda-llamador (caller-saving)* significa que el procedimiento que llama debe guardar los registros que quiere preservar para los accesos después de la llamada. *Guarda-llamado (called-saving)* significa que el procedimiento llamado debe guardar los registros que quiera utilizar. Hay veces que debe utilizarse guarda-llamador debido a patrones de acceso a variables globalmente visibles en dos procedimientos diferentes. Por ejemplo, supongamos que tenemos un procedimiento P_1 que llama al procedimiento P_2, y ambos procedimientos manipulan la variable global x. Si P_1 hubiese ubicado a x en un registro, debe asegurarse de guardarla en una posición conocida por P_2 antes de la llamada a P_2. La posibilidad de que un compilador descubra cuándo un procedimiento llamado, puede acceder a cantidades ubicadas en un registro es complicado por la posibilidad de compilación separada, y situaciones donde P_2 puede no tocar a x, pero pueda llamar a otro procedimiento, P_3, que pueda acceder a x. Debido a estas complicaciones, la mayoría de los compiladores harán que el llamador guarde, conservadoramente, **cualquier variable** que pueda ser **accedida** durante una llamada.

En los casos donde puedan ser utilizados uno de los dos convenios, algunos son más óptimos con el guarda-llamador *(callee-save)* y otros con el guarda-

llamado *(called-save)*. Como resultado, los compiladores más sofisticados utilizan una combinación de los dos mecanismos, y el ubicador de registros puede elegir qué registro utilizar para una variable según el convenio. Más tarde, en este capítulo, examinaremos cómo las instrucciones más sofisticadas se adaptan a las necesidades del compilador para esta función, y en el Capítulo 8 examinaremos esquemas hardware de memoria para soportar el almacenamiento y la restauración de registros.

3.6 Tipo y tamaño de los operandos

¿Cómo se designa el tipo de un operando? Hay dos alternativas importantes: Primero, el tipo de un operando puede designarse al codificarlo en el código de operación —éste es el método utilizado con más frecuencia. Alternativamente, el dato puede anotarse con identificadores que son interpretados por el hardware. Estos identificadores especifican el tipo de operando y, consiguientemente, la operación escogida. Sin embargo, máquinas con datos identificados son extremadamente raras. Las arquitecturas de Burroughs son el ejemplo más amplio de arquitecturas identificadas *(tagged architectures)*. Symbolics también construyó una serie de máquinas que utilizaban elementos de datos identificados para implementaciones LISP.

Habitualmente el tipo de un operando —por ejemplo, entero, simple precisión, punto flotante, carácter— da efectivamente su tamaño. Los tipos de operando comunes incluyen carácter (un byte), media palabra (16 bits), palabra (32 bits), punto flotante en simple precisión (también una palabra), y punto flotante en doble precisión (dos palabras). Los caracteres se representan o como EBCDIC, utilizado por las arquitecturas de los grandes computadores IBM, o ASCII, utilizado por las demás. Los enteros están casi universalmente representados como números binarios en complemento a 2. Hasta hace poco, la mayoría de los fabricantes de computadores elegían su propia representación en punto flotante. Sin embargo, en los últimos años, un estándar para el punto flotante, el estándar 754 del IEEE ha sido la elección de la mayoría de los nuevos computadores. El estándar de punto flotante el IEEE se explica con detalle en el Apéndice A.

Algunas arquitecturas proporcionan operaciones sobre cadenas de caracteres, aunque estas operaciones están, habitualmente, bastante limitadas y tratan cada byte de la cadena como un carácter. Las operaciones normalmente soportadas sobre cadenas de caracteres son comparaciones y desplazamientos.

Para aplicaciones comerciales, algunas arquitecturas soportan un formato decimal, denominado habitualmente *decimal empaquetado*. El decimal empaquetado es *decimal codificado binario* —se utilizan 4 bits para codificar los valores 0-9, y en cada byte se empaquetan dos dígitos decimales. Las cadenas de caracteres numéricos se denominan, a veces, *decimal desempaquetado*, y las operaciones —denominadas de empaquetamiento y desempaquetamiento— normalmente se suministran para realizar conversiones entre ellos.

Nuestros benchmarks utilizan los tipos de datos byte o carácter, media palabra (entero corto), palabra (entero) y punto flotante. La Figura 3.23 muestra la distribución dinámica de los tamaños de los objetos referenciados desde

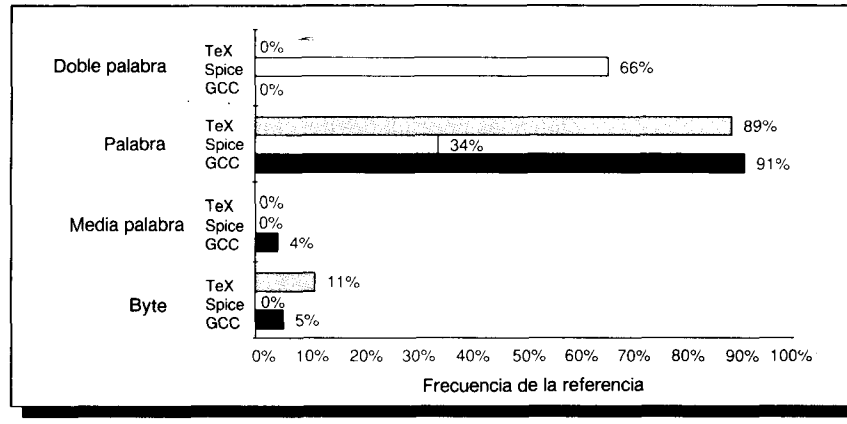


FIGURA 3.23 Distribución de los accesos a los datos por tamaños para los programas benchmark. Los accesos a los tipos principales de datos (palabra o doble palabra) dominan claramente. Las lecturas exceden al número de escrituras de datos en un factor de 1,6 para TeX y en un factor de 2,5 para Spice. El tipo de dato doble palabra se utiliza únicamente para punto flotante con doble precisión en Spice. Spice hace poco uso del punto flotante en simple precisión; la mayoría de las referencias a palabras en Spice son para los enteros. Estas medidas se tomaron en el tráfico de memoria generado en una arquitectura de carga/almacenamiento.

memoria por estos programas. La frecuencia de accesos a los diferentes tipos de datos ayuda a decidir qué tipos son más importantes de soportar eficientemente. ¿Debería tener la máquina un camino de acceso de 64 bits, o debería emplear dos ciclos para que el acceso a una doble palabra fuese satisfactorio? ¿Es muy importante soportar accesos de bytes como primitivas, que, como vimos antes, requieren una red de alineación? En la Figura 3.23, se utilizan referencias a memoria para examinar los tipos de datos a los que se va a acceder. En algunas arquitecturas, los objetos de los registros pueden ser accedidos como bytes o medias palabras. Sin embargo, tal acceso es muy infrecuente —en el VAX, no contabiliza más del 12 por 100 de las referencias de registros, o aproximadamente el 6 por 100 de todos los accesos a operandos en estos programas.

En el capítulo siguiente examinaremos ampliamente lás diferencias de las mezclas de instrucciones y de otras medidas arquitectónicas sobre cuatro máquinas muy diferentes. Pero antes de hacer eso, será útil echar un breve vistazo a la tecnología de los modernos compiladores y su efecto en las propiedades de los programas.

3.7 El papel de los lenguajes del alto nivel y los compiladores

Hoy día la mayor parte de la programación se realiza en lenguajes de alto nivel. Esto significa que como la mayoría de las instrucciones ejecutadas son la

salida de un compilador, una arquitectura a nivel lenguaje máquina es esencialmente un objeto del compilador. En los primeros tiempos, las decisiones sobre arquitectura se tomaban, con frecuencia, para realizar la programación en lenguaje ensamblador. Debido a que el rendimiento de un computador estará significativamente afectado por el compilador, la comprensión de la tecnología de los compiladores actuales es crítica para diseñar e implementar eficientemente un repertorio de instrucciones. En los primeros días era popular intentar aislar la tecnología de compiladores y su efecto sobre el rendimiento hardware de la arquitectura y su rendimiento, lo mismo que era popular intentar separar una arquitectura de su implementación. Esto es extremadamente difícil, si no imposible, con los compiladores y arquitecturas de hoy. Las elecciones arquitectónicas afectan a la calidad del código que se puede generar para una máquina y a la complejidad de construir un buen compilador para ella. Aislar el compilador del hardware es probablemente un gran error. En esta sección explicaremos los objetivos críticos del repertorio de instrucciones, principalmente desde el punto de vista del compilador. ¿Qué características nos llevarán a un código de alta calidad? ¿Qué hace fácil escribir compiladores eficientes para una arquitectura?

La estructura de los compiladores recientes

Para comenzar, veamos cómo son los compiladores actuales. La estructura de los compiladores recientes se muestra en la Figura 3.24.

El primer objetivo de un escritor de compiladores es la exactitud —todos los programas válidos deben ser compilados correctamente. El segundo objetivo es habitualmente la velocidad del código compilado. Normalmente, a estos dos primeros objetivos siguen un conjunto completo de más objetivos. Se incluyen compilación rápida, soporte de depuración e interoperabilidad entre lenguajes. Normalmente, los pasos del compilador transforman representaciones más abstractas de más alto nivel en representaciones progresivamente de más bajo nivel, alcanzando eventualmente el repertorio de instrucciones. Esta estructura ayuda a gestionar la complejidad de las transformaciones y hace más fácil la escritura de un compilador libre de errores. La complejidad de escribir un compilador correcto es una limitación importante en la cantidad de optimizaciones que puedan hacerse. Aunque la estructura de múltiples pasos ayuda a reducir la complejidad del compilador, también significa que el compilador debe ordenar y realizar unas transformaciones antes que otras. En el diagrama del compilador optimizador de la Figura 3.24, podemos ver que ciertas optimizaciones de alto nivel se realizan mucho antes que se sepa cómo será el código resultante en detalle. Una vez que tal transformación se haga, el compilador no puede permitirse ir hacia atrás y revisitar todos los pasos, posiblemente deshaciendo transformaciones. Esto sería prohibitivo, tanto en tiempo de compilación como en complejidad. Así, los compiladores hacen hipótesis sobre la posibilidad de pasos posteriores para tratar ciertos problemas. Por ejemplo, los compiladores habitualmente tienen que elegir qué llamadas a procedimiento expandir en línea antes de saber el tamaño exacto del procedimiento que se va a llamar. Los escritores de compiladores llaman a este problema el *problema de ordenación de fase (phase-ordering).*

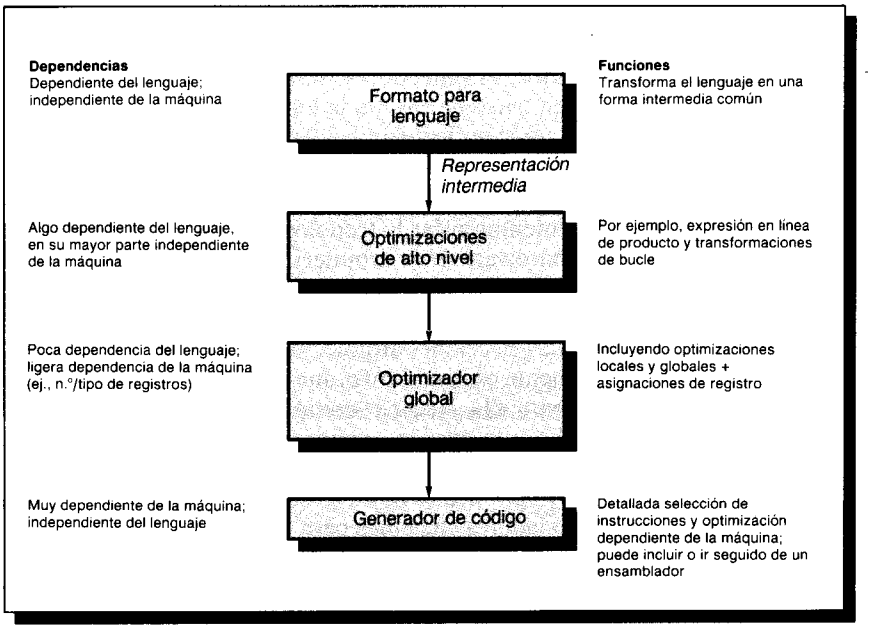


FIGURA 3.24 Los compiladores actuales, normalmente, constan de dos a cuatro pasos; con más pasos los compiladores que realizan una mayor optimización. Un *paso* es simplemente una fase en la cual el compilador lee y transforma el programa completo. (El término «fase» es, con frecuencia, intercambiable con «paso».) Los pasos de optimización se designan para que sean opcionales y puedan saltarse cuando el objetivo sea una compilación más rápida y sea aceptable un código de menor calidad. Esta estructura maximiza la probabilidad de que un programa compilado en varios niveles de optimización produzca la misma salida cuando tenga la misma entrada. Debido a que los pasos de optimización también están separados, múltiples lenguajes pueden utilizar los mismos pasos de optimización y generación de código. Para un nuevo lenguaje, se requiere solamente un nuevo formato. La optimización de alto nivel mencionada aquí, procedimiento en línea, también se denomina *integración de procedimiento.*

¿Cómo interactúa esta ordenación de las transformaciones con la arquitectura a nivel lenguaje máquina? Un buen ejemplo se presenta con la optimización denominada *eliminación global de subexpresiones comunes.* Esta optimización encuentra dos instancias de una expresión, que calculan el mismo valor, y guarda el valor del primer cálculo de forma temporal. Entonces utiliza el valor temporal, eliminando el segundo cálculo de la expresión. Para que esta optimización sea significativa, el valor temporal se debe ubicar en un registro. De otro modo, el costo de almacenar el valor temporal en memoria y más tarde recargarlo puede anular los ahorros obtenidos al no recalcular la expresión. En efecto, hay casos donde esta optimización ralentiza, realmente, el código cuando el valor temporal no está ubicado en el registro. La ordenación de fase complica este problema, ya que la ubicación de registros se realiza normalmente casi al final del paso de optimización global, justo antes de la

generación de código. Por tanto, un optimizador que realice esta optimización **debe** suponer que el ubicador de registros colocará los valores temporales en un registro.

Debido al papel central que juega la ubicación de registros, tanto en acelerar el código como en hacer útiles otras optimizaciones, es una de las optimizaciones más importantes —si no la más—. Los algoritmos recientes de ubicación de registros están basados en una técnica denominada *coloreado de grafos*. La idea básica de esta técnica es construir un grafo que represente los posibles candidatos a ubicar en un registro, y después utilizar el grafo para asignar los registros. Como muestra la Figura 3.25, cada candidato a un registro corresponde a un nodo del grafo, llamado *grafo de interferencias*. Los arcos entre los nodos muestran dónde se solapan los rangos de utilización de las variables (llamados *rangos vivos)*. El compilador intenta entonces colorear el grafo utilizando una serie de colores igual al número de registros disponibles para la ubicación. En una coloración del grafo, los nodos adyacentes no pueden tener el mismo color. Esta restricción es equivalente a decir que dos variables con solapamiento no se pueden ubicar en el mismo registro. Sin embargo, los nodos que no estén conectados por ningún arco pueden tener el

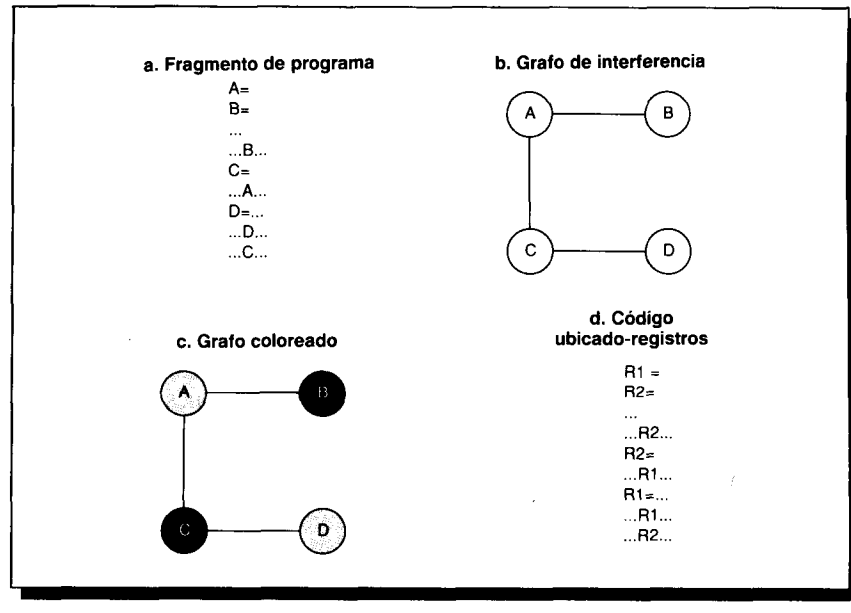


FIGURA 3.25 El coloreado de grafos se utiliza para ubicar registros al construir un grafo de interferencia que se colorea heurísticamente usando un número de colores que se corresponden con el número de registros. La parte b muestra el grafo de interferencia correspondiente al fragmento de código mostrado en la parte a. Cada variable corresponde a un nodo, y los arcos muestran el solapamiento de los rangos activos de las variables. El grafo puede ser coloreado con dos colores, como se muestra en la parte c, y esto corresponde a la ubicación de registros de la parte d.

mismo color, permitiendo utilizar el mismo registro a las variables cuyos usos no se solapen Por tanto, una coloración del grafo corresponde a una ubicación de las variables activas en los registros. Por ejemplo, los cuatro nodos de la Figura 3.25 pueden colorearse con dos colores, significando que el código sólo necesita dos registros para la ubicación. Aunque el problema de colorear un grafo es NP-completo, en la práctica, hay algoritmos heurísticos que funcionan bien.

La coloración de grafos funciona mejor cuando hay disponibles como mínimo 16 (y preferiblemente más) registros de propósito general para la ubicación de variables enteras y registros adicionales para la de variables en punto flotante. Desgraciadamente, la coloración del grafo no funciona bien cuando el número de registros es pequeño porque, probablemente, fallan los algoritmos heurísticos de coloración del grafo. El énfasis en la aproximación es conseguir el 100 por 100 de ubicación de las variables activas.

Las optimizaciones realizadas por los compiladores modernos pueden clasificarse por el estilo de la transformación, como sigue:

1. Optimizaciones de alto nivel: realizadas con frecuencia en la fuente con la salida conectada a pasos de optimización posteriores.
2. Optimizaciones locales: optimizan código sólo de un fragmento de código lineal (denominado *bloque básico* por la gente de los compiladores).
3. Optimizaciones globales: extienden las optimizaciones locales a través de los saltos e introducen un conjunto de transformaciones dedicadas a optimizar bucles.
4. Ubicación de registros.
5. Optimizaciones dependientes de la máquina: intentan aprovechar el conocimiento de arquitecturas específicas.

A veces es difícil separar algunas de las optimizaciones más sencillas —optimizaciones locales y dependientes de la máquina— de las transformaciones realizadas en el generador de código. Ejemplos de optimizaciones típicas se dan en la Figura 3.26. La última columna de la Figura 3.26 indica la frecuencia con la que las transformaciones de optimización listadas se aplicaron al programa fuente. En la figura se muestran los datos sobre el efecto de diversas optimizaciones en el tiempo de ejecución de un programa. Los datos de la Figura 3.27 demuestran la importancia de la ubicación de registros, que proporciona la mejora más importante. Examinaremos más tarde, en esta sección, el efecto global de la optimización en nuestros tres benchmarks.

El impacto de la tecnología de compiladores en las decisiones del arquitecto

La interacción de los compiladores y lenguajes de alto nivel afecta significativamente a la forma en que los programas utilizan un repertorio de instrucciones. Para comprender mejor esta interacción, tres importantes preguntas a realizar son:

Nombre de la optimización	Explicación	Porcentaje del número total de transformaciones de optimización
Alto nivel	**En o próximo al nivel fuente; independiente de la máquina**	
Integración de procedimientos	Sustituye la llamada al procedimiento por el cuerpo del procedimiento	N.M.
Local	**Con código lineal**	
Eliminación de subexpresiones comunes	Sustituye dos instancias del mismo cálculo por simple copia	18 %
Propagación de constantes	Sustituye todas las instancias de una variable que se asigna a una constante por la constante	22 %
Reducción de la altura de la pila	Reorganiza el árbol de la expresión para minimizar los recursos necesarios para la evaluación de la expresión	N.M.
Global	**A través de un salto**	
Eliminación global de subexpresiones comunes	Igual que la local, pero esta versión cruza los saltos	13 %
Propagación de copia	Sustituye por X todas las instancias de una variable A a la que se le ha asignado X (p. ej., $A=X$)	11 %
Movimiento de código	Elimina código de un bucle que calcula el mismo valor en cada iteración del bucle	16 %
Eliminación de variables de inducción	Simplifica/elimina cálculo de direccionamiento del array en bucles	2 %
Dependiente de la máquina	**Depende del conocimiento de la máquina**	
Reducción de potencia	Muchos ejemplos, como sustituir multiplicar por una constante por sumas y desplazamientos	N.M.
Planificación de segmentación	Reordenar las instrucciones para mejorar rendimiento de la segmentación	N.M.
Optimización del desplazamiento de saltos	Escoger el desplazamiento de salto más corto que consiga el objeto	N.M.

FIGURA 3.26 Tipos principales de optimizaciones y ejemplos de cada clase. La tercera columna lista la frecuencia estática con la que se aplican algunas de las optimizaciones comunes a un conjunto de 12 pequeños programas en FORTRAN y Pascal. El porcentaje es la porción de las optimizaciones estáticas que son del tipo especificado. Estos datos nos indican aproximadamente la frecuencia relativa de las ocurrencias de diversas optimizaciones. Hay nueve optimizaciones globales y locales hechas por el compilador incluidas en las medidas. Seis de estas optimizaciones aparecen en la figura y las tres restantes contabilizan el 18 por 100 del total de ocurrencias estáticas. La abreviatura «N.M.» significa que no se midió el número de ocurrencias de esa optimización. Las optimizaciones dependientes de la máquina habitualmente se hacen en un generador de código, y ninguna de esas fue medida en este experimento. Los datos son de Chow [1983], y fueron tomados utilizando el compilador Stanford UCODE.

Optimizaciones realizadas	Porcentaje más rápido
Sólo integración de procedimiento	10 %
Sólo optimizaciones locales	5 %
Optimizaciones locales + ubicación de registros	26 %
Optimizaciones globales y locales	14 %
Optimizaciones locales y globales + ubicación de registros	63 %
Optimizaciones locales y globales + integración de procedimientos + ubicación de registros	81 %

FIGURA 3.27 Efectos del rendimiento de varios niveles de optimización. Las ganancias de rendimiento se muestran como el porcentaje más rápido en que los programas optimizados se compararon con programas no optimizados. Cuando se desactiva la ubicación de registros, los datos se cargan en, o almacenan desde, los registros en cada uso individual. Estas medidas también son de Chow [1983] y son para 12 pequeños programas en Pascal y FORTRAN.

1. ¿Cómo se ubican y direccionan las variables? ¿Cuántos registros se necesitan para ubicar adecuadamente las variables?
2. ¿Cuál es el impacto de las técnicas de optimización en las mezclas de instrucciones?
3. ¿Qué estructuras de control se utilizan y con qué frecuencia?

Para responder las primeras preguntas, debemos examinar las tres áreas separadas, en las cuales los lenguajes de alto nivel actuales ubican sus datos:

- La *pila:* utilizada para ubicar variables locales. La pila crece y decrece en las llamadas o retornos de procedimiento, respectivamente. Los objetos en la pila tienen direcciones relativas al puntero de pila y son principalmente escalares (variables simples) en lugar de arrays. La pila se utiliza para registros de activación, **no** como una pila para evaluar expresiones. Por consiguiente los valores casi nunca se introducen ni se sacan de la pila.
- El área *global de datos:* utilizada para ubicar objetos declarados estáticamente, como variables globales y constantes. Un gran porcentaje de estos objetos son los arrays u otras estructuras de datos globales.
- El *montículo (heap):* utilizado para ubicar objetos dinámicos que no se adhieren a una disciplina de pila. Los objetos del montículo son accedidos con punteros y normalmente no son escalares.

La asignación de registros es mucho más efectiva para los objetos ubicados en pilas que para las variables globales, y la asignación de registros es esencialmente imposible para los objetos ubicados en montículos, ya que se accede a ellos con punteros. Las variables globales y algunas variables de pila

son imposible de ubicar porque tienen *alias,* lo que significa que hay múltiples formas de referenciar la dirección de una variable haciendo ilegal ponerla en un registro. (Todas las variables del montículo tienen alias.) Por ejemplo, considerar la siguiente secuencia de código (donde & devuelve la dirección de una variable y * desrreferencia un puntero):

```
p = &a      -- pone dirección de «a» en «p»
a = ...     -- asigna a «a» directamente
*p = ...    -- usa p para asignar a «a»
...a...     -- accede «a»
```

La variable «a» no puede ser ubicada en registro a través de la asignación a `*p` sin generar código incorrecto. Los alias causan un problema sustancial porque, con frecuencia, es difícil o imposible decidir qué objetos puede referenciar un puntero. Un compilador debe ser conservador; muchos compiladores no ubican **ninguna** variable local de un procedimiento en un registro cuando hay un puntero que puede referenciar a **una** de las variables locales.

Después de la ubicación de registros, el tráfico de memoria consta de cinco tipos de referencias:

1. Referencia desubicada: una referencia de memoria potencialmente ubicable que no fue asignada a ningún registro.
2. Escalar global: una referencia a una variable escalar global no ubicada en ningún registro. Estas variables habitualmente son accedidas dispersamente y, por tanto, raramente ubicadas.
3. Referencia de guardar/restaurar en memoria: una referencia a memoria realizada para guardar o restaurar un registro (durante una llamada a procedimiento) que contenga una variable ubicada y no tenga alias.
4. Referencia de pila requerida: cuando se requiere una referencia a una variable de pila debido a posibilidades de alias. Por ejemplo, si se tomase la dirección de la variable de pila, entonces esa variable no podría ubicarse habitualmente en ningún registro. También se incluyen en esta categoría los items de datos que fueron guardados por el llamador debido al comportamiento del alias —tales como una referencia potencial por el procedimiento llamado.
5. Una referencia calculada: cualquier referencia al montículo o cualquier referencia a una variable de pila vía un puntero o un índice de array.

La Figura 3.28 muestra cómo estas clases de tráfico de memoria contribuyen al tráfico total de memoria para los benchmarks GCC y TeX ejecutados con un compilador optimizador en una máquina de carga/almacenamiento y variando el número de registros usados. El número de referencias de memoria a los objetos, comprendidos en las categorías 2 a 5 anteriores, es constante porque nunca pueden ser ubicados en registros por este compilador. (Las referencias de guardar/restaurar se midieron con el conjunto completo de registros.) El número de referencias ubicables que no se ubican cae cuando aumenta la cuenta de registros. Las referencias a objetos que podían ser ubi-

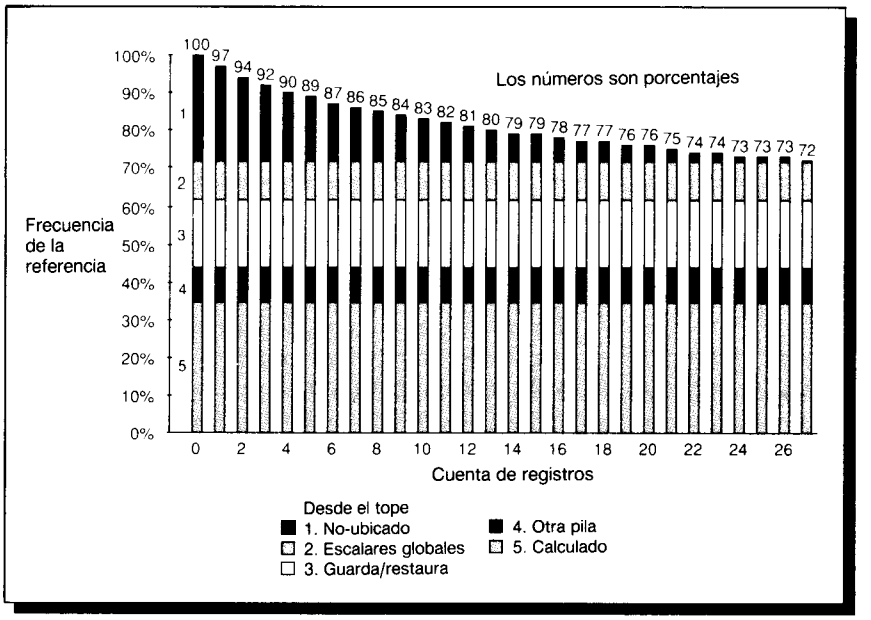


FIGURA 3.28 El porcentaje de referencias a memoria hecho a diferentes tipos de variables cuando aumenta el número de registros. Estos datos son promediados entre TeX y GCC, que utilizan sólo variables enteras. El porcentaje decreciente representado por la barra superior es el conjunto de referencias que son candidatas para la ubicación pero que realmente no están ubicadas en el número indicado de registros. Estos datos fueron tomados con la máquina de carga/almacenamiento DLX descrita en el capítulo siguiente. El ubicador de registros tiene 27 registros enteros; los primeros siete registros enteros capturan aproximadamente la mitad de las referencias que pueden ser ubicadas en los registros. Aunque cada uno de los otros cuatro componentes contribuye algo al tráfico de memoria restante, la contribución dominante de referencias se obtiene para objetos basados en montículo y elementos de array, que no pueden ser ubicados en registros. Un pequeño porcentaje de las referencias de pila requeridas puede deberse a que el ubicador de registros adapta los registros; sin embargo, a partir de otras medidas sobre el ubicador de registros sabemos que esta contribución es muy pequeña [Chow y Hennessy, 1990].

cados, pero que son accedidos sólo una vez, son ubicados por el generador de código utilizando un conjunto de registros temporales. Estas referencias están contabilizadas como referencias requeridas de pila; otras estrategias de ubicación pueden hacer que sean tratadas como tráfico de «guardar/restaurar».

Los datos de la Figura 3.28 muestran solamente los registros enteros. El porcentaje de referencias ubicables, con un número dado de registros, se calcula examinando la frecuencia de accesos a los registros con un compilador que, generalmente, trate de utilizar un número de registros tan pequeño como sea posible. El porcentaje de las referencias capturadas en un número dado de registros depende, estrechamente, del compilador y de su estrategia de ubica-

ción de registros. Este compilador no puede utilizar más de 27 registros enteros disponibles para ubicar variables; adicionalmente algunos registros tienen un uso preferente (como los utilizados para parámetros). De estos datos no podemos predecir cómo el compilador podría utilizar mejor 100 registros. Dado un número sustancialmente mayor de registros, el compilador los podría utilizar para reducir las referencias de «guardar/restaurar» en memoria y las referencias a escalares globales. Sin embargo, ninguna clase de referencia a memoria puede eliminarse completamente. En el pasado, la tecnología de compiladores había hecho progresos uniformes para utilizar siempre conjuntos mayores de registros y probablemente podemos esperar que esto continúe, aunque el porcentaje de referencias ubicables pueda limitar el beneficio de conjuntos mayores de registros.

La Figura 3.29 muestra el mismo tipo de datos, pero esta vez para Spice, que utiliza ambos registros: enteros y de punto flotante. El efecto de la ubicación de registros es muy diferente para Spice comparado con GCC y TeX. En primer lugar, el porcentaje de tráfico de memoria restante es menor. Esto

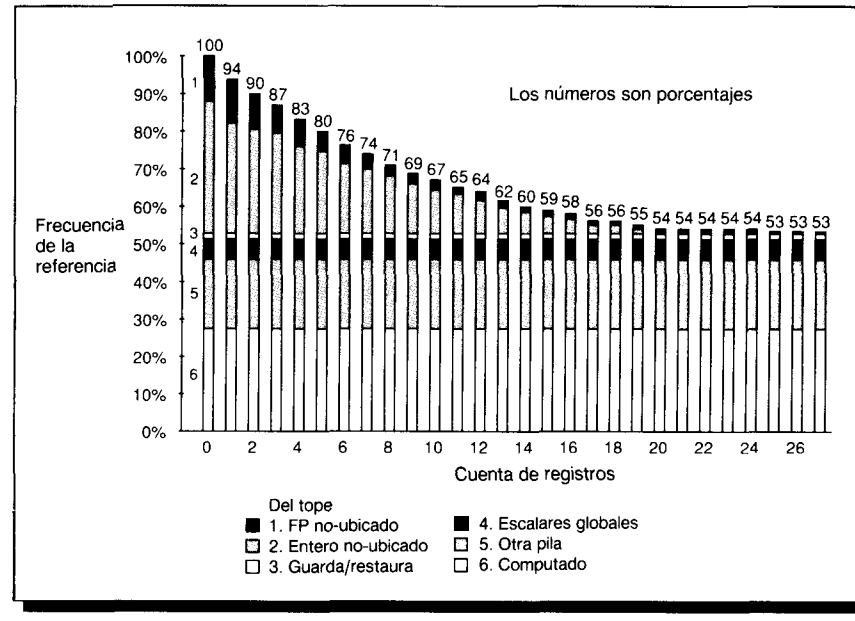


FIGURA 3.29 El porcentaje de referencias capturadas por los ficheros de registros enteros y de punto flotante para Spice incrementa en casi el 50 por 100 con un conjunto completo de registros. Cada incremento en el eje x suma un registro entero y otro registro de simple precisión (SP) de punto flotante (FP). Así, el punto que corresponde a una cuenta de 12 registros significa 12 registros enteros y 12 registros FP, SP. Recordar que la mayoría de los datos FP del Spice son de doble precisión, por lo que requiere dos registros FP por dato. Igual que en la Figura 3.28, unos siete registros enteros capturan la mitad de las referencias enteras, pero sólo se necesitan unos cinco registros para capturar la mitad de las referencias FP.

se debe probablemente a la ausencia de punteros en FORTRAN que hace más efectiva la ubicación de registros para Spice que para programas en C (por ejemplo, GCC y TeX). En segundo lugar, la cantidad de tráfico de «guardar/restaurar» es mucho menor. Además de estas diferencias, podemos ver que se usan menos registros para capturar las referencias ubicables de punto flotante (FP). Esto probablemente es debido a que un menor porcentaje de referencias de FP son ubicables, ya que la mayoría son para arrays.

Nuestra segunda pregunta está relacionada con la forma que un optimizador afecta a la mezcla de instrucciones ejecutadas. Las Figuras 3.30 y 3.31 contestan esta pregunta para los benchmarks utilizados aquí. Los datos se tomaron en máquinas de carga/almacenamiento utilizando una optimización global completa que incluye todas las optimizaciones locales y globales que aparecen en la Figura 3.26. Las diferencias entre los códigos, optimizando y no optimizando, se muestran en términos absolutos y relativos. El efecto más obvio de la optimización —además de disminuir la cuenta total de instrucciones— es incrementar la frecuencia relativa de los saltos, al disminuir el número de referencias a memoria y operaciones de la ALU más rápidamente que el número de saltos (que sólo disminuye ligeramente). En la sección de falacias y pifias presentamos un ejemplo, para ver cómo difieren en un VAX los códigos optimizado y no optimizado.

Finalmente, ¿con qué frecuencia se utilizan las diversas estructuras de control? Estos números son importantes, ya que los saltos están entre las restricciones más difíciles de hacer más rápidas y son muy difíciles de reducir con el compilador. Los datos de las Figuras 3.30 y 3.31 nos dan una buena idea de la frecuencia de saltos —se ejecutan de 6,5 a 18 instrucciones entre dos saltos o bifurcaciones (incluyendo la misma bifurcación por salto)—. Las llamadas a procedimientos se presentan con una frecuencia 12 ó 13 veces menor que los saltos, o en el rango de una vez entre cada 87 a 200 instrucciones para nuestros programas. Spice tiene el porcentaje más bajo de saltos y las menores llamadas a procedimientos por instrucción, aproximadamente en un factor de dos.

Cómo el arquitecto puede ayudar al escritor de compiladores

Hoy día, la complejidad de un compilador no proviene de traducir instrucciones sencillas como A = B + C. La mayoría de los programas son «localmente simples», y las traducciones simples funcionan bien. La complejidad surge, más bien, porque los programas son grandes y globalmente complejos en sus interacciones, y porque la estructura de los compiladores indica que deben tomarse decisiones, en cada paso, sobre la secuencia de código que es mejor.

Los escritores de compiladores trabajan, con frecuencia, bajo su propio corolario de un principio básico en arquitectura: «Hacer los casos frecuentes rápidos y el caso raro correcto.» Es decir, si conocemos los casos frecuentes y los raros, y si la generación de código para ambos es sencilla, entonces la calidad del código para el caso raro puede no ser muy importante —¡pero debe ser correcta!

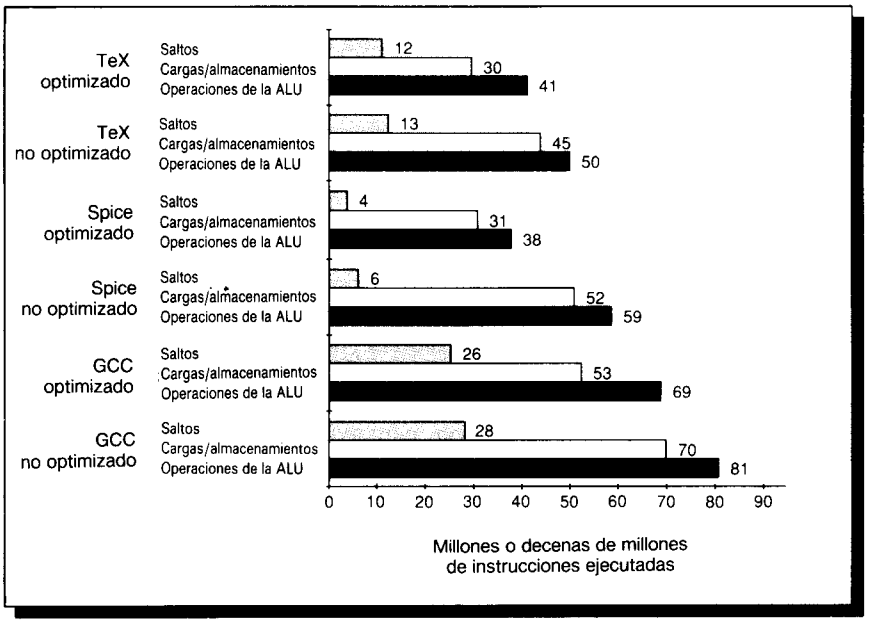


FIGURA 3.30 Los efectos de la optimización en recuentos absolutos de instrucciones. El eje x es el número de millones de instrucciones ejecutadas para GCC y TeX y de decenas de millones para Spice. Los programas no optimizados ejecutan el 21, 58 y 30 por 100 más instrucciones para GCC, Spice y TeX, respectivamente. Este dato se tomó en una DECstation 3100 utilizando optimización -O2, como muestran los datos de la Figura 3.31. Las optimizaciones que no afectan al recuento de instrucciones, pero pueden afectar al recuento de ciclos, no se miden aquí.

Algunas propiedades de los repertorios de instrucciones ayudan al escritor de compiladores. Estas propiedades no deberían ser consideradas como reglas rígidas y rápidas, sino más bien como pautas que harán más facil la escritura de un compilador que genere código eficiente y correcto.

1. *Regularidad.* Siempre que tenga sentido, los tres componentes principales de un repertorio de instrucciones —las operaciones, los tipos de datos y los modos de direccionamiento— deberían ser ortogonales. Dos aspectos de una arquitectura se dice que son *ortogonales* si son independientes. Por ejemplo, las operaciones y modos de direccionamiento son ortogonales si, para cada operación a la que puede aplicarse un cierto modo de direccionamiento, son aplicables todos los modos de direccionamiento. Esto ayuda a simplificar la generación de código, y es particularmente importante cuando la decisión sobre el código que se genera se divide en dos pasos en el compilador. Un buen contraejemplo de esta propiedad es restringir los registros que se pueden utilizar para un cierto tipo de instrucciones. Esto puede dar como consecuencia que el compilador tenga muchos registros disponibles, ¡pero ninguno del tipo correcto!

2. *Proporcionar primitivas, no soluciones.* Características especiales que «coincidan» con una construcción del lenguaje son inutilizables, con frecuencia. Intentos para soportar lenguajes de alto nivel pueden funcionar sólo con un lenguaje, o hacer más o menos lo que se requiere para una implementación correcta y eficiente del lenguaje. Algunos ejemplos de cómo han fallado estos intentos se dan en la Sección 3.9.

3. *Simplificar compromisos entre alternativas.* Uno de los trabajos más fuertes, que tiene el escritor de compiladores, es imaginar la secuencia de instrucciones que será mejor para cada segmento de código que se presente. En los primeros tiempos, el número de instrucciones o el tamaño total del código podían haber sido buenas métricas, pero —como vimos en el último capítulo— esto no es completamente cierto. Con las caches y la segmentación, los compromisos han llegado a ser muy complejos. Cualquier cosa que el diseñador pueda hacer, para ayudar al escritor de compiladores a comprender los costos de secuencias alternativas de código, ayudará a mejorar el código. Una de las instancias más difíciles de compromisos complejos se presenta en las arquitecturas memoria-memoria, al decidir el número de veces que debe ser referenciada una variable antes de que sea más barato cargarla en un registro. Este umbral es difícil de calcular y, en efecto, puede variar entre modelos de la misma arquitectura.

4. *Proporcionar instrucciones que consideren las cantidades conocidas en tiempo de compilación como constantes.* El escritor de compiladores odia la

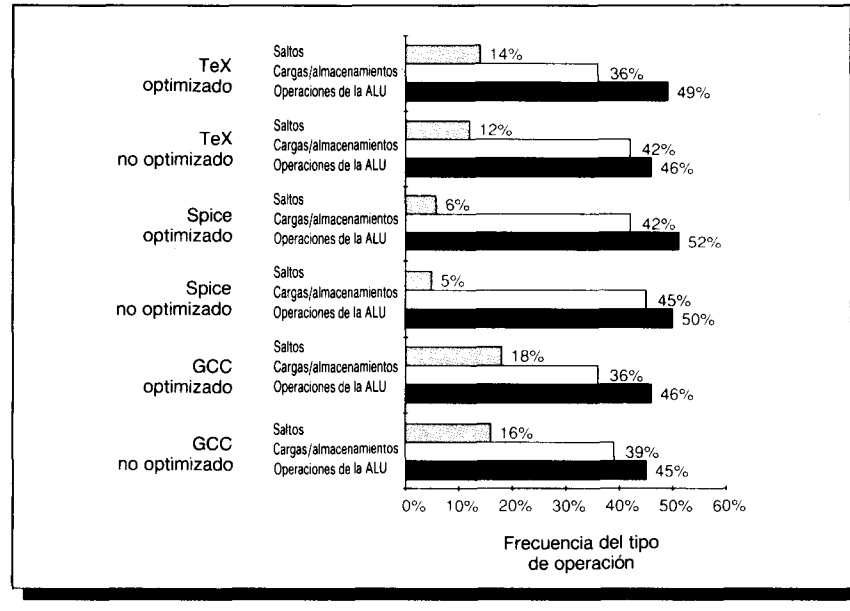


FIGURA 3.31 Los efectos de la optimización en la mezcla relativa de instrucciones para los datos de la Figura 3.30.

idea de que la máquina interprete, en tiempo de ejecución, un valor que fuese conocido en tiempo de compilación. Buenos contraejemplos de este principio son las instrucciones que interpretan valores fijados en tiempo de compilación. Por ejemplo, la instrucción de llamar a procedimiento del VAX (CALLS) interpreta dinámicamente una máscara que indica los registros que hay que guardar en una llamada, pero la máscara está fijada en tiempo de compilación, aunque en algunos casos pueda no ser conocida, por el llamador, si se utiliza compilación separada.

3.8 Juntando todo: cómo los programas utilizan los repertorios de instrucciones

¿Qué hacen los programas normales? Esta sección investigará y comparará el comportamiento de nuestros programas de benchmark ejecutados en una arquitectura de carga/almacenamiento y en una arquitectura memoria-memoria. La tecnología de compiladores es diferente para estas dos arquitecturas distintas y estas diferencias afectan a las medidas globales.

Podemos examinar el comportamiento de los programas normales, observando la frecuencia de tres operaciones básicas: referencias a memoria, operaciones de la ALU, e instrucciones de flujo de control (saltos y bifurcaciones). La Figura 3.32 hace esto para una arquitectura de carga/almacenamiento con un modo de direccionamiento (una máquina hipotética llamada DLX que definimos en el capítulo siguiente), y para una arquitectura memoria-memoria con muchos modos de direccionamiento (el VAX). La arquitectura de carga/almacenamiento tiene más registros, y su compilador pone más énfasis en la reducción del tráfico de memoria. Considerando las enormes diferencias en los repertorios de instrucciones de estas dos máquinas, los resultados son bastante similares.

Las mismas máquinas y programas se utilizan en la Figura 3.33, pero los datos representan recuentos absolutos de instrucciones ejecutadas, palabras de instrucción y referencias a datos. Este diagrama muestra una clara diferencia en el recuento de instrucciones: la máquina de carga/almacenamiento requiere más instrucciones. Recordar que esta diferencia no implica nada con respecto al rendimiento de las máquinas basadas en estas arquitecturas.

Este diagrama también muestra el número de referencias a datos hechas por cada máquina. A partir de los datos de la Figura 3.32 y de los recuentos de instrucciones, podríamos suponer que el número total de accesos a memoria, hechos por la máquina de memoria-memoria, será mucho menor que el realizado por la máquina de carga/almacenamiento. Pero los datos de la Figura 3.33 indican que esta hipótesis es falsa. La gran diferencia en las referencias a los datos equilibra la diferencia en las referencias a las instrucciones entre las arquitecturas, de modo que la máquina de carga/almacenamiento utiliza, aproximadamente, el mismo ancho de banda de memoria a nivel de arquitectura. Esta diferencia en las referencias a los datos surge probablemente porque la máquina de carga/almacenamiento tiene muchos registros, y su compilador hace una mejor tarea de ubicación de registros. Para ubicar

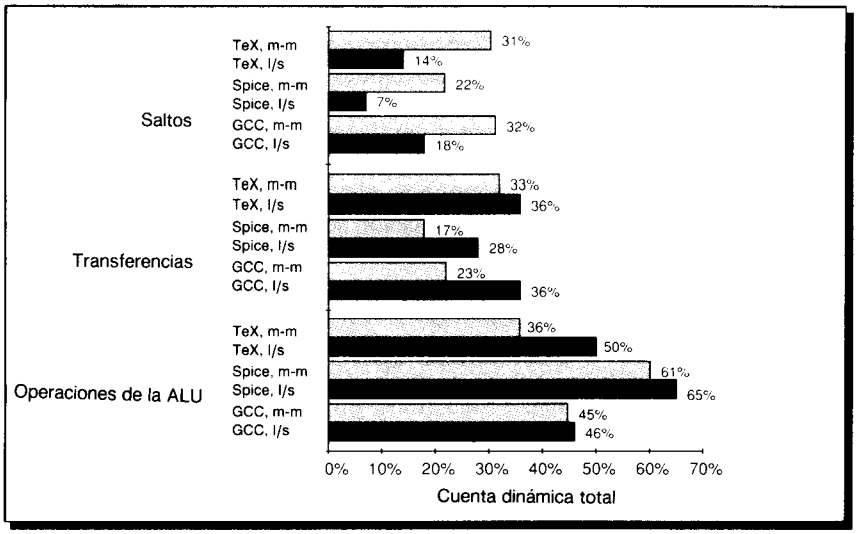


FIGURA 3.32 Las distribuciones de instrucciones para nuestros benchmarks difieren de manera clara cuando corren en una arquitectura de carga/almacenamiento (l/s) y en una arquitectura memoria-memoria (m-m). En la máquina de carga/almacenamiento, los movimientos de datos son cargas o almacenamientos. En la máquina memoria-memoria los movimientos de datos incluyen transferencias entre dos posiciones; que según los operandos pueden ser registros o posiciones de memoria. Sin embargo, la mayoría de los movimientos de datos involucran un registro y una posición de memoria. La máquina de carga/almacenamiento muestra un mayor porcentaje de movimientos de datos ya que es una máquina de carga/almacenamiento —para operar con los datos deben transferirse a los registros—. La frecuencia relativa más baja de los saltos principalmente es una función del uso de más instrucciones de las otras dos clases en la máquina de carga/almacenamiento. Estos datos se midieron con optimización en un VAXstation 3100 para la máquina memoria-memoria y en DLX, que explicamos con detalle en el siguiente capítulo, para la máquina de carga/almacenamiento. La entrada usada es más pequeña que la del Capítulo 2 para hacer posible la recolección de datos en el VAX.

cantidades enteras, la máquina de carga/almacenamiento tiene disponibles más del doble de registros. En total, para variables enteras y de punto flotante están disponibles el cuádruple de registros, para que los utilice el compilador en la arquitectura de carga/almacenamiento. Este salto en el número de registros, combinado con las diferencias del compilador, es la base más probable para la diferencia en el ancho de banda de los datos.

Hemos visto cómo las medidas arquitectónicas pueden ir en contra de la intuición del diseñador, y cómo algunas de estas medidas no afectan directamente al rendimiento. En la siguiente sección veremos que los intentos del arquitecto para modelar máquinas directamente a partir de las características software de alto nivel pueden fracasar.

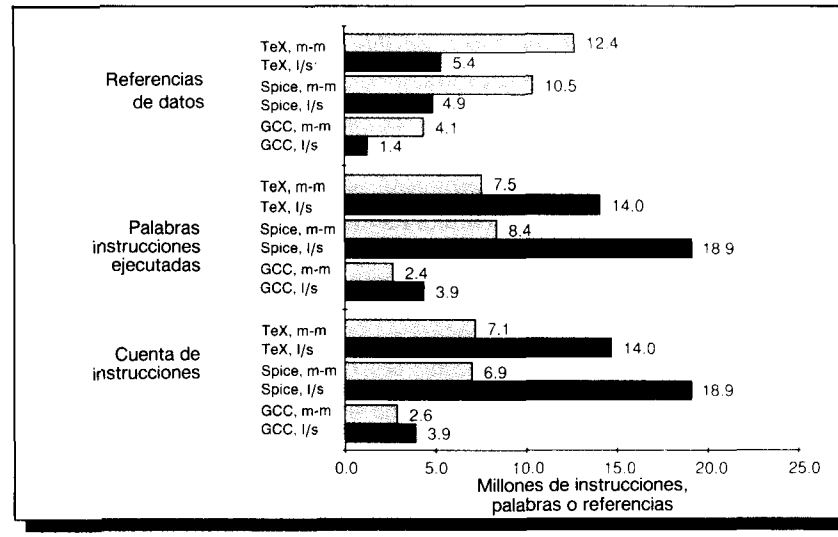


FIGURA 3.33 Recuentos absolutos para eventos dinámicos en una máquina de carga/almacenamiento y memoria-memoria. Los recuentos son (desde abajo arriba) instrucciones dinámicas, palabras de instrucción (bytes de instrucción divididos por cuatro), y referencias a datos (éstas pueden ser byte, palabra o doble palabra). Cada referencia se cuenta una vez. Diferencias en el tamaño del conjunto de registros y del compilador probablemente explican la gran diferencia en el número de referencias a datos. En el caso de Spice, la gran diferencia en el número total de registros disponibles para ubicación probablemente es la razón básica para la gran diferencia de los accesos totales a los datos. Estos datos se consiguieron con los mismos programas, entradas y máquinas que los datos de la Figura 3.32.

3.9 Falacias y pifias

A lo largo del tiempo los arquitectos han caído en creencias comunes, pero erróneas. En esta sección examinaremos algunas de ellas.

Pifia: Diseñar una característica del repertorio de instrucciones de «alto nivel» orientada específicamente para soportar una estructura de lenguaje de alto nivel.

Intentos de incorporar características de lenguajes de alto nivel en el repertorio de instrucciones han llevado a los arquitectos a suministrar instrucciones potentes con un amplio rango de flexibilidad. Pero, a menudo, estas instrucciones hacen más trabajo que el que se requiere en el caso frecuente, o no cumplen exactamente los requerimientos del lenguaje. Muchos de estos esfuerzos han estado encaminados a eliminar lo que en los años setenta se denominaba «desnivel semántico» («semantic gap»). Aunque la idea es complementar el repertorio de instrucciones con adiciones que acerquen el hardware al nivel del lenguaje, las adiciones pueden generar lo que Wulf [1981] ha denominado «choque semántico» *(semantic clash)*:

...al dar demasiado contenido semántico a la instrucción, el diseñador de máquinas hizo posible utilizar la instrucción solamente en contextos limitados. [p. 43]

A menudo muchas instrucciones son sencillamente excesivas —son muy generales para el caso más frecuente, conduciendo a un trabajo innecesario y a una instrucción más lenta. De nuevo, la instrucción CALLS de VAX es un buen ejemplo. La CALLS utiliza una estrategia de guardar-llamado *(callee-save)* (los registros que se van a guardar se especifican en el procedimiento llamado) **pero** el almacenamiento lo realiza la instrucción «call» del procedimiento llamador. La instrucción CALLS comienza con los argumentos introducidos en la pila, y después realiza los siguientes pasos:

1. Alinear la pila si es necesario.
2. Introducir el número de argumentos en la pila.
3. Guardar los registros indicados por la máscara de llamada del procedimiento en la pila (como se mencionó en la Sección 3.7). La máscara se encuentra en el código del procedimiento llamado —esto permite que el almacenamiento lo realice el procedimiento llamador aun con compilación separada.
4. Introducir la dirección de retorno en la pila, después, introducir el tope y base de los punteros de pila para el registro de activación.
5. Borrar los códigos de condición, que inicializan la habilitación de traps a un estado conocido.
6. Introducir en la pila una palabra para información sobre el estado y una palabra cero.
7. Actualizar los dos punteros de pila.
8. Saltar a la primera instrucción del procedimiento.

La inmensa mayoría de llamadas en programas reales no necesitan tanto gasto. La mayoría de los procedimientos conocen su número de argumentos y se puede establecer un convenio de enlace mucho más rápido, utilizando en lugar de la pila, registros que pasen argumentos. Además, la instrucción de llamada *(call)* fuerza a que se utilicen dos registros para enlace, mientras que muchos lenguajes solamente necesitan uno. Han fallado muchos intentos para soportar las llamadas a procedimientos y la gestión de la activación de pila, bien porque no coinciden con las necesidades del lenguaje o porque son muy generales y, por tanto, muy caros de utilizar.

Los diseñadores del VAX proporcionaron una instrucción más simple, JSB, que es mucho más rápida, ya que solamente introduce el PC de vuelta en la pila y bifurca al procedimiento (ver Ejercicio 3.11). Sin embargo, la mayoría de los compiladores VAX utilizan las instrucciones CALLS más costosas. Las instrucciones de llamada se incluyeron en la arquitectura, para estandarizar el convenio de enlace de procedimientos. Otras máquinas han estandarizado sus convenios de llamada, por acuerdo entre los escritores de compiladores y, sin

la necesidad de gastar una instrucción de llamada de procedimiento muy general y compleja.

Falacia: No cuesta nada proporcionar un nivel de funcionalidad que exceda el que se requiere en el caso habitual.

Una pifia arquitectónica más seria que la anterior se encontró en algunas máquinas, como el Intel 432, que proporcionaban solamente una instrucción de llamada de alto coste que manipulaba los casos más raros. La instrucción de llamada en el Intel 432 siempre crea un nuevo contexto protegido y, por tanto, es más costosa (ver Capítulo 8 para una discusión adicional sobre protección de memoria). Sin embargo, la mayoría de las llamadas están en el mismo módulo y no necesitan una llamada protegida. Si se dispusiese de un mecanismo de llamada más simple y se utilizase cuando fuese posible, Dhrystone correría el 20 por 100 más rápido en el 432 (ver Colwell y cols. [1985]). Cuando los arquitectos eligen solamente una instrucción general y cara, los escritores de compiladores no tienen otra elección que utilizar la instrucción costosa, y sufrir el gasto innecesario. En la sección histórica del Capítulo 8 se da una explicación de la experiencia de los diseñadores para proporcionar, en hardware, dominios de protección de grano fino. La discusión posterior ilustra esta falacia.

Pifia: Utilizar un compilador que no optimice, para medir la utilización del repertorio de instrucciones realizada por un compilador optimizador.

La utilización del repertorio de instrucciones por un compilador que optimice puede ser bastante diferente de la que haga otro que no optimice. Vimos algunos ejemplos en la Figura 3.31. La Figura 3.34 muestra las diferencias en el

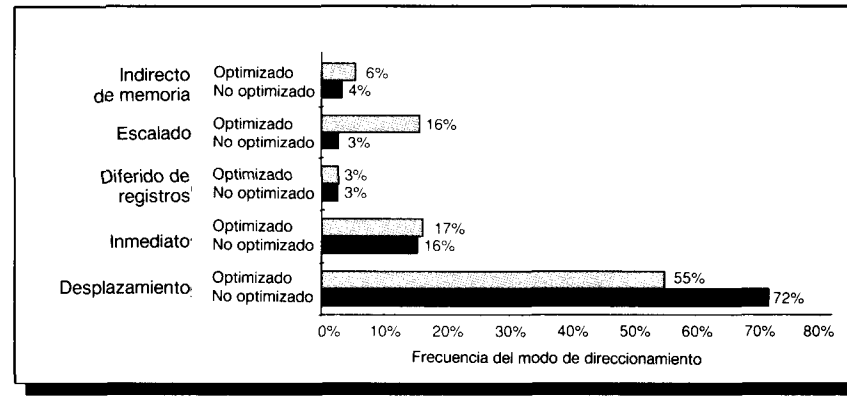


FIGURA 3.34 La utilización de los modos de direccionamiento por un compilador optimizador y no optimizador puede diferir significativamente. Estas medidas muestran el uso de los modos de direccionamiento VAX por Spice cuando se compila utilizando un compilador no optimizador (f77) y un compilador optimizador *(fort)*. En particular, el uso del modo escalado es mucho mayor para el compilador optimizador. Los demás modos de direccionamiento VAX contabilizan el restante 2-3 por 100 de las referencias a memoria de datos.

uso de los modos de direccionamiento en un VAX para Spice, cuando se compila con el compilador no optimizador UNIX F77 y cuando se compila con el compilador optimizador FORTRAN DEC.

3.10 Observaciones finales

Las primeras arquitecturas estaban limitadas en sus repertorios de instrucciones por la tecnología hardware de la época. Tan pronto como lo permitió la tecnología hardware, los arquitectos comenzaron a buscar formas de soportar lenguajes de alto nivel. Esta búsqueda condujo a tres períodos distintos de pensamiento sobre cómo soportar los programas eficientemente. En los años sesenta, las arquitecturas de pila se hicieron populares. Se consideraron como un buen acierto para los lenguajes de alto nivel —y probablemente lo fueron, dada la tecnología de compiladores de entonces—. En los años setenta, el objetivo principal de los arquitectos fue reducir los costos del software. Este objetivo se consiguió principalmente sustituyendo el software por el hardware, o proporcionando arquitecturas de alto nivel que podían simplificar la tarea de los diseñadores de software. El resultado fue el cambio hacia las arquitecturas de computadores de lenguajes de alto nivel y potentes arquitecturas como el VAX, que tenían un gran número de modos de direccionamiento, múltiples tipos de datos y una arquitectura enormemente ortogonal. En los años ochenta, la tecnología de compiladores más sofisticada y un énfasis renovado en el rendimiento de la máquina han propiciado una vuelta a arquitecturas más simples, basadas principalmente en el estilo de la máquina de carga/almacenamiento. Los cambios continuos en la forma que se programa, la tecnología de compiladores que se usa y la tecnología hardware fundamental harán sin duda otra dirección más atractiva en el futuro.

3.11 Perspectiva histórica y referencias

Uno elevaría las cejas siempre que una arquitectura futura se desarrollase con un repertorio de instrucciones orientadas a registro o a pila.

Meyers [1978, 20]

Los primeros computadores, incluyendo las máquinas UNIVAC I, EDSAC e IAS, fueron máquinas basadas en acumulador. La simplicidad de este tipo de máquinas hizo la elección natural, cuando los recursos hardware eran muy limitados. La primera máquina de registros de propósito general fue la Pegasus, construida por Ferranti, Ltd. en 1956. La Pegasus tenía ocho registros de propósito general, R0 contenía siempre cero. Las transferencias de bloques cargaban los ocho registros del tambor.

En 1963, Burroughs presentó la B5000. La B5000 fue, quizá, la primera máquina que consideró seriamente el software y los compromisos hardware-

software. Barton y los diseñadores de Burroughs hicieron de la B5000 una arquitectura de pila (como se describe en Barton [1961]). Diseñada para soportar lenguajes de alto nivel como ALGOL, esta arquitectura de pila usaba un sistema operativo (MCP) escrito en un lenguaje de alto nivel. La B5000 también fue la primera máquina, de un fabricante de Estados Unidos, que soportaba memoria virtual. La B6500, introducida en 1968 (y explicada en Hauck y Dent [1968]), añadía la gestión por hardware de registros de activación. En ambas, la B5000 y la B6500, los dos elementos de la cabeza de la pila se conservaban en la CPU y el resto de la pila en memoria. La arquitectura de pila producía buena densidad de código, pero solamente proporcionaba dos posiciones de almacenamiento de alta velocidad. Los autores del artículo original del IBM 360 [Amdahl y cols., 1964] y el artículo original del PDP-11 [Bell y cols., 1970] argüían contra la organización de pilas. Citaban tres puntos principales en sus argumentos contra las pilas:

1. El rendimiento se deriva de los registros rápidos, no de la forma que son utilizados.
2. La organización de pila es muy limitada, y requiere muchas operaciones de intercambio y copia.
3. La pila tiene un extremo inferior, y cuando se coloca en la memoria más lenta hay pérdida de rendimiento.

Las máquinas basadas en pila dejaron de ser populares a finales de los setenta y desaparecieron, básicamente, en los años ochenta.

El término «arquitectura de computadores» fue acuñado por IBM en 1964 para utilizarlo con el IBM 360. Amdahl, Blaauw y Brooks [1964] utilizaron el término para referenciar la parte del repertorio de instrucciones visible por el programador. Creían que una familia de máquinas de la misma arquitectura deberían poder correr el mismo software. Aunque esta idea pueda parecer obvia hoy día, fue bastante novedosa en esa época. IBM, aunque era la compañía líder en la industria, tenía **cinco** arquitecturas diferentes antes del 360. Por tanto, la noción de una estandarización de la compañía en una sola arquitectura fue radical. Los diseñadores del 360 esperaban que seis divisiones diferentes de IBM podrían trabajar juntas para definir una arquitectura común. Su definición de arquitectura era

...la estructura de un computador que un programador en lenguaje máquina debe comprender para escribir un programa correcto (independiente del tiempo) para esa máquina.

El término «programador en lenguaje máquina» significaba que podía mantenerse la compatibilidad aun en lenguaje ensamblador, mientras que «independiente del tiempo» permitía diferentes implementaciones.

El IBM 360 fue la primera máquina vendida en grandes cantidades con direccionamiento de bytes, utilizando bytes de 8 bits y registros de propósito general. El 360 también tenía instrucciones de registro-memoria e instrucciones limitadas de memoria-memoria.

En 1964, Control Data proporcionó el primer supercomputador, el CDC 6600. Como se explica en Thornton [1964], él, Cray y otros diseñadores del

6600 fueron los primeros en explorar en profundidad la segmentación *(pipelining)*. El 6600 fue la primera máquina de carga/almacenamiento de propósito general. En los años sesenta, los diseñadores del 6600 se dieron cuenta de la necesidad de simplificar la arquitectura con el fin de lograr una segmentación eficiente. Esta interacción entre simplicidad de la arquitectura e implementación fue muy despreciada durante los años setenta por los diseñadores de microprocesadores y minicomputadores, pero fue tenida en cuenta en los ochenta.

Al final de los años sesenta y principio de los setenta, la gente se daba cuenta que los costos del software crecían con más rapidez que los del hardware. McKeeman [1967] argüía que los compiladores y sistemas operativos tendían a ser muy grandes y muy complejos y llevaba demasiado tiempo desarrollarlos. Debido a los compiladores inferiores y a las limitaciones de memoria de las máquinas, la mayoría de los programas de los sistemas en esa época se escribían todavía en lenguaje ensamblador. Muchos investigadores propusieron aliviar la crisis del software creando arquitecturas más potentes orientadas al software. Tanenbaum [1978] estudió las propiedades de los lenguajes de alto nivel. Con otros investigadores, encontró que la mayoría de los programas son sencillos. Entonces arguyó que las arquitecturas debían ser diseñadas teniendo esto en cuenta, y que se debería optimizar el tamaño de los programas y facilitar la compilación. Tanenbaum propuso una máquina de pila con formatos de instrucciones codificados por frecuencia, para conseguir estos objetivos. Sin embargo, como hemos observado, el tamaño del programa no se traduce directamente en coste/rendimiento, y las máquinas de pila desaparecieron lentamente después de este trabajo.

El artículo de Strecker [1978] explica cómo él y los demás arquitectos de DEC respondieron a esto diseñando la arquitectura VAX. El VAX fue diseñado para simplificar la compilación de los lenguajes de alto nivel. Los escritores de compiladores se habían quejado de la falta de completa ortogonalidad del PDP-11. La arquitectura VAX se diseñó para que fuese altamente ortogonal y permitiese la correspondencia entre una instrucción del lenguaje de alto nivel con una simple instrucción VAX. Adicionalmente, los diseñadores de VAX intentaron optimizar el tamaño del código, debido a que los programas compilados, con frecuencia, eran muy grandes para las memorias disponibles. Cuando se introdujo esto en 1978, el VAX fue la primera máquina con una verdadera arquitectura memoria-memoria.

Mientras se estaba diseñando el VAX, una aproximación más radical, llamada *Arquitectura de Computadores de Lenguajes de Alto Nivel* (HLLCA), se estaba recomendando en la comunidad de investigadores. Este movimiento estaba encaminado a eliminar el desnivel existente entre los lenguajes de alto nivel y el hardware de computadores —que Gagliardi [1973] llamó «el desnivel semántico»— acercando el hardware «hasta» el nivel del lenguaje de programación. Meyers [1982] proporcionó un buen resumen de los argumentos y una historia de los proyectos de arquitectura de computadoras de lenguajes de alto nivel.

Smith, Rice y cols. [1971] explicaron el Proyecto SYMBOL que comenzaron en Fairchild. SYMBOL se convirtió en el mayor y más famoso de los intentos de HLLCA. Su objetivo fue construir una máquina de lenguaje de alto nivel y tiempo compartido, que redujera enormemente el tiempo de pro-

gramación. La máquina SYMBOL interpretaba directamente programas, escritos en su nuevo lenguaje de programación propio; el compilador y sistema operativo se construyeron en hardware. El lenguaje de programación era muy dinámico —no había declaraciones de variables porque el hardware interpretaba dinámicamente cada sentencia.

SYMBOL tuvo muchos problemas, los más importantes fueron inflexibilidad, complejidad y rendimiento. El hardware de SYMBOL incluía el lenguaje de programación, el sistema operativo e incluso el editor de textos. Los programadores no podían realizar ninguna elección en el lenguaje de programación que utilizaban, por ello no se pudieron incorporar los avances posteriores en sistemas operativos y lenguajes de programación. La máquina también era complicada para el diseño y depuración. Debido a que el hardware se utilizaba para casi todo, los casos raros y complejos necesitaban ser manipulados completamente en hardware, así como los casos comunes más simples.

Ditzel [1980] observó que SYMBOL tenía enormes problemas de rendimiento. Aunque los casos exóticos corrían relativamente rápidos, los casos comunes y sencillos con frecuencia corrían lentamente. Se necesitaban muchas referencias a memoria para interpretar una sencilla sentencia de un programa. Aunque el objetivo de eliminar el desnivel semántico parecía loable, cualquiera de los tres problemas planteados por SYMBOL habría sido suficiente para condenar este planteamiento.

HLLCA nunca tuvo un impacto comercial significativo. El incremento en el tamaño de memoria de las máquinas y la utilización de memoria virtual eliminaban los problemas de tamaño de código, que surgían en los lenguajes de alto nivel y sistemas operativos escritos en lenguajes de alto nivel. La combinación de arquitecturas más simples junto con el software ofrecían grandes rendimientos y más flexibilidad a bajo costo y menor complejidad.

Los estudios sobre la utilización de repertorios de instrucciones comenzaron a finales de los años cincuenta. La mezcla de Gibson, descrita en el último capítulo, procede de un estudio de utilización de instrucciones en el IBM 7090. En los años setenta había varios estudios sobre el uso de los repertorios de instrucciones. Entre los mejor conocidos están el de Foster y cols. [1971] y el de Lunde [1977]. La mayoría de estos estudios pioneros empleaban pequeños programas porque las técnicas utilizadas para coleccionar datos eran caras. Empezando a finales de los años setenta, el área de medida y análisis de los repertorios de instrucciones se hizo muy activa. Debido a que utilizamos datos de la mayor parte de estos artículos en el capítulo siguiente, revisaremos allí las contribuciones.

Otros estudios realizados en los años setenta examinaron la utilización de las características de los lenguajes de programación. Aunque muchos de éstos estudiaron solamente propiedades estáticas, los artículos de Alexander y Wortman [1975] y Elshoff [1976] estudiaron las propiedades dinámicas de los programas HLL. El interés en la utilización por parte de los compiladores de los repertorios de instrucciones, y de la interacción entre compiladores y arquitecturas creció en los años ochenta. Se creó una conferencia para tratar la interacción entre los sistemas software y hardware denominada Soporte Arquitectónico para Lenguajes de Programación y Sistemas Operativos *(Architectural Support for Programming Languages and Operating Systems)* (AS-

PLOS). En esta conferencia bianual, se han publicado muchos artículos sobre medidas de los repertorios de instrucciones e interacción entre compiladores y arquitecturas.

A principios de los años ochenta, la dirección de las arquitecturas comenzó a dejar de proporcionar soporte hardware de alto nivel para los lenguajes. Ditzel y Patterson [1980] analizaron las dificultades encontradas por las arquitecturas de los lenguajes de alto nivel y arguyeron que la respuesta yacía en arquitecturas más simples. En otro artículo [Patterson y Ditzel, 1980], estos autores discutieron primero la idea de computadores de repertorio de instrucciones reducido (RISC) y presentaron argumentos para arquitecturas más simples. Su propuesta fue refutada por Clark y Strecker [1980]. Hablaremos más sobre los efectos de estas ideas en el capítulo siguiente.

Al mismo tiempo, otros investigadores publicaban artículos que defendían un acoplamiento más cercano de arquitecturas y compiladores, en lugar de intentos de complementar los compiladores. Entre éstos se incluyen Wulf [1981] y Hennessy y cols. [1982].

La primitiva tecnología de compiladores desarrollada para FORTRAN fue bastante buena. Muchas de las técnicas de optimización que se utilizan en los compiladores actuales fueron desarrolladas e implementadas a finales de los sesenta o principios de los setenta (ver Cocke y Schwartz [1970]). Debido a que FORTRAN tenía que competir con el lenguaje ensamblador, hubo una tremenda presión para lograr eficiencia en los compiladores FORTRAN. Sin embargo, una vez que los beneficios de la programación en HLL fueron obvios, se desplazó la atención de las tecnologías de optimización. La mayor parte del trabajo de optimización de los años setenta tuvo una orientación más teórica que experimental. A principios de los años ochenta, hubo un nuevo enfoque en el desarrollo de compiladores optimizadores. Cuando se estabilizó esta tecnología, algunos investigadores escribieron artículos examinando el impacto de diversas optimizaciones de los compiladores sobre el tiempo de ejecución de los programas. Cocke y Markstein [1980] describen las medidas utilizando el compilador IBM PL.8; Chow [1983] describe las ganancias obtenidas con el compilador UCODE de Stanford en diversas máquinas. Como hemos visto en este capítulo, la ubicación de registros es la piedra angular de la optimización de los modernos compiladores. La formulación de la ubicación de registros, como un problema de coloración de grafos, la realizaron originalmente Chaitin y cols. [1982]. Chow y Hennessy [1984, 1990] extendieron el algoritmo para utilizar prioridades al escoger las cantidades a ubicar. El progreso en la optimización y ubicación de registros ha llevado a un uso más amplio de los compiladores optimizadores, y el impacto de las tecnologías de compiladores sobre los compromisos arquitectónicos ha aumentado considerablemente en la pasada década.

Referencias

ALEXANDER, W. G. AND D. B. WORTMAN [1975]. «Static and dynamic characteristics of XPL programs», *Computer* 8:11 (November) 41-46.

AMDAHL, G. M., G. A. BLAAUW, AND F. P. BROOKS, JR. [1964]. «Architecture of the IBM System 360», *IBM J. Research and Development* 8:2 (April) 87-101.

BARTON, R. S. [1961]. «A new approach to the functional design of a computer», *Proc. Western Joint Computer Conf.*, 393-396.

BELL, G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. «A new architecture for mini-computers: The DEC PDP-11», *Proc. AFIPS SJCC*, 657-675.

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1982]. «Register allocation via coloring», *Computer Languages* 6, 47-57.

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Desing and Measurements*, Ph. D. Thesis, Stanford Univ. (December).

CHOW, F. C. AND J. L. HENNESSY [1984]. «Register allocation by priority-based coloring», *Proc. SIGPLAN '84 Compiler Construction (ACM SIGPLAN Notices 19:6* June) 222-232.

CHOW, F. C. AND J. L. HENNESSY [1990]. «The Priority-Based Coloring Approach to Register Allocation», *ACM Trans. on Programming Languages and Systems* 12:4 (October).

CLARK, D. AND W. D. STRECKER [1980]. «Comments on "the case for the reduced instruction set computer"», *Computer Architecture News* 8:6 (October) 34-38.

COCKE, J., AND J. MARKSTEIN [1980]. «Measurement of code improvement algorithms», *Information Processing* 80, 221-228.

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers*, Courant Institute, New York Univ., New York City.

COHEN, D. [1981]. «On holy wars and a plea for peace», *Computer* 14:10 (October) 48-54.

COLWELL, R. P., C. Y. HITCHCOCK, III, E. D. JENSEN, H. M. B. SPRUNT, AND C. P. KOLLAR, [1985]. «Computers, complexity, and controversy», *Computer* 18:9 (September) 8-19.

DITZEL, D. R. [1981]. «Reflections on the high-level language Symbol computer system», *Computer* 14:7 (July) 55-66.

DITZEL, D. R. AND D. A. PATTERSON [1980]. «Retrospective on high-level language computer architecture», in *Proc. Seventh Annual Symposium on Computer Architecture*, La Baule, France (June) 97-104.

ELSHOFF, J. L. [1976]. «An analysis of some commercial PL/I programs», *IEEE Trans. on Software Engineering* SE-2 2 (June) 113-120.

FOSTER, C. C., R. H. GONTER, AND E. M. RISEMAN [1971]. «Measures of opcode utilization», *IEEE Trans. on Computers* 13:5 (May) 582-584.

GAGLIARDI, U. O. [1973]. «Report of workshop 4-software-related advances in computer hardware», *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99-120.

HAUCK, E. A., AND B. A. DENT [1968]. «Burroughs' B6500/B7500 stack mechanism», *Proc. AFIPS SJCC*, 245-251.

HENNESSY, J. L., N. JOUPPI, F. BASKETT, T. R. GROSS, AND J. GILL [1982]. «Hardware/software tradeoffs for increased performance», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), 2-11.

LUNDE, A. [1977]. «Empirical evaluation of some features of instruction set processor architecture», *Comm. ACM* 20:3 (March) 143-152.

MCKEEMAN, W. M. [1967]. «Language directed computer design», *Proc. 1967 Fall Joint Computer Conf.*, Washington, D.C., 413-417.

MEYERS, G. J. [1978]. «The evaluation of expressions in a storage-to-storage architecture», *Computer Architecture News* 7:3 (October), 20-23.

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, N.Y.

PATTERSON, D. A. AND D. R. DITZEL [1980]. «The case for the reduce instruction set computer», *Computer Architecture News* 8:6 (October) 25-33.

SMITH, W. R., R. R. RICE, G. D. CHESLEY, T. A. LALIOTIS, S. F. LUNDSTROM, M. A. CHALHOUN, L. D. GEROULD, AND T. C. COOK [1971]. «SYMBOL: A large experimental system exploring major hardware replacement of software», *Proc AFIPS Spring Joint Computer Conf.*, 601-616.

STRECKER, W. D. [1978]. «VAX-11/780: A virtual address extension of the PDP-11 family», *Proc. AFIPS National Computer Conf.* 47, 967-980.

TANENBAUM, A. S. [1978], «Implications of structured programming for machine architecture», *Comm. ACM* 21:3 (March) 237-246.

THORNTON, J. E. [1964]. «Parallel operation in Control Data 6600», *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33-40.

WILKES, M. V. [1982]. «Hardware support for memory protection: Capability implementations», *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems* (March) 107-116.

WILKES, M. V. AND W. RENWICK [1949]. *Report of a Conf. on High Speed Automatic Calculating Machines*, Cambridge, England.
WULF, W. [1981]. «Compilers and computer architecture», *Computer* 14:7 (July) 41-47.

EJERCICIOS

3.1 [15/10] <3.7> Para este problema utilizar los datos de las Figuras 3.30 y 3.31 para GCC, suponer los siguientes CPI:

Operación ALU	1
Carga/almacenamiento	3
Salto	5

a) [15] Calcular el CPI para las versiones optimizada y no optimizada de GCC.

b) [10] ¿Cuántas veces es más rápido el programa optimizado que el no optimizado?

3.2 [15/15/10] <3.8> En este problema utilizar los datos de la Figura 3.33. Suponer que cada palabra de instrucción y cada referencia a un dato requieren un acceso a memoria.

a) [15] Determinar el porcentaje de accesos a memoria de las instrucciones para cada uno de los tres benchmarks en la máquina de carga/almacenamiento.

b) [15] Determinar el porcentaje de accesos a memoria de las instrucciones para cada uno de los tres benchmarks en la máquina memoria-memoria.

c) [10] ¿Cuál es la razón de accesos totales a memoria en la máquina de carga/almacenamiento frente a la máquina memoria-memoria para cada benchmark?

3.3 [20/15/10] <3.3,3.8> Estamos diseñando formatos de repertorios de instrucciones para una arquitectura de carga/almacenamiento e intentamos decidir si es útil tener múltiples longitudes de desplazamiento para los saltos y referencias a memoria. Utilizando medidas de los tres benchmarks, hemos decidido que los desplazamientos sean iguales para estas dos clases de instrucciones. La longitud de una instrucción será igual a 16 bits + longitud del desplazamiento en bits. Las instrucciones de la ALU serán de 16 bits. La Figura 3.35 contiene los datos de las Figuras 3.13 y 3.19 promediados y puestos en forma acumulativa. Suponer que se necesita un bit adicional para el signo del desplazamiento.

Para las frecuencias del repertorio de instrucciones, utilizar los datos de la media de los tres benchmarks para la máquina de carga/almacenamiento de la Figura 3.32.

a) [20] Suponer que se permiten desplazamientos de 0, 8 o 16 bits de longitud incluyendo el bit de signo. Basándose en las estadísticas dinámicas de la Figura 3.32, ¿cuál es la longitud media de una instrucción ejecutada?

b) [15] Suponer que queremos instrucciones de longitud fija y seleccionamos para todas las instrucciones una longitud de 24 bits (incluyendo las instrucciones de la ALU). Para cada desplazamiento mayor de 8 bits, se necesita una instrucción adicional. Determinar el número de bytes de las instrucciones buscadas en esta máquina con tamaño de instrucción fijo frente a los buscados con una instrucción de tamaño variable.

Bits de desplazamiento	Referencias acumulativas de datos	Saltos acumulativos
0	16 %	0 %
1	16 %	0 %
2	21 %	10 %
3	29 %	27 %
4	32 %	47 %
5	44 %	66 %
6	55 %	79 %
7	62 %	89 %
8	66 %	94 %
9	68 %	97 %
10	73 %	99 %
11	78 %	100 %
12	80 %	100 %
13	86 %	100 %
14	87 %	100 %
15	100 %	100 %

FIGURA 3.35 Las columnas segunda y tercera contienen el porcentaje acumulativo de las referencias a datos y saltos, respectivamente, que pueden ser acomodadas en el número correspondiente de bits de magnitud del desplazamiento. Estos datos se obtienen al promediar y acumular los datos de las Figuras 3.13 y 3.19.

c) [10] ¿Qué ocurriría si la longitud del desplazamiento fuese 16 y y no se necesitase nunca una instrucción adicional? ¿Cómo se compararían los bytes de las instrucciones buscadas al elegir sólo un desplazamiento de 8 bits? Suponer que las instrucciones de la ALU son de 16 bits.

3.4 [15/10] <3.2> Algunos investigadores han sugerido que puede ser útil añadir un modo de direccionamiento registro-memoria a una máquina de carga/almacenamiento. La idea es sustituir secuencias de

```
LOAD    R1,0(Rb)
ADD     R2,R2,R1
```

por

```
ADD     R2,0(Rb)
```

Suponer que la nueva instrucción hace que el ciclo de reloj se incremente en un 10 por 100. Utilizar las frecuencias de instrucciones para el benchmark GCC en la máquina de carga/almacenamiento de la Figura 3.32 y suponer que los dos tercios de los movimientos son cargas y el resto almacenamientos. La nueva instrucción afecta solamente a la velocidad del reloj y no al CPI.

a) [15] ¿Qué porcentaje de las instrucciones de carga debe eliminarse de la máquina con la nueva instrucción para que como mínimo tenga el mismo rendimiento?

b) [12] Mostrar una situación en una secuencia de múltiples instrucciones donde una carga de R1 seguida inmediatamente de una utilización de R1 (con algún tipo de código de operación) no se pueda sustituir por una simple instrucción de la forma propuesta, suponiendo que existe el mismo código de operación.

3.5 [15/20] <3.1-3.3> En los dos apartados siguientes, hay que comparar la eficiencia de memoria de cuatro estilos diferentes de repertorios de instrucciones para dos secuencias de código. Los estilos de arquitectura son:

Acumulador

Memoria-Memoria. Los tres operandos de cada instrucción están en memoria.

Pila. Todas las operaciones se realizan en la cabeza de la pila. Solamente las instrucciones de introducir y sacar *(push y pop)* acceden a memoria, y las demás instrucciones eliminan sus operandos de la pila y los sustituyen por el resultado. La implementación utiliza una pila para las dos entradas de la cabeza; los accesos que utilizan las demás posiciones de la pila son referencias a memoria.

Carga/almacenamiento. Todas las operaciones se realizan en registros, y las instrucciones registro-registro tienen tres operandos por instrucción. Hay 16 registros de propósito general, y los especificadores de registro son de 4 bits.

Para medir la eficiencia de memoria, hacer las siguientes hipótesis sobre los cuatro repertorios de instrucciones:

- El código de operación es siempre de 1 byte (8 bits).
- Todas las direcciones de memoria son de 2 bytes (16 bits).
- Todos los operandos datos son de 4 bytes (32 bits).
- Todas las instrucciones tienen de longitud un número entero de bytes.

No hay más optimizaciones para reducir el tráfico de memoria, y las variables `A`, `B`, `C` y `D` están inicialmente en memoria.

Inventar los propios nemotécnicos para el lenguaje ensamblador, para escribir el mejor código en lenguaje ensamblador equivalente para los fragmentos de lenguaje de alto nivel dados.

a) [15] Escribir las cuatro secuencias de código para

```
A = B + C;
```

Para cada secuencia de código, calcular los bytes de las instrucciones buscadas y los bytes transferidos de la memoria de datos. ¿Qué arquitectura es más eficiente cuando se mide por tamaño de código? ¿Qué arquitectura es más eficiente cuando se mide el ancho de banda total de memoria requerida (código + datos)?

b) [20] Escribir las cuatro secuencias de código para

```
A = B + C;
B = A + C;
D = A - B;
```

Para cada secuencia de código, calcular los bytes de las instrucciones buscadas y los bytes transferidos (leídos o escritos) de la memoria de datos. ¿Qué arquitectura es más eficiente cuando se mide por tamaño de código? ¿Qué arquitectura es más eficiente cuando se mide el ancho de banda total de memoria requerida (código + datos)? Si las respuestas son diferentes de las de la parte a, ¿por qué son diferentes?

3.6 [20] <3.4> Suponer que los accesos de byte y media palabra requieren una red de alineación, como en la Figura 3.10. Algunas máquinas tienen solamente accesos de palabra, ya que la carga de un byte o media palabra emplea dos instrucciones (una carga y una extracción), y el almacenamiento parcial de una palabra emplea tres instrucciones (cargar, insertar, almacenar). Utilizar los datos del benchmark TeX de la Figura 3.23 para determinar el porcentaje de los accesos que son para bytes o medias palabras, y utilizar los datos de TeX en la máquina de carga/almacenamiento de la Figura 3.32 para calcular la frecuencia de las transferencias de datos. Suponer que las cargas son el doble de frecuentes que los almacenamientos, independientemente del tamaño de los datos. Si todas las instrucciones de la máquina emplean un ciclo, ¿qué incremento de la frecuencia de reloj debe obtenerse para hacer que la eliminación de accesos parciales sea una buena alternativa?

3.7 [20] <3.3> Tenemos una propuesta para tres máquinas diferentes: M_0, M_8 y M_{16}, que difieren en su cuenta de registros. Las tres máquinas tienen instrucciones de tres operandos, y cualquier operando puede ser una referencia a memoria o un registro. El coste de un operando de memoria en estas máquinas es de seis ciclos y el coste de un operando de registro es de un ciclo. Cada uno de los tres operandos tiene igual probabilidad de estar en un registro.

Las diferencias entre las máquinas se describen en la tabla siguiente. Los ciclos de ejecución por operación se suman al coste del acceso al operando. La probabilidad de que un operando esté en un registro se aplica individualmente a cada operando y está basada en las Figuras 3.28 y 3.29.

Máquina	Cuenta de registros	Ciclos de ejecución por operación ignorando accesos a operando	Probabilidad de que un operando esté en un registro en contraposición a memoria
M_0	0	4 ciclos	0,0
M_8	8	5 ciclos	0,5
M_{16}	16	6 ciclos	0,8

¿Cuál es el número de ciclos para una instrucción media en cada máquina?

3.8 [15/10/10] <3.1,3.7> Una forma en la que un arquitecto puede volver loco a un escritor de compiladores es hacerle difícil decir si una «optimización» del compilador puede ralentizar un programa en la máquina.

Considerar un acceso a A[*i*], donde A es un array de enteros e *i* es un desplazamiento entero en un registro. Deseamos generar código para utilizar el valor de A[*i*] como operando fuente en todo este problema. Suponer que todas las instrucciones emplean un ciclo de reloj más el costo del modo de direccionamiento de memoria.

- El direccionamiento indexado cuesta cuatro ciclos de reloj para la referencia a memoria (para un total de cinco ciclos de reloj por instrucción).
- El direccionamiento indirecto de registros cuesta tres ciclos de reloj para la referencia a memoria (para un total de cuatro ciclos de reloj).
- Las instrucciones de registro-registro no tienen coste de acceso a memoria, requiriendo solamente un ciclo.

Suponer que el valor A[*i*] debe almacenarse en memoria al final de la secuencia de código y que la dirección base de A ya está en R1 y el valor de *i* en R2.

a) [15] Suponer que el elemento del array A[*i*] no puede mantenerse en un registro, pero la dirección de A[*i*] puede estar en un registro una vez calculada. Por tanto, hay dos métodos diferentes de acceder a A[*i*]:

   1) calcular la dirección de A[*i*] en un registro y utilizar un registro indirecto, y
   2) Utilizar el modo de direccionamiento indexado.

   Escribir la secuencia de código para ambos métodos. ¿Cuántas referencias a A[*i*] deben presentarse para que el método 1 sea mejor?

b) [10] Suponer que se elige el método 1, pero que no se tienen registros y hay que guardar la dirección de A[*i*] en la pila y restaurarla. ¿Cuántas referencias hay que realizar ahora para que el método 1 sea mejor?

c) [10] Suponer que el valor A[*i*] puede estar en un registro (frente a la dirección de A[*i*]). ¿Cuántas referencias deben realizarse para que sea ésta la mejor aproximación en vez de utilizar el método 2?

3.9 [Discusión] <3.2-3.8> ¿Cuáles son los argumentos **económicos** (por ejemplo, más máquinas vendidas) **en pro y en contra** de cambiar la arquitectura a nivel lenguaje máquina?

3.10 [25] <3.1-3.3> Encontrar un manual del repertorio de instrucciones de alguna máquina antigua (bibliotecas y estanterías privadas son buenos sitios para buscar). Resumir el repertorio de instrucciones con las características discriminatorias utilizadas en las Figuras 3.1 y 3.5. ¿Se ajusta la máquina perfectamente en alguna de las categorías mostradas en las Figuras 3.4 y 3.6? Escribir la secuencia de código para esta máquina correspondiente a las sentencias de los dos apartados del Ejercicio 3.5.

3.11 [30] <3.7,3.9> Encontrar una máquina que tenga una característica potente de repertorio de instrucciones, tal como la instrucción CALLS del VAX. Sustituir la ins-

trucción potente por una secuencia más simple que realice lo que se necesite. Medir el tiempo de ejecución resultante. Comparar los resultados. ¿Por qué pueden ser diferentes? A principios de los ochenta, los ingenieros de DEC hicieron un rápido experimento para evaluar el impacto de sustituir las CALLS. Encontraron una mejora del 30 por 100 en el tiempo de ejecución en un programa que hacía un uso intensivo de llamadas (calls) cuando se sustituían las CALLS (los parámetros permanecían en la pila). Comparar este resultado con los obtenidos anteriormente.

El énfasis en el rendimiento mejor que en la estética es deliberado. Sin interés en el rendimiento el estudio de la arquitectura es un ejercicio estéril, ya que todos los problemas computables pueden resolverse utilizando arquitecturas triviales, dando tiempo suficiente. El desafío es diseñar computadores que hagan el mejor uso de la tecnología disponible; haciendo eso podemos estar seguros que cada aumento en la velocidad de procesamiento se puede utilizar para avanzar en los problemas actuales o hacer tratables problemas que antes eran impracticables.

Leonard J. Shustek, *Análisis y rendimiento de los repertorios de instrucciones de los computadores* (1978)

4.1. Medidas de los repertorios de instrucciones: qué y por qué

4.2. La arquitectura VAX

4.3. La arquitectura 360/370

4.4. La arquitectura 8086

4.5. La arquitectura DLX

4.6. Juntando todo: medidas de utilización del repertorio de instrucciones

4.7. Falacias y pifias

4.8. Observaciones finales

4.9. Perspectiva histórica y referencias

Ejercicios

4 Ejemplos y medidas de utilización de los repertorios de instrucciones

4.1 Medidas de los repertorios de instrucciones: qué y por qué

En este capítulo examinaremos algunas arquitecturas específicas y después medidas detalladas de las arquitecturas. Sin embargo, antes de hacer eso, expliquemos qué podemos querer medir y por qué, y también como medirlo.

Para comprender el rendimiento, habitualmente estamos más interesados en *medidas dinámicas* —medidas que se hacen al contar el número de ocurrencias de un evento durante la ejecución. Algunas medidas, como por ejemplo, tamaño de código, son inherentemente *medidas estáticas,* que se hacen en un programa independientemente de su ejecución. Las medidas estáticas y dinámicas pueden diferir enormemente, como muestra la Figura 4.1 —usar solamente los datos estáticos para este programa sería significativamente erróneo. En este texto los datos que se dan son dinámicos, a menos que se especifique otra cosa. Las excepciones surgen cuando solamente tienen sentido las medidas estáticas (como con el tamaño de código —el uso más importante de las medidas estáticas) y cuando es interesante comparar medidas estáticas y dinámicas. Como veremos en la sección de Falacias y Pifias y en los Ejercicios, la frecuencia de ocurrencia de dos instrucciones y el tiempo empleado en esas dos instrucciones puede a veces ser muy diferente.

Nuestro enfoque principal en este capítulo será introducir las arquitecturas y medir la utilización de las instrucciones para cada arquitectura. Aunque esto sugiere una concentración en los códigos de operación, también examinaremos la utilización de los modos de direccionamiento y del formato de las instrucciones.

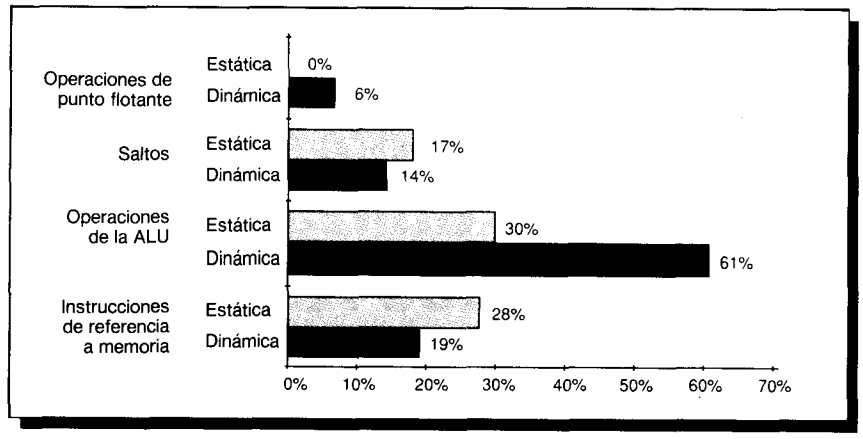


FIGURA 4.1 Datos de una medida de benchmark FORTRAN de IBM 360, que describimos con detalle en la Sección 4.6. Las 20 instrucciones más frecuentemente ejecutadas dinámicamente se han descompuesto en cuatro clases, mostrando las diferencias de las ocurrencias estáticas y dinámicas. En el caso de medidas dinámicas, estas 20 instrucciones contabilizan aproximadamente el 100 por 100 de las ejecuciones de instrucciones, pero sólo el 75 por 100 de las ocurrencias estáticas.

Las medidas del repertorio de instrucciones de la Sección 4.6 pueden utilizarse de dos formas. En un nivel alto, las medidas permiten formar aproximaciones generales de los patrones de utilización de las instrucciones en cada enfoque arquitectónico. Por ejemplo, veremos que las instrucciones que «cambian el PC» para un potente repertorio de instrucciones como el del VAX, promedian aproximadamente el 25 por 100 de las ejecuciones de todas las instrucciones. Esto nos indica que las técnicas que tratan de optimizar la búsqueda de la siguiente instrucción secuencial (buffers de prebúsqueda de instrucciones —explicado en la Sección 8.7 del Cap. 8) estarán significativamente limitadas ya que cada cuarta instrucción es un salto. Los datos del IBM 360 mostrarán que el uso de las instrucciones de cadena y decimales es casi inexistente en programas escritos en lenguajes distintos de COBOL. Esto nos lleva a la conclusión de que no es necesario incluir soporte para estas operaciones en una máquina concebida para el mercado científico. Las medidas de la frecuencia de operandos de memoria —aproximadamente el 40 por 100 de los operandos en el 8086— pueden utilizarse en el diseño de la segmentación del procesador y de la cache. Este tipo de medidas generales de alto nivel son los datos de fondo que un arquitecto de computadores debe utilizar casi a diario.

El otro propósito de estas medidas es que sirvan como una base de datos que pueda utilizar un arquitecto para hacer diseños detallados. Muchas veces hay que decidir qué se incluye y qué se omite en un repertorio de instrucciones, o al implementar un repertorio definido de instrucciones hay que elegir los casos que se desea hacer más rápidos. Por ejemplo, la baja frecuencia de uso de los modos de direccionamiento indirecto de memoria en el VAX po-

dría indicar al arquitecto que en una nueva arquitectura, omita este modo de direccionamiento. Si hubiese implementado un VAX sabría que la penalización de rendimiento por desfavorecer este complejo modo de direccionamiento sería pequeña. Otro ejemplo donde se debería utilizar información detallada, podría ser en la evaluación de saltos basados en códigos de condición. Examinando la frecuencia de los saltos condicionales y las instrucciones cuya única función es modificar el código de condición, podemos evaluar la frecuencia con la que implícitamente se modifica el código de condición (aproximadamente el 35 por 100 de las ocurrencias en el VAX). Este valor se puede utilizar para decidir qué tipos de saltos condicionales diseñar en una nueva arquitectura, o para optimizar la implementación de los saltos condicionales en un VAX. En este capítulo y los siguientes veremos muchos ejemplos de cómo se aplican estos datos a problemas específicos de diseño.

Hemos escogido cuatro máquinas para examinarlas: el VAX de DEC, el IBM 360, el Intel 8086, y una máquina genérica de carga/almacenamiento llamada DLX. Estas arquitecturas juegan un papel dominante en el mercado de computadores, y cada una tiene una serie de características únicas e interesantes. El Intel 8086 es el computador de propósito general más popular del mundo; se han vendido diez millones de máquinas con este microprocesador. El IBM 360 y el DEC VAX representan arquitecturas que han existido durante grandes períodos de tiempo (25+ y 10+ años, respectivamente) y de cada una de ellas se han vendido cientos de miles de unidades. DLX es representativa de una nueva raza de máquinas, que se ha hecho muy popular desde finales de los años ochenta. Como veremos estas máquinas también son muy diferentes en su estilo arquitectónico.

Para tratar de simplificar la tarea del lector, se utiliza un formato común para la sintaxis de las instrucciones. Este formato pone en primer lugar el destino de una instrucción de múltiples operandos, seguido de los operandos fuente primero y segundo. Así, una instrucción que reste `R3` de `R2` y ponga el resultado en `R1` se escribe:

```
SUB    R1,R2,R3
```

Este formato sigue el convenio utilizado en el Intel 8086, y es muy parecido al del 360. La única diferencia significativa con el 360 está en las instrucciones de almacenamiento que colocan primero el registro fuente. Aunque la sintaxis del VAX siempre pone en primer lugar el operando fuente y en último lugar el destino, el código del VAX lo veremos en nuestro formato común. Por supuesto, esta ordenación es puramente un convenio sintáctico y la arquitectura define la codificación de los operandos en el formato binario de la instrucción.

Las cuatro secciones siguientes son resúmenes de las cuatro arquitecturas. Aunque estos resúmenes son concisos, se exponen los atributos importantes y características más intensamente utilizadas. Las tablas que contienen todas las operaciones de los repertorios de instrucciones están en el Apéndice B. Para describir con precisión estas arquitecturas, es necesario introducir algunas extensiones adicionales a nuestro lenguaje C de descripción, para explicar las funciones de las instrucciones. Las adiciones son las siguientes:

- Se añade un subíndice al símbolo ←siempre que no esté clara la longitud del dato que se va a transferir. Por tanto $\leftarrow_n$ significa transferir una cantidad de n-bits.
- Se utiliza un subíndice para indicar que se ha seleccionado un bit de un campo. Los bits se etiquetan desde el bit más significativo comenzando en 0. El subíndice puede ser un solo dígito (por ejemplo, $R4_0$ contiene el bit de signo de R4) o un subrango (por ejemplo, $R3_{24...31}$ contiene el byte menos significativo de R3).
- Se utiliza un superíndice para replicar un campo (por ejemplo, 0^{24} contiene un campo de ceros cuya longitud es de 24 bits).
- La variable `M` se utiliza como un array que indica la memoria principal. El array está indexado por una dirección de bytes y puede transferir cualquier número de bytes.
- El símbolo # # se utiliza para concatenar dos campos y puede aparecer en cualquier parte de una transferencia de datos.

Un resumen del lenguaje de descripción completo aparece en la contraportada posterior del libro. A título de ejemplo, suponer que `R8` y `R10` son registros de 32 bits:

$$R10_{16..31} \leftarrow_{16} (M[R8]_0)^8 \#\# M[R8]$$

significa que el byte en la posición de memoria direccionada por el contenido de `R8` extendiendo el signo para formar una cantidad de 16 bits, se almacena en la mitad inferior de `R10`. (La mitad superior de `R10` permanece inalterada.)

Siguiendo los resúmenes de las arquitecturas a nivel lenguaje máquina en las cuatro secciones siguientes, examinaremos y contrastaremos las medidas de uso dinámico de las cuatro arquitecturas.

4.2 La arquitectura VAX

El primer modelo VAX de DEC, el VAX-11/780, apareció en 1977. El VAX fue diseñado como una extensión de 32 bits de la arquitectura PDP-11. Entre los objetivos del VAX, dos destacan por importantes y han tenido un impacto sustancial en las arquitecturas VAX.

Primero, los diseñadores querían que los usuarios del PDP-11 existente se sintiesen cómodos con la arquitectura VAX, y la considerasen una extensión de la del PDP-11. Esto motivó el nombre de VAX-11/780, el uso de una sintaxis muy similar del lenguaje ensamblador, la inclusión de los tipos de datos del PDP-11, y el soporte para emulación del PDP-11. Segundo, los diseñadores querían facilitar la tarea de escribir compiladores y sistemas operativos (OS). Esto se tradujo en una serie de objetivos que incluían definir interfaces entre lenguajes, hardware, y OS; y soportar una arquitectura altamente ortogonal.

En términos de modos de direccionamiento y operaciones soportados por las instrucciones, las demás arquitecturas explicadas en este capítulo son sub-

conjuntos muy grandes del VAX. Por esta razón nuestra discusión comienza con el VAX, que servirá como base para las comparaciones. El lector debe saber que hay libros completos dedicados a la arquitectura VAX, así como gran número de artículos que informan sobre las medidas del repertorio de instrucciones. Nuestro resumen sobre el repertorio de instrucciones del VAX —igual que los demás resúmenes de los repertorios de instrucciones de este capítulo— se centran en los principios generales y en las partes más relevantes de la arquitectura para comprender las medidas examinadas aquí. En el Apéndice B se da una lista del repertorio de instrucciones completo del VAX.

El VAX es una máquina de registros de propósito general con un repertorio de instrucciones muy ortogonal. La Figura 4.2 muestra los tipos de datos soportados. El VAX utiliza el nombre «palabra» para referenciar cantidades

Bits	Tipo de datos	Nuestro nombre	Nombre de DEC
8	Entero	Byte	Byte
16	Entero	Media palabra	Palabra (Word)
32	Entero	Palabra	Palabra larga (Long word)
32	Punto flotante	Simple precisión	F_flotante
64	Entero	Doble palabra	Cuad palabra (Quad-word)
64	Punto flotante	Doble precisión	D_flotante o G_flotante
128	Entero	Cuad palabra	Octa-palabra
128	Punto flotante	Huge (Enorme)	H_flotante
8n	Cadena de caracteres	Carácter	Carácter
4n	Decimal codificado binario	Empaquetado	Empaquetado (Packed)
8n	Cadena numérica	Desempaquetado	Cadenas numéricas: (Numeric String)

FIGURA 4.2 Tipos de datos del VAX, sus longitudes y nombres. La primera letra del tipo DEC (B, W, L, F, Q, D, G, O, H, C, P, T, S) se utiliza con frecuencia para completar un nombre de código de operación. Como ejemplos, los códigos de operación incluyen `MOVB`, `MOVW`, `MOVL`, `MOVF`, `MOVQ`, `MOVD`, `MOVG`, `MOVO`, `MOVH`, `MOVC3`, `MOVP`. Cada instrucción *(move)* transfiere un operando del tipo de dato indicado por la letra que sigue a `MOV`. (No hay diferencia entre operaciones «move» de caracteres y de cadenas numéricas, así que sólo se necesitan las operaciones de desplazamiento de caracteres.) Los campos de longitud que aparezcan como *X*n indican que la longitud puede ser cualquier múltiplo de *X* en bits. El tipo de dato empaquetado es especial ya que la longitud para las operaciones de este tipo se da siempre en dígitos, cada uno de los cuales es de cuatro bits. Los objetos empaquetados se ubican y direccionan en unidades de bytes. Para cualquier tipo de cadena de datos la dirección de comienzo es la dirección de orden inferior de la cadena.

de 16 bits, aunque en este texto utilicemos el convenio de que una *palabra* es de 32 bits. Tener cuidado al leer nemotécnicos de las instrucciones del VAX, ya que con frecuencia referencian los nombres de los tipos de datos del VAX. La Figura 4.2 muestra la conversión entre los nombres de los tipos de datos utilizados en este texto y los nombres que utiliza el VAX. Además de los tipos de datos de la Figura 4.2, el VAX proporciona soporte para cadenas de bits de longitudes fija y variable, de hasta 32 bits.

El VAX tiene 16 registros de propósito general, pero cuatro registros son utilizados realmente por la arquitectura. Por ejemplo, R14 es el puntero de pila y R15 es el PC (contador de programa). Por consiguiente, R15 no puede ser utilizado como registro de propósito general, y utilizar R14 es muy difícil porque interfiere con instrucciones que manipulan la estructura de la pila. Los códigos de condición se utilizan para saltos y son inicializados por todas las operaciones aritméticas y lógicas y por la instrucción de transferir *(move)*. La instrucción «move» transfiere datos entre dos posiciones direccionables cualesquiera y engloba cargas, almacenamientos, transferencias de registro-registro, y transferencias de memoria-memoria como casos especiales.

Modos de direccionamiento del VAX

Los modos de direccionamiento incluyen la mayoría de los explicados en el Capítulo 3: literal, registro (el operando está en un registro), diferido de registro (indirecto de registro), autodecremento, autoincremento, autoincremento diferido, desplazamiento de byte/palabra/largo, desplazamiento diferido de byte/palabra/largo, y escalado (llamado «indexado» en la arquitectura VAX). El modo de direccionamiento escalado puede aplicarse a cualquier modo de direccionamiento general excepto al de registro o literal. El registro es un modo de direccionamiento no diferente de cualquier otro en el VAX. Por tanto, una instrucción VAX de tres operandos puede incluir desde cero a tres referencias de operandos a memoria, cada una de las cuales puede ser cualquier modo de direccionamiento de memoria. Como los modos indirectos de memoria, requieren un acceso adicional a memoria. Para una instrucción de tres operandos se pueden necesitar hasta 6 accesos a memoria. Cuando se utilizan los modos de direccionamiento con R15 (el PC), sólo algunos están definidos, y su significado es especial. Los modos de direccionamiento definidos con R15 son los siguientes:

- *Inmediato:* un valor inmediato está en el flujo de instrucciones; este modo está codificado como autoincremento con el PC.
- *Absoluto:* una dirección absoluta de 32 bits está en el flujo de instrucciones; este modo se codifica como autoincremento diferido con el PC como registro.
- *Desplazamiento de byte/palabra/largo:* igual que el modo general, pero la base es el PC, dando direccionamiento relativo al PC.
- *Desplazamiento diferido de byte/palabra/largo:* igual que el modo general, pero la base es el PC, dando direccionamiento indirecto mediante una posición de memoria que es relativa al PC.

Una instrucción VAX consta de un código de operación seguido por cero o más especificadores de operandos. El código de operación casi siempre es un solo byte que especifica la operación, el tipo de dato y el número de operandos. Casi todas las operaciones son completamente ortogonales con respecto a los modos de direccionamiento —cualquier combinación de modos de direccionamiento funciona con casi todos los códigos de operación, y muchas operaciones están soportadas para todos los posibles tipos de datos.

La longitud de los especificadores de operando puede variar desde un byte a muchos, dependiendo de la información que se vaya a transferir. El primer byte de cada especificador de operando consta de dos campos de 4 bits: el tipo del especificador de direcciones y un registro que es parte del modo de direccionamiento. Si el especificador de operando requiere bytes adicionales para especificar un desplazamiento, registros adicionales, o un valor inmediato éste aumenta mediante incrementos de 1 byte. El nombre, sintaxis del ensamblador, y número de bytes de cada especificador de operando se muestran en la Figura 4.3. El formato y la longitud total de la instrucción son fáciles de es-

Modo de direccionamiento	Sintaxis	Longitud en bytes
Literal	#valor	1(bit-6 valor signo)
Inmediato	#valor	1 + longitud del inmediato
Registro	Rn	1
Registro diferido	(Rn)	1
Desplazamiento de byte/palabra/largo	Desplazamiento (Rn)	1 + longitud del desplazamiento
Desplazamiento de byte/palabra/largo	@Desplazamiento (Rn)	1 + longitud del desplazamiento
Escalado (indexado)	Modo base [Rx]	1 + longitud del modo de direccionamiento base
Autoincremento	(Rn)+	1
Autodecremento	-(Rn)	1
Autoincremento diferido	@(Rn)+	1

FIGURA 4.3 Longitud de los especificadores de operandos VAX. La longitud de cada modo de direccionamiento es 1 byte más la longitud de cualquier desplazamiento o campo inmediato que esté en el modo. El modo literal utiliza un señalizador especial de 2 bits y los 6 bits restantes codifican el valor constante. Los datos que examinamos en el Capítulo 3 sobre constantes mostraban el uso intenso de pequeñas constantes; la misma observación motivó esta optimización. La longitud de un inmediato la dicta el tipo de datos indicado en el código de op, no el valor del inmediato.

tablecer: simplemente sumar los tamaños de los especificadores de operando e incluir un byte (o raramente dos) para el código de operación.

Ejemplo

¿Qué longitud tiene la siguiente instrucción?

```
ADDL3 R1,737(R2),#456
```

Respuesta

La longitud del código de operación es 1 byte, igual que el especificador del primer operando (R1). El especificador del segundo operando tiene dos partes: la primera es un byte que especifica el modo de direccionamiento y registro base; la segunda es un desplazamiento de 2 bytes. El especificador del tercer operando también tiene dos partes: el primer byte especifica modo inmediato, y la segunda parte contiene el inmediato. Debido a que el tipo de datos es largo (ADDL3), el valor del inmediato necesita 4 bytes.

Por tanto, la longitud total de la instrucción es 1 + 1 + (1 + 2) + (1 + 4) = 10 bytes.

Tipo	Ejemplo (*)	Significado de la instrucción
Transferencias de datos		**Transferencia de datos entre operandos de byte, media palabra, palabra o doble palabra; *es el tipo de datos**
	MOV*	Transferencia entre dos operandos
	MOVZB*	Transfiere un byte a una media palabra o palabra, extendiéndolo con ceros
	MOVA*	Transfiere dirección de operando; el tipo de datos es la parte final
	PUSH*	Introduce operando en pila
Aritmética, lógica		**Operaciones sobre bytes enteros o lógicos, medias palabras (16 bits), palabras (32 bits); *es el tipo de datos**
	ADD*_	Suma con 2 ó 3 operandos
	CMP*	Compara e inicializa códigos de condición
	TST*	Compara con cero e inicializa códigos de condición
	ASH*	Desplazamiento aritmético
	CLR*	Pone a cero
	CVTB*	Byte de signo extendido para tamaño de tipo de datos
Control		**Saltos condicionales e incondicionales**
	BEQL, BNEQ	Salta igual/no igual
	BCS, BCC	Salta acarreo a 1, salta acarreo a 0
	BRB, BRW	Salto incondicional con un desplazamiento de 8 ó 16 bits
	JMP	Bifurcación utilizando cualquier modo de direccionamiento para especificar el objeto
	AOBLEQ	Suma uno al operando; salta si el resultado $\leqslant$ segundo operando
	CASE	Salta basado en el selector case

Tipo	Ejemplo (*)	Significado de la instrucción
Procedimiento		**Llamada/retorno de procedimiento**
	CALLS	Llama a procedimiento con argumentos en pila (ver Sección 3.9)
	CALLG	Llama a procedimiento con lista de parámetros de estilo FORTRAN
	JSB	Salta a subrutina, guardando dirección de vuelta
	RET	Retorno de llamada de procedimiento
Carácter decimal de campo de bits		**Opera sobre campos de bits de longitud variable, cadenas de caracteres y cadenas decimales, ambas en formato de caracteres y BCD**
	EXTV	Extrae un campo de bits de longitud variable en una palabra de 32 bits
	MOVC3	Transfiere una cadena de caracteres de longitud dada
	CMPC3	Compara dos cadenas de caracteres de longitud dada
	MOVC5	Transfiere cadena de caracteres con truncación o ajuste
	ADDP4	Suma cadena decimal de longitud indicada
	CVTPT	Convierte cadena decimal empaquetada en cadena de caracteres
Punto flotante		**Operaciones de punto flotante sobre formatos D, F, G y H**
	ADDD_	Suma números flotantes en formato D en doble precisión
	SUBD_	Resta números flotantes en formato D en doble precisión
	MULF_	Multiplica punto flotante en formato F en simpre precisión
	POLYF	Evalúa un polinomio utilizando una tabla de coeficientes en formato F
Sistema		**Cambia a modo de sistema, modifica registros protegidos**
	CHMK, CHME	Cambia modo a kernel (núcleo)/ejecutivo
	REI	Retorno de excepción o interrupción
Otros		**Operaciones especiales**
	CRC	Calcula comprobación de redundancia cíclica
	INSQUE	Inserta una entrada de cola en una cola

FIGURA 4.4 Clases de instrucciones VAX con ejemplos. El asterisco significa múltiples tipos de datos —B, W, L, y habitualmente D, F, G, H y Q; recordar cómo estos tipos VAX están relacionados los nombres usados en el texto (ver Fig. 4.2). Por ejemplo, MOVW transfiere una palabra según el tipo de datos VAX, que es de 16 bits y en este texto se denomina media palabra. El subrayado, como en ADDD_, significa que en esta instrucción hay 2 operandos (ADDD2) y 3 operandos (ADDD3). El número de operandos está explícito en el código de operación.

Operaciones en el VAX

¿Qué tipos de operadores proporciona el VAX? Las operaciones del VAX pueden dividirse en clases, como muestra la Figura 4.4. (Listas detalladas de las instrucciones del VAX están en el Apéndice B.) La Figura 4.5 da ejemplos de instrucciones típicas del VAX y sus significados. La mayoría de las instrucciones actualizan los códigos de condición del VAX de acuerdo con su resultado; las instrucciones sin resultado, como los saltos, no. Los códigos de con-

Ejemplo de instrucción de ensamblador	Longitud	Significado
MOVL @40(R4),30(R2)	5	M[M[40+R4]] $\leftarrow_{32}$ M[30+R2]
MOVAW R2,(R3)[R4]	4	R2 $\leftarrow_{32}$ R3+(R4*2)
ADDL3 R5,(R6)+,(R6)+	4	i←M[R6];R6←R6+4; R5←i+M[R6]; R6←R6+4
CMPL -(R6),#100	7	R6←R6-4; Inicializa el código de condición usando: M[R6]-10
CVTBW R10,(R8)	3	$R10_{16..31} \leftarrow_{16} (M[R8]_0)^8$ ## M[R8]
BEQL name	2	if equal(CC) {PC←name} PC-128 $\leqslant$ name < PC+128
BRW name	3	PC←name PC-32768 $\leqslant$ name < PC+32768
EXTZV (R8),R5,R6,-564(R7)	7	$t \leftarrow_{40}$ M[R7-564+(R5>>3)]; i←R5 & 7; j←if R6>=32 then 32 else if R6<0 then 0 else R6; M[R8] $\leftarrow_{32} 0^{32-j}$ ## $t_{39-i-j+1..39-i}$;
MOVC3 @36(R9),(R10),35(R11)	6	R1←35+R11; R3←M[36+R9]; for (R0←M[R10];R0!=0;R0--) {M[R3] $\leftarrow_8$ M[R1]; R1++; R3++} R2=0; R4=0; R5=0
ADDD3 R0,R2,R4	4	(R0##R1) $\leftarrow_{64}$ (R2##R3) + (R4##R5) contenido del registro son tipo D punto flotante

FIGURA 4.5 Algunos ejemplos de instrucciones típicas del VAX. La sintaxis del lenguaje ensamblador VAX pone al final el operando resultado; lo hemos puesto primero por consistencia con las otras máquinas. La longitud de la instrucción se da en bytes. La condición `equal (CC)` es verdadera si la inicialización del código de condición refleja la igualdad después de una comparación. Recordar que la mayoría de las instrucciones inicializan el código de condición; la única función de las instrucciones de comparación es inicializar el código de condición. Los nombres `t`, `i`, `j` se utilizan como temporales en las descripciones de las instrucciones; t es de 40 bits de longitud, mientras que i y j son de 32 bits. La instrucción `EXTZV` puede parecer misteriosa. Su propósito es extraer campos de longitud variable (0 a 32 bits) y extender el cero a 32 bits. Los operandos fuente para `EXTZV` son la posición del bit de comienzo (que puede ser cualquier distancia desde la dirección del byte de comienzo), la longitud del campo y la dirección de comienzo de la cadena de bits para extraer el campo. El VAX numera sus bits desde el orden más bajo al más alto, pero nosotros numeramos los bits en el orden inverso. Por tanto, los subíndices ajustan los desplazamientos de los bits adecuadamente (¡lo que hace que `EXTV` parezca más misteriosa!). Aunque el resultado de las operaciones con cadenas de bits variables es siempre de 32 bits, la `MOVC3` cambia los valores de los registros R0 a R5 como se indica (aunque cualquiera de los R0, R2, R4 y R5 pueda ser utilizado para que contenga la cuenta). En la Sección 5.6 del capítulo siguiente aparece una discusión de por qué `MOVC3` utiliza los GPR como registros de trabajo.

dición son N (Negativo), Z (Cero-Zero), V (Desbordamiento-Overflow), y C (Acarreo-Carry).

4.3 La arquitectura 360/370

El IBM 360 se introdujo en 1964. Sus objetivos oficiales incluían lo siguiente:

1. Explotar la memoria: gran memoria principal, jerarquías de memoria (ROM utilizada para microcódigo).
2. Soportar E/S concurrentes: hasta 5 MB/segundo con una interfaz estándar en todas las máquinas.
3. Crear una máquina de propósito general con nuevas facilidades de los OS y muchos tipos de datos.
4. Mantener estricta compatibilidad ascendente y descendente del lenguaje máquina.

El Sistema 370, se dió a conocer en 1970, como sucesor del Sistema 360. El Sistema 370 es completamente compatible de forma ascendente con el Sistema 360, aun en el modo sistema. Las principales extensiones sobre el 360 incluían

- Memoria virtual y traducción dinámica de direcciones (ver Cap. 8, Sección 8.5)
- Algunas instrucciones nuevas: soporte de sincronización, instrucciones de cadenas largas (desplazamientos largos y comparaciones largas), instrucciones adicionales para manipular bytes en registros, y algunas instrucciones adicionales decimales
- Eliminación de los requerimientos de alineación de datos

Además, se introdujeron algunas diferencias importanes de implementación en el 370 entre las que se incluyen memoria principal MOS en lugar de núcleos, y memoria de control escribible (ver Cap. 5).

En 1983, IBM introdujo el 370-XA, la arquitectura eXtendida. Hasta esta extensión, utilizada primero en la serie 3080, la arquitectura 360/370 tenía un espacio de direcciones de 24 bits. Se añadieron bits a la palabra de estado del programa para poder ampliar el contador de programa. Por desgracia, una práctica de programación común en la 360, era utilizar el byte de orden superior de una dirección para el estado. Por tanto, los antiguos programas de 24 bits no podían ejecutarse en el modo de 32 bits (realmente una dirección de 31 bits), mientras que los programas nuevos y recompilados podían aprovechar el mayor espacio de direcciones. También se cambió la estructura de E/S, para permitir mayores niveles de multiprocesamiento.

La última extensión a la arquitectura fue la ESA/370, introducida en el modelo 3090 en 1986. La ESA/370 añadió formatos adicionales de instruc-

ción, llamados formatos extendidos, con códigos de operación de 16 bits. La ESA/370 incluye soporte para Facilidad Vectorial *(Vector Facility)* (incluyendo un conjunto de registros vectorales) y formato de punto flotante extendido (128 bits). El espacio de direcciones se amplió al añadir segmentos al espacio de direcciones de 31 bits (ver Cap. 8, Secciones 8.5 y 8.6); también se añadió un nuevo y más potente modelo de protección.

El resto de esta sección da una visión general de la arquitectura IBM 360 y presenta medidas para cargas de trabajo. Primero, examinamos los aspectos básicos de la arquitectura 360, después consideramos los formatos del repertorio de instrucciones y algunas instrucciones de ejemplo.

Arquitectura a nivel lenguaje máquina del 360/370

El IBM System/360 es una máquina de 32 bits con direccionabilidad por bytes que soporta diversos tipos de datos: byte, media palabra (16 bits), palabra (32 bits), doble palabra (doble precisión real), decimal empaquetado y cadenas de caracteres desempaquetados. El System/360 tiene restricciones de alineación, que se eliminaron en la arquitectura System/370.

El estado interno del 360 tiene los siguientes componentes:

- Dieciséis registros de propósito general de 32 bits; el registro 0 es especial cuando se utiliza en un modo de direccionamiento, donde se sustituye siempre un cero.
- Cuatro registros de punto flotante de doble precisión (64 bits).
- La palabra de estado de programa (PSW) contiene el PC, algunos señalizadores, y los códigos de condición.

Versiones posteriores de la arquitectura ampliaron este estado con registros adicionales de control.

Modos de direccionamiento y formatos de instrucción

El 360/370 tiene cinco formatos de instrucción. Cada formato está asociado a un modo de direccionamiento y tiene un conjunto de operaciones definidas para ese formato. Algunas de estas operaciones están definidas en formatos múltiples, aunque la mayoría no. Los formatos de instrucción se muestran en la Figura 4.6. Aunque muchas instrucciones siguen el paradigma de operar sobre fuentes y poner el resultado en un destino, otras instrucciones (como las de control BAL, BALR, BC) no siguen este paradigma, pero utilizan los mismos campos para otros propósitos. Los modos de direccionamiento asociados son los siguientes:

RR *(registro-registro).*—Ambos operandos son sencillamente el contenido de los registros. El primer operando fuente es también el destino.

RX *(registro-indexado).*—El primer operando y el destino están en un registro. El segundo operando es el contenido de la posición de memoria dada por

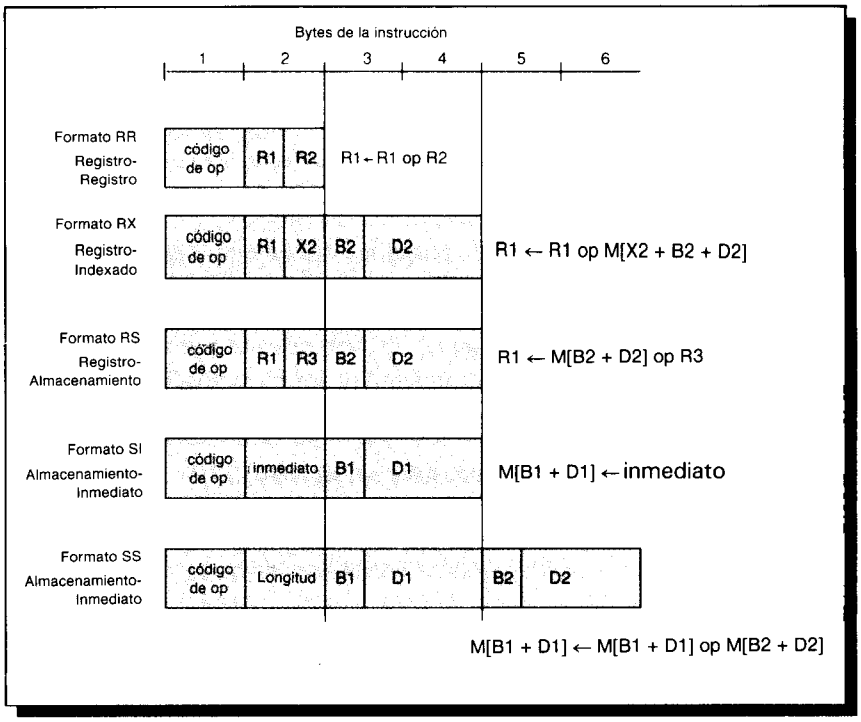


FIGURA 4.6 Formatos de instrucciones 360/370. Los posibles operandos de instrucciones son un registro (R1, R2 o R3), un inmediato de 8 bits, o una posición de memoria. El código de operación especifica dónde residen los operandos y el modo de direccionamiento. Las direcciones efectivas para operandos de memoria se forman utilizando la suma de uno o dos registros (llamados B1, B2 o X2) y un campo de desplazamiento de 12 bits sin signo (llamado D1 o D2). Además, las instrucciones memoria-memoria, que están todas orientadas a cadena, especifican un campo de longitud de 8 bits. Otros formatos de instrucción se han añadido en extensiones arquitectónicas posteriores. Estos formatos permitían que se extendiese el espacio del código de operación y se añadiesen nuevos tipos de datos. Para cargas, almacenamientos y desplazamientos sólo se utiliza un operando fuente y la operación sólo transfiere el dato (ver Fig. 4.8). Para instrucciones SS, la longitud es uno mayor que el valor en la instrucción.

la suma de un campo D2 de desplazamiento de 12 bits, el contenido del registro B2, y el del registro X2. Este formato se utiliza cuando se necesita un registro índice (y para la mayoría de las cargas y almacenamientos).

RS *(registro-memoria).*—El primer operando es un registro que es el destino. El tercer operando es un registro que se utiliza como segunda fuente. El segundo operando es el contenido de la posición de memoria dada por la suma del campo de desplazamiento de 12 bits, D2, y el contenido del registro B2. El modo RS se diferencia del RX en que soporta una forma de 3 operandos, pero elimina el registro índice. Este formato se utiliza sólo para un pequeño número de instrucciones.

SI *(memoria-inmediato).*—El destino es un operando de memoria dado por la suma del contenido del registro B1 y el valor del desplazamiento D1. El segundo operando, un campo inmediato de 8 bits, es la fuente.

SS *(memoria-memoria).*—Las direcciones de los dos operandos de memoria son la suma del contenido de un registro base Bi y un desplazamiento Di. El primer operando es el destino. Esta operación memoria-a-memoria se utiliza para operaciones decimales y para cadenas de caracteres. El campo de longitud puede especificar una longitud de 1 a 256, o dos longitudes, cada una de 1 a 16. Para las instrucciones de cadena se utiliza una sola longitud, mientras que las instrucciones decimales especifican una longitud para cada operando.

El desplazamiento de los formatos RS, RX, SI y SS es de 12 bits sin signo.

Operaciones en el 360/370

Justo como en el VAX, las instrucciones de la 360 pueden dividirse en cuatro clases. Están soportados cuatro tipos básicos de operaciones sobre los datos:

1. *Operaciones lógicas sobre bits, cadenas de caracteres, y cadenas fijas.* Son la mayoría de los formatos RR y RX con algunas instrucciones RS.
2. *Operaciones decimales o de caracteres sobre cadenas de caracteres o dígitos decimales.* Son las instrucciones de formato SS.
3. *Aritmética binaria de punto fijo.* Está soportada en los formatos RR y RX.
4. *Aritmética de punto flotante.* Está soportada principalmente con instrucciones RR y RX.

Tipo	Instrucción ejemplo	Significado
RR	`AR R4,R5`	`R4←R4+R5`
RX	`A R4,10(R5,R6)`	`R4←R4+M[R5+R6+10]`
RX	`BC Mask,20(R5,R6)`	`if (CC & Mask)!=0 {PC←20+R5+R6}`
RS	`STM 20(R14),R2,R8`	`for(i=2;i<=8;i++) {M[R14+20+(i-2)*4]←32Ri}`
SI	`MVI 20(R5),#40`	`M[R5+20]←8 40`
SS	`MVC 10(R2),Len,20(R6)`	`for(i=0;i<Len+1;i++) {M[R2+10+i]←8 M[R6+20+i]}`

FIGURA 4.7 Instrucciones típicas del IBM 360 con sus significados. La instrucción MVC se muestra con la longitud como segundo operando. El campo de longitud es una constante en la instrucción; la sintaxis estándar del lenguaje ensamblador 360 incluye la longitud como primer operando. La variable i usada en la MVC y STM es temporal.

Los saltos utilizan el formato de instrucción RX especificando la dirección destino del salto mediante la dirección efectiva. Como los saltos no son relativos al PC, puede ser necesario cargar un registro base para especificar el destino del salto. Esto tiene un impacto sustancial: en general, significa que debe haber registros que apunten a cada región que contenga un destino de salto. Los códigos de condición los inicializan todas las operaciones aritméticas y lógicas. Los saltos condicionales examinan los códigos de condición bajo una máscara para determinar si hay salto o no lo hay.

En la Figura 4.7 se muestran algunas instrucciones ejemplo y sus formatos. Cuando se define una operación para más de un formato, se utilizan códigos de operación separados para especificar el formato de instrucción. Por ejemplo, el código de operación AR (suma de registro) indica que el tipo de

Clase o instrucción	Formato	Significado de la instrucción
Control		**Cambia el PC**
BC_	RS,RR	Examina la condición y salta condicionalmente
BAL_	RS,RR	Salta y enlaza (la dirección de la siguiente instrucción se coloca en R15)
Aritmética, lógica		**Operaciones aritméticas y lógicas**
A_	RX,RR	Suma
S_	RX,RR	Resta
SLL	RS	Desplazamiento lógico a la izquierda; desplaza un registro una cantidad inmediata
LA	RX	Carga dirección—pone la dirección efectiva en el destino
CLI	SI	Compara byte de memoria con inmediato
NI	SI	AND inmediato en byte de memoria
C_	RX,RR	Compara y modifica los códigos de condición
TM	RS	Test bajo máscara—realiza una AND lógica del operando y un campo inmediato; inicializa los códigos de condición según el resultado
MH	RX	Multiplica media palabra
Transferencia de datos		**Transferencias entre registros o registro y memoria**
L_	RX,RR	Carga un registro desde memoria u otro registro
MVI	SI	Almacena un byte inmediato en memoria
ST	RX	Almacena un registro
LD	RX	Carga un registro de punto flotante de doble precisión
STD	RX	Almacena un registro de punto flotante de doble precisión
LPDR	RR	Transfiere un registro de punto flotante de doble precisión a otro
LH	RX	Carga media palabra desde memoria en un registro
IC	RX	Inserta un byte de memoria en el byte de orden inferior de un registro
LTR	RS	Carga un registro e inicializa códigos de condición

Clase o instrucción	Formato	Significado de la instrucción
Punto flotante		**Operaciones de punto flotante**
AD_	RS,RR	Suma en punto flotante y doble precisión
MD_	RS,RR	Multiplica FP doble precisión
Cadena, decimal		**Operaciones sobre decimales y cadenas de caracteres**
MVC	SS	Transfiere caracteres
AP	SS	Suma cadenas decimales empaquetadas, sustituyendo la primera con la suma
ZAP	SS	Pone a cero y suma empaquetada—sustituye el destino por la fuente
CVD	RX	Convierte una palabra binaria en doble palabra decimal
MP	SS	Multiplica dos cadenas decimales empaquetadas
CLC	SS	Compara dos cadenas de caracteres
CP	SS	Compara dos cadenas de decimales empaquetadas
ED	SS	Edita—convierte decimal empaquetado en cadena de caracteres

FIGURA 4.8 Instrucciones del IBM 360 más frecuentemente utilizadas. El subrayado significa que el código de operación son dos códigos de operación distintos: uno con formato RX y otro con formato RR. Por ejemplo A_ significa AR y A. El repertorio completo de instrucciones se muestra en el Apéndice B.

instrucción es RR; por tanto, los operandos están en registros. El código de operación A (suma) indica que el formato es RX; por tanto, un operando está en memoria, a la que se accede con el modo de direccionamiento RX. La Figura 4.8 presenta un listado mayor de operaciones que incluye las más comunes; en el Apéndice B se da una tabla completa de las instrucciones.

4.4 La arquitectura 8086

La arquitectura del Intel 8086 fue anunciada en 1978 como una extensión ascendente compatible del entonces famoso 8080. Aunque el 8080 claramente era una máquina de acumulador, el 8086 amplió la arquitectura con registros adicionales. Sin embargo, el 8086 falla si realmente se considera como una máquina de registros de propósito general, porque prácticamente cada registro tiene un uso dedicado. Por tanto, su arquitectura está comprendida entre la de una máquina de acumulador y una máquina de registros de propósito general. El 8086 es una arquitectura de 16 bits; todos los registros internos son de 16 bits. Para obtener una direccionabilidad mayor de 16 bits los diseñadores añadieron segmentos a la arquitectura, logrando así un espacio de direcciones de 20 bits, organizados en fragmentos de 64 KB. El Capítulo 8 explica con detalle la segmentación, mientras que en este capítulo nos centraremos solamente en las implicaciones para un compilador.

Los 80186, 80286, 80386 y 80486 son extensiones «compatibles» de la arquitectura 8086 y se referencian colectivamente como procesadores 80×86. Son compatibles en el sentido de que todos pertenecen a la misma familia arquitectónica. Hay más ejemplares de esta familia en el mundo que de cualquier otra. El 80186 añadió un pequeño número de extensiones (unas 16) a la arquitectura 8086 en 1981. El 80286, introducido en 1982, amplió la arquitectura del 80186 creando un elaborado modelo de protección y de mapa de memoria y aumentando el espacio de direcciones a 24 bits (ver Cap. 8, Sección 8.6). Debido a que los programas del 8086 necesitaban ser compatibles binarios, el 80286 ofreció un modo de direccionamiento real para hacer que la máquina se comportase como un 8086.

El 80386 se introdujo en 1985. Es una verdadera máquina de 32 bits cuando corre en modo nativo. Igual que el 80286, está provisto de un modo de direccionamiento real para que sea compatible con el 8086. Hay también un modo virtual 8086 que porporciona, en la memoria del 80386, múltiples particiones de direcciones del 8086 de 20 bits. Además de una arquitectura de 32 bits con registros de 32 bits y un espacio de direcciones de 32 bits, el 80386 tiene un nuevo conjunto de modos de direccionamiento y operaciones adicionales. Las instrucciones añadidas hacen al 80386 prácticamente una máquina de registros de propósito general —para la mayoría de las operaciones se puede utilizar cualquier registro como operando. El 80386 también tiene soporte de paginación (ver Cap.8). El 80486 se introdujo en 1989 con algunas instrucciones nuevas, aunque con un incremento sustancial del rendimiento.

Como el modo de compatibilidad con el 8086 es el uso dominante de todos los procesadores 80×86, en esta sección daremos un vistazo detallado a la arquitectura 8086. Comenzaremos resumiendo la arquitectura y después explicaremos su utilización por programas típicos.

Resumen del repertorio de instrucciones del 8086

El 8086 soporta tipos de datos de 8 bits (byte) y de 16 bits (palabra). Las distinciones de tipos de datos se aplican a las operaciones de registros así como a los accesos a memoria.

El espacio de direcciones en el 8086 es de 20 bits; sin embargo, está descompuesto en segmentos, de 64 KB, direccionables con desplazamientos de 16 bits. Una dirección de 20 bits se forma tomando una dirección efectiva de 16 bits —como un desplazamiento en un segmento— y sumándola a una dirección del segmento base de 20 bits. La dirección del segmento base se obtiene desplazando 4 bits a la izquierda el contenido de un registro de 16 bits.

El 8086 tiene un total de 14 registros, divididos en cuatro grupos —registros de datos, registros de dirección, registros de segmento y registros de control— como muestra la Figura 4.9. El registro segmento para un acceso a memoria habitualmente está implicado por el registro base utilizado para formar la dirección efectiva en el segmento.

Los modos de direccionamiento para los datos en el 8086 utilizan los registros segmento implicados por el modo de direccionamiento o especificado en la instrucción con una anulación del modo implícito. Explicaremos cómo

se relacionan los saltos y bifurcaciones con la segmentación en la sección sobre operaciones.

Clase	Registro	Propósitos de clase o registro
Dato		**Usado para que contenga y opere sobre datos**
	AX	Usado para multiplicar, dividir, y E/S; a veces un operando implícito; AH y AL también tienen usos dedicados en multiplicación, división, aritmética decimal de bytes
	BX	También puede usarse como registro de dirección base
	CX	Usado para operaciones de cadena e instrucciones de bucle; CL es la cuenta de desplazamientos dinámicos
	DX	Utilizado para multiplicar, dividir y E/S
Dirección		**Usado para formar direcciones efectivas de memoria de 16 bits (en segmento)**
	SP	Puntero de pila
	BP	Registro base—usado en modo de direccionamiento basado
	SI	Registro índice, y también utilizado como registro base de cadena fuente
	DI	Registro índice, y también utilizado como registro base de cadena destino
Segmento		**Usado para formar direcciones reales de memoria de 20 bits**
	CS	Segmento de código—utilizado en acceso de instrucciones
	SS	Segmento de pila—utilizado para referencias de pila (SP) o cuando BP es registro base
	DS	Segmento de datos—utilizado cuando una referencia no es para el código (usado CS), para la pila (usado SS), o un destino cadena (usado ES)
	ES	Segmento extra—utilizado cuando el operando es cadena destino
Control		**Usado para control y estado de programa**
	IP	Puntero de instrucción —proporciona el desplazamiento de la instrucción que actualmente se está ejecutando (éstos son los 16 bits inferiores del PC efectivo)
	FLAGS (señalizadores)	Contiene seis bits de código de condición —acarreo, cero, signo, préstamo, paridad y desbordamiento— y tres bits de control de estado

FIGURA 4.9 Los 14 registros del 8086. La tabla los divide en cuatro clases que tienen usos restringidos. Además, muchos de los registros individuales son requeridos para ciertas instrucciones. Los registros de datos tienen una mitad superior y otra inferior: *x*L referencia el byte inferior y *x*H el byte superior del registro *x*.

Modos de direccionamiento

La mayoría de los modos de direccionamiento para formar la dirección efectiva de un operando o dato están entre los explicados en el Capítulo 3. Las instrucciones aritméticas, lógicas y de transferencia de datos son instrucciones de dos operandos que permiten las combinaciones mostradas en la Figura 4.10.

Tipo de operando fuente/destino	Segundo operando fuente
Registro	Registro
Registro	Inmediato
Registro	Memoria
Memoria	Registro
Memoria	Inmediato

FIGURA 4.10 Tipos de instrucciones para las instrucciones aritméticas, lógicas y de transferencia de datos. El 8086 permite las combinaciones mostradas. Los inmediatos pueden ser de 8 o de 16 bits de longitud; un registro es cualquiera de los 12 registros principales de la Figura 4.9 (ninguno de los registros de control). La única restricción es la ausencia del modo memoria-memoria.

Los modos de direccionamiento de memoria soportados son el absoluto (dirección absoluta de 16 bits), indirecto de registro, registro base, indexado, y registro base indexado con desplazamiento (no mencionado en el Capítulo 3). Aunque un operando de memoria pueda utilizar cualquier modo de direccionamiento, hay restricciones sobre los registros que se pueden utilizar en un determinado modo. Los registros utilizables al especificar la dirección efectiva son los siguientes:

- *Registro indirecto:* BX, SI, DI.
- *Modo registro base con desplazamiento de 8 o 16 bits:* BP, BX, SI, DI. (Intel da dos nombres a este modo de direccionamiento, Basado e Indexado, pero como son especialmente idénticos los combinamos.)
- *Indexado:* la dirección es la suma de dos registros. Las combinaciones permisibles son BX + SI, BX + DI, BP + SI, y BP + DI. Este modo se denomina registro base Indexado en el 8086.
- *Registro base Indexado con desplazamiento de 8 o 16 bits:* la dirección es la suma del desplazamiento y de los contenidos de dos registros. Se aplican las mismas restricciones sobre los registros, que en el modo indexado.

Operaciones en el 8086

Las operaciones del 8086 pueden dividirse en cuatro clases principales:

1. Instrucciones de transferencia *(movement)* de datos, donde se incluyen transferencia, introducir, y sacar *(move, push, pop)*
2. Instrucciones aritméticas y lógicas, donde se incluyen operaciones lógicas, de test, desplazamentos *(shifts)* y operaciones aritméticas decimales y enteras
3. Instrucciones de flujo de control, donde se incluyen saltos condicionales e incondicionales, llamadas y retornos
4. Instrucciones de cadena, donde se incluyen transferencia *(move)* de cadenas y comparación de cadenas

Además, existe un prefijo de repetición que puede preceder a cualquier instrucción de cadenas, que indica que la instrucción deberá repetirse un número de veces igual al valor contenido del registro CX. La Figura 4.11 muestra algunas instrucciones típicas del 8086 y sus funciones.

Las instrucciones de flujo de control han de tener la posibilidad de direccionar destinos en otro segmento. Esto se logra con dos tipos de instrucciones

Instrucción	Función
JE nombre	if equal (CC) {IP←nombre}; IP-128 ≤ nombre < IP+128
JMP nombre	IP←nombre
CALLF nombre,seg	SP←SP-2; M[SS:SP]←IP+5; SP←SP-2; M[SS:SP]←CS; IP←nombre; CS←seg;
MOVW BX,[DI+45]	BX←$_{16}$M[DS:DI+45]
PUSH SI	SP←SP-2; M[SS:SP]←SI
POP DI	DI←M[SS:SP]; SP←SP+2
ADD AX,#6765	AX←AX+6765
SHL BX,1	BX←BX$_{1..15}$ ## 0
TEST DX,#42	Set CC señalizadores con DX & 42
MOVSB	M[ES:DI]←$_{8}$M[DS:SI]; DI←DI+1; SI←SI+1

FIGURA 4.11 Algunas instrucciones típicas del 8086 y sus funciones. En la Figura 4.12 se da una lista de las operaciones más frecuentes. Utilizamos la abreviatura SR:X para indicar la formación de una dirección con el registro segmento SR y el desplazamiento X. La dirección efectiva que corresponde a SR:X es (SR<<4)+X. La CALLF guarda el IP de la siguiente instrucción y el CS actual en la pila.

de flujo de control: transferencias «próximas» para intrasegmentos (en un segmento) y transferencias «lejanas» para intersegmentos (entre segmentos). En las bifurcaciones lejanas, que deben ser incondicionales, a continuación del código de operación van dos cantidades de 16 bits; una se utiliza como puntero de instrucción (IP) mientras que la otra se carga en el CS y se convierte

Instrucción	Significado
Control	**Saltos condicionales e incondicionales**
JNZ, JZ	Salta si condición a IP + desplazamiento de 8 bits; JNE (para JNZ), JE (para JZ) son nombres alternativos
JMP, JMPF	Bifurcación incondicional —versiones de desplazamiento intrasegmento (próxima) de 8 ó 16 bits, e intersegmento (lejana)
CALL, CALLF	Llamada a subrutina —desplazamiento de 16 bits; guarda dirección de retorno en pila; versiones próxima y lejana
RET, RETF	Saca dirección de retorno de la pila y bifurca a ella; versiones próxima y lejana
LOOP	Salto de bucle —decrementa CX; bifurca a IP + desplazamiento de 8 bits si CX $\neq$ 0
Transferencia de datos	**Transferencia de datos entre registros o entre registros y memoria**
MOV	Transferencia entre dos registros o entre registro y memoria
PUSH	Introduce operando en la pila
POP	Saca operando de la cabeza de la pila a un registro
LES	Carga ES y uno de los GPR desde memoria
Aritmética, lógica	**Operaciones aritméticas y lógicas utilizando los registros de datos y memoria**
ADD	Suma fuente a destino; formato registro-memoria
SUB	Resta fuente de destino; formato registro-memoria
CMP	Compara fuente y destino; formato registro-memoria
SHL	Desplazamiento a la izquierda
SHR	Desplazamiento lógico a la derecha
RCR	Rotación a la derecha con acarreo como relleno
CBW	Convierte byte de AL a palabra en AX
TEST	AND lógica de fuente y destino modifica señalizadores
INC	Incrementa destino; formato registro-memoria
DEC	Decrementa destino; formato registro-memoria
OR	OR lógica; formato registro-memoria
XOR	OR exclusiva; formato registro-memoria
Cadena de instrucciones	**Transferencia entre operandos cadena; longitud dada por un prefijo de repetición**
MOVS	Copia de la cadena fuente en la destino; puede repetirse
LODS	Carga un byte o palabra de una cadena en el registro A

FIGURA 4.12 Algunas operaciones típicas en el 8086. Muchas operaciones utilizan formato registro-memoria, donde o bien la fuente o bien el destino pueden ser memoria y el otro operando puede ser un registro o inmediato.

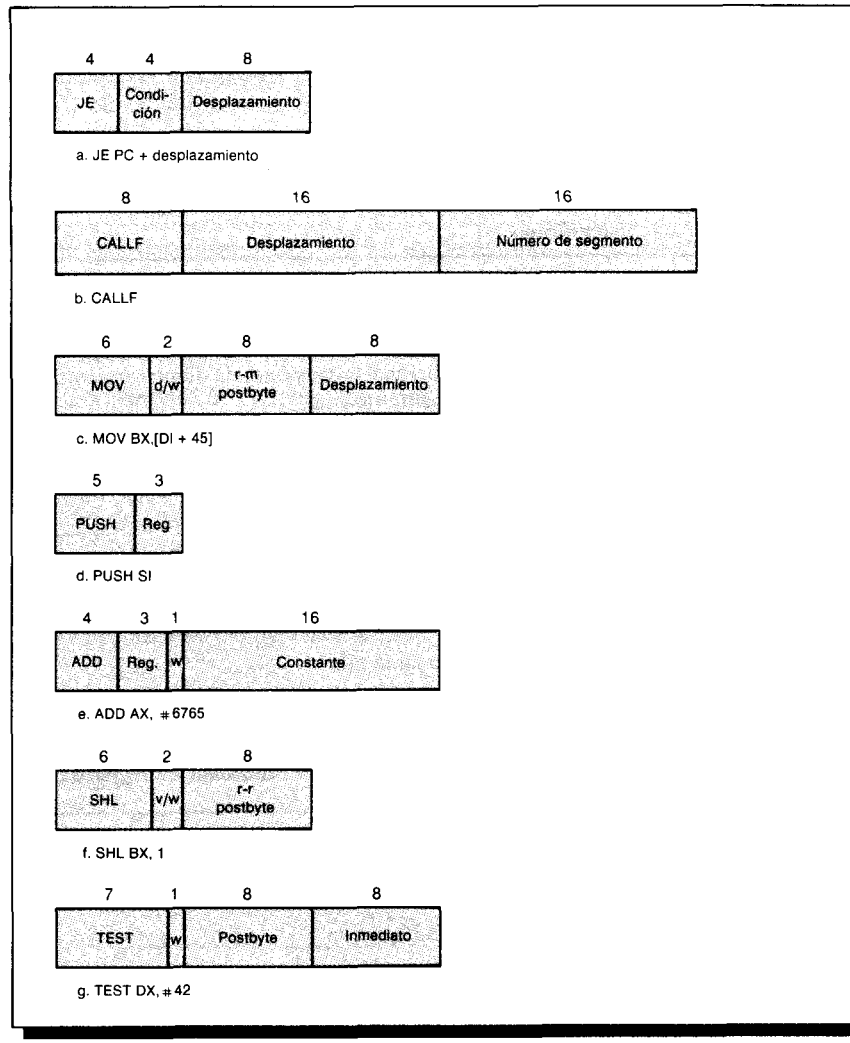


FIGURA 4.13 Formatos de instrucción típicos del 8086. La codificación del postbyte se muestra en la Figura 4.14. Muchas instrucciones contienen el campo w de 1 bit, que indica si la operación es un byte o palabra. Los campos de la forma v/w o d/w son un campo d o campo v seguidos por el campo w. El campo d en MOV se utiliza en instrucciones que pueden transferir a/o desde memoria y muestra la dirección de la transferencia. El campo v en la instrucción SHL indica un desplazamiento de longitud variable; los desplazamientos de longitud variable utilizan un registro que contiene la cuenta del desplazamiento. La instrucción ADD muestra una codificación corta optimizada utilizable sólo cuando el primer operando es AX. Las instrucciones en su conjunto pueden variar de uno a seis bytes de longitud.

en el nuevo segmento de código. Las llamadas y retornos funcionan de manera análoga —una llamada lejana introduce en la pila el puntero de la instrucción de retorno y el segmento de retorno, y carga el puntero de instrucción y el segmento de código. Un retorno lejano saca el puntero de instrucción y el segmento de código de la pila. Los programadores o escritores de compiladores deben asegurarse siempre de utilizar el mismo tipo de llamada y vuelta para un procedimiento —un retorno cercano no funciona con una llamada lejana, y viceversa.

La Figura 4.12 resume las instrucciones más populares del 8086. Muchas están disponibles en formatos de byte y de palabra. En el Apéndice B aparece un listado completo de las instrucciones.

La codificación de las instrucciones en el 8086 es compleja, y hay muchos formatos de instrucción diferentes. Las instrucciones pueden variar desde un byte, cuando no hay operandos, hasta seis bytes, cuando la instrucción contiene inmediatos de 16 bits y utiliza direccionamiento de desplazamiento de 16 bits. La Figura 4.13 muestra el formato de instrucción para algunas instrucciones del ejemplo de la Figura 4.11. El byte del código de operación contiene habitualmente un bit que indica si la instrucción es de una palabra o de un byte. Para algunas instrucciones el código de operación puede incluir el modo de direccionamiento y el registro. Esto es cierto en muchas instrucciones que tienen la forma «registro←registro op inmediato». Para otras instrucciones un «postbyte» o byte extra del código de operación contiene información sobre el modo de direccionamiento. Este postbyte lo utilizan muchas instrucciones que direccionan memoria. La codificación del postbyte aparece en la Figura 4.14. Finalmente hay un byte prefijo que se utiliza para tres propósitos diferentes. Puede anular la utilización del segmento implícito de las

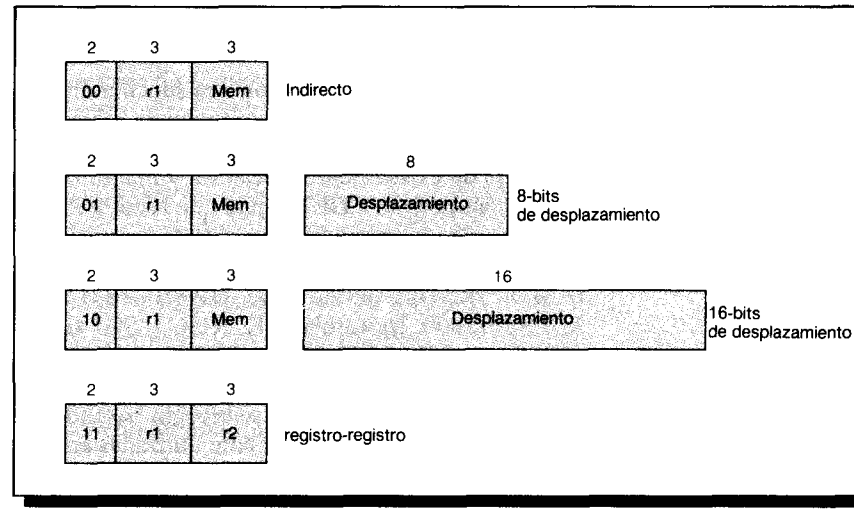


FIGURA 4.14 Hay cuatro codificaciones postbyte en el 8086 designadas por un señalizador de 2 bits. Las tres primeras indican una instrucción registro-memoria, donde Mem es el registro base. La cuarta forma es registro-registro.

instrucciones, y se puede utilizar para repetir una instrucción de cadena el número de veces indicado por el contenido de CX. (Esta última función es útil para instrucciones de cadena que operan con un solo byte o palabra en un instante y utilizan direccionamiento de autoincremento.) Tercero, se puede utilizar para generar un acceso atómico a memoria para utilizarlo en implementar la sincronización.

4.5 La arquitectura DLX

En muchas partes de este libro tendremos ocasión de referirnos a un «lenguaje máquina» del computador. La máquina que utilizamos es un computador mítico denominado «MIX». MIX, probablemente, es muy parecido a la mayoría de computadores existentes, excepto que es, quizás, más agradable... MIX es el primer computador poliinsaturado del mundo. Igual que muchas máquinas, tiene un número de identificación —el 1009. Este número se calculó tomando 16 computadores reales muy similares a MIX y en los que MIX se puede simular fácilmente, promediando entonces sus números con igual peso:

[(360 + 650 + 709 + 7070 + U3 + SS80 + 1107 + 1604 + G20 + B220 + S2000 + 920 + 601 + H800 + PDP-4 + 11)/16] = 1009

El mismo número puede obtenerse de una forma más sencilla empleando números romanos.

Donald Knuth, *The Art of Computer Programming. Volume I: Fundamental Algorithms*

En esta sección describiremos una sencilla arquitectura de carga/almacenamiento denominada DLX (pronunciada «Deluxe»). Los autores piensan que DLX es el segundo computador poliinsaturado del mundo —el promedio de una serie de experimentos recientes y máquinas comerciales que son muy similares en filosofía a DLX. Igual que Knuth, obtuvimos el nombre de nuestra máquina de un promedio expresado en números romanos:

(AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260)/13 = 560 = DLX

La arquitectura de DLX se escogió basándose en las observaciones sobre las primitivas más frecuentemente utilizadas en los programas. Funciones más sofisticadas (y menos rendimiento-crítico) se implementan en software con instrucciones múltiples. En la Sección 4.9 explicamos cómo y por qué estas arquitecturas llegaron a ser populares.

Igual que, en las máquinas de carga/almacenamiento más recientes, DLX hace énfasis en

- Un sencillo repertorio de instrucciones de carga/almacenamiento
- Diseño de segmentación *(pipelining)* eficiente (explicado en el Capítulo 6)
- Un repertorio de instrucciones fácilmente decodificables
- Eficiencia como objeto del compilador

DLX proporciona un buen modelo arquitectónico para su estudio, no sólo debido a la reciente popularidad de este tipo de máquinas, sino también porque es una arquitectura fácil de comprender.

DLX: nuestra arquitectura genérica de carga/almacenamiento

En esta sección, se define el repertorio de instrucciones de DLX. De nuevo, utilizaremos esta arquitectura en los Capítulos 5 a 7, y es la base de una serie de ejercicios y proyectos de programación.

- La arquitectura tiene 32 registros de propósito general GPR de 32 bits; el valor de R0 siempre es 0. Adicionalmente, hay un conjunto de registros de punto flotante (FPR), que se pueden utilizar como 32 registros de simple precisión (32 bits), o como parejas par-impar que contienen valores de doble precisión. Así, los registros de punto flotante de 64 bits se denominan F0, F2,..., F28, F30. Se pueden realizar operaciones en simple y doble precisión. Hay un conjunto de registros especiales utilizados para acceder a la información sobre el estado. El registro de estado de FP (punto flotante) se utiliza para comparaciones y excepciones de FP. Todas las transferencias al y del registro de estado se realizan a través de los GPR; hay un salto que examina el bit de comparación del registro de estado de FP.
- La memoria es direccionable por bytes en el modo «Big Endian» con una dirección de 32 bits. Todas las referencias a memoria se realizan a través de cargas o almacenamientos entre memoria y los GPR o FPR. Los accesos que involucran los GPR pueden realizarse a un byte, a media palabra, o a una palabra. Los FPR se pueden cargar y almacenar con palabras en simple o doble precisión (utilizando un par de registros para DP). Todos los accesos a memoria deben estar alineados. También hay instrucciones para transferencia *(move)* entre un FPR y un GPR.
- Todas las instrucciones son de 32 bits y deben estar alineadas.
- Hay también unos pocos registros especiales que se pueden transferir a y desde registros enteros. Un ejemplo es el registro de estado de punto flotante, utilizado para almacenar la información sobre los resultados de las operaciones en punto flotante.

Operaciones

Hay cuatro clases de instrucciones: cargas y almacenamientos, operaciones ALU, saltos y bifurcaciones y operaciones en punto flotante.

Cualquiera de los registros de propósito general o de punto flotante se pueden cargar o alamacenar, excepto cargar R0 que no tiene efecto. Hay un modo único de direccionamiento, registro base + desplazamiento de 16 bits con signo. Las cargas de media palabra y de byte ubican el objeto cargado en la parte inferior del registro. La parte superior del registro se rellena, bien con la extensión del signo de los valores cargados o con ceros, dependiendo del código de operación. Los números en punto flotante y simple precisión ocupan un registro de punto flotante, mientras que los valores en doble precisión ocupan un par. Las conversiones entre simple y doble precisión deben realizarse explícitamente. El formato de punto flotante es el del IEEE 754 (ver Apéndice A). La Figura 4.15 da un ejemplo de las instrucciones de carga y almacenamiento. Una lista completa de las instrucciones aparece en la Figura 4.18.

Todas las instrucciones de la ALU son instrucciones de registro-registro. Las operaciones incluyen operaciones aritméticas sencillas y operaciones lógicas: suma, resta, AND, OR, XOR, y desplazamientos *(shifts)*. Se proporcionan las formas inmediatas de todas estas instrucciones, con un inmediato con signo-extendido a 16 bits. La operación `LHI` (carga superior inmediato) carga la mitad superior de un registro, mientras pone a 0 la mitad inferior. Esto permite que, con dos instrucciones, se construya una constante completa de 32 bits.

(A veces utilizamos el nemotécnico `LI`, que significa carga inmediato, como una abreviatura de una suma inmediata donde uno de los operandos fuente

Instrucción ejemplo	Nombre de la instrucción	Significado
`LW R1,30(R2)`	Cargar palabra	$R1 \leftarrow_{32} M[30+R2]$
`LW R1,1000(R0)`	Cargar palabra	$R1 \leftarrow_{32} M[1000+0]$
`LB R1,40(R3)`	Cargar byte	$R1 \leftarrow_{32} (M[40+R3]_0)^{24} \# \# M[40+R3]$
`LBU R1,40(R3)`	Cargar byte sin signo	$R1 \leftarrow_{32} 0^{24} \# \# M[40+R3]$
`LH R1,40(R3)`	Cargar media palabra	$R1 \leftarrow_{32} (M[40+R3]_0)^{16} \# \# M[40+R3] \# \# M[41+R3]$
`LF F0,50(R3)`	Cargar flotante	$F0 \leftarrow_{32} M[50+R3]$
`LD F0,50(R2)`	Cargar doble	$F0 \# \# F1 \leftarrow_{64} M[50+R2]$
`SW 500(R4),R3`	Almacenar palabra	$M[500+R4] \leftarrow_{32} R3$
`SF 40(R3),F0`	Almacenar flotante	$M[40+R3] \leftarrow_{32} F0$
`SD 40(R3),F0`	Almacenar doble	$M[40+R3] \leftarrow_{32} F0$; $M[44+R3] \leftarrow_{32} F1$
`SH 502(R2),R3`	Almacenar media	$M[502+R2] \leftarrow_{16} R3_{16..31}$
`SB 41(R3),R2`	Almacenar byte	$M[41+R3] \leftarrow_{8} R2_{24..31}$

FIGURA 4.15 Las instrucciones de carga y almacenamiento en DLX. Todas utilizan un modo único de direccionamiento y requieren que el valor de memoria esté alineado. Por supuesto, ambas carga y almacenamiento están disponibles para todos los tipos de datos mostrados.

Instrucción ejemplo	Nombre de la instrucción	Significado
`ADD R1,R2,R3`	Suma	`R1←R2+R3`
`ADDI R1,R2,#3`	Suma inmediato	`R1←R2+3`
`LHI R1,#42`	Cargar alto inmediato	`R1←42##0`^{16}
`SLL R1,R2,#5`	Desplazamiento lógico a la izquierda	`R1←R2<<5`
`SLT R1,R2,R3`	Inicializar menor que	`if (R2<R3) R1←1 else R1←0`

FIGURA 4.16 Ejemplos de instrucciones aritméticas/lógicas en DLX, ambas con y sin inmediatos.

es R0; de igual forma, el nemotécnico `MOV` a veces se utiliza para una suma (`ADD`) donde una de las fuentes es R0.)

También hay instrucciones de comparación, que comparan dos registros ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$). Si la condición es cierta, estas instrucciones colocan un 1 en el registro destino (para representar verdadero-*true*); en otro caso colocan el valor 0. Debido a que estas operaciones «inicializan» un registro se denominan inicializa-igual, inicializa-no-igual, inicializa-menor que, etc. También hay formas inmediatas para estas comparaciones. La Figura 4.16 da algunos ejemplos de instrucciones aritméticas/lógicas.

El control se realiza mediante un conjunto de bifurcaciones y saltos. Las tres instrucciones de bifurcación están diferenciadas por las dos formas de especificar la dirección destino y por si existe o no existe enlace. Dos bifurcaciones utilizan un desplazamiento *(offset)* con signo de 26 bits sumado al contador de programa (de la instrucción que sigue secuencialmente la bifurcación) para determinar la dirección destino; las otras dos instrucciones de bifurcación especifican un registro que contiene la dirección destino. Hay dos tipos de bifurcación: bifurcación simple, y bifurcación y enlace (utilizada para llamadas a procedimientos). La última coloca la dirección de vuelta en R31.

Todos los saltos son condicionales. La condición de salto se especifica en la instrucción, que puede examinar el registro fuente para compararlo cero o no cero; este puede ser el valor de un dato o el resultado de una comparación. La dirección destino del salto se especifica con un desplazamiento con signo de 16 bits que se suma al contador de programa. La Figura 4.17 da algunas instrucciones típicas de bifurcación y de salto.

Las instrucciones de punto flotante manipulan los registros de punto flotante e indican si la operación a realizar es en simple o doble precisión. Las operaciones en simple precisión se pueden aplicar a cualquiera de los registros, mientras que las operaciones en doble precisión se aplican sólo a una pareja par-impar (por ejemplo, F4, F5), que se designa por el número de registro par. Las instrucciones de carga y almacenamiento, para los registros de punto flotante, transfieren *(move)* datos entre los registros de punto flotante y memoria en simple y doble precisión. Las operaciones `MOVF` y `MOVD` copian, en punto flotante y simple precisión (`MOVF`) o doble precisión (`MOVD`), un re-

Instrucción ejemplo	Nombre de la instrucción	Significado
J nombre	Bifurcación	PC←nombre; ((PC+4)−2^{25}) ≤ nombre < ((PC+4)+2^{25})
JAL nombre	Bifurcación y enlace	R31←PC+4; PC←nombre; ((PC+4)−2^{25}) ≤ nombre < ((PC+4)+2^{25})
JALR R2	Bifurcación y enlaza registro	R31←PC+4; PC←R2
JR R3	Bifurcación a registro	PC←R3
BEQZ R4,nombre	Salta igual a cero	if (R4==0) PC←nombre; ((PC+4)−2^{15}) ≤ nombre < ((PC+4)+2^{15})
BNEZ R4,nombre	Salta no igual a cero	if (R4!=0) PC←nombre; ((PC+4)−2^{15}) ≤ nombre < ((PC+4)+2^{15})

FIGURA 4.17 Instrucciones típicas de flujo de control en DLX. Todas las instrucciones de control, excepto las bifurcaciones a una dirección en un registro son relativas al PC. Si el registro operando es R0, el salto es incondicional, pero el compilador habitualmente preferirá utilizar una bifurcación con un desplazamiento mayor sobre este «salto incondicional».

gistro en otro registro del mismo tipo. Las operaciones MOVFP2I y MOVI2FP transfieren datos entre un registro en punto flotante y un registro entero; transferir un valor en doble precisión a dos registros enteros requiere dos instrucciones. También existen multiplicaciones y divisiones enteras que funcionan en registros de punto flotante de 32 bits, así como conversiones de entero a punto flotante y viceversa.

Las operaciones de punto flotante son suma, resta, multiplicación, y división; se utiliza el sufijo D para doble precisión y el F para simple precisión (por ejemplo, ADDD, ADDF, SUBD, SUBF, MULTD, MULTF, DIVD, DIVF). Las comparaciones en punto flotante inicializan un bit del registro de estado especial de punto flotante que puede ser examinado por un par de saltos: BFPT y BFPF, salto en punto flotante cierto y salto en punto flotante falso.

La Figura 4.18 contiene una lista de todas las operaciones y su significado.

Formatos de las instrucciones

Todas las instrucciones son de 32 bits con un código de operación principal de 6 bits. La Figura 4.19 muestra la organización de las instrucciones.

Máquinas próximas a DLX

Entre 1985 y 1990 se anunciaron muchas máquinas de carga/almacenamiento que eran similares a DLX. La Figura 4.20 describe las características principales de estas máquinas. Todas tienen instrucciones de 32 bits y son arquitecturas de carga/almacenamiento. En la figura aparecen sus diferencias.

Tipo de instrucción/Código de op.	Significado de la instrucción
Transferencias de datos	**Transfiere datos entre registros y memoria, o entre registros enteros y FP o registros especiales; sólo el modo de direccionamiento de memoria es un desplazamiento de 16 bits + contenido de un GPR**
LB,LBU,SB	Carga byte, carga byte sin signo, almacena byte
LH,LHU,SH	Carga media palabra, carga media palabra sin signo, almacena media palabra
LW,SW	Carga palabra, almacena palabra (a/desde registros enteros)
LF,LD,SF,SD	Carga punto flotante SP, carga punto flotante DP, almacena punto flotante SP, almacena punto flotante DP
MOVI2S, MOVS2I	Transfiere desde/a GPR a/desde un registro especial
MOVF, MOVD	Copia un registro de punto flotante o un par en DP en otro registro o par
MOVFP2I,MOVI2FP	Transfiere 32 bits desde/a registros FP a/desde registros enteros
Aritmética/Lógica	**Operaciones sobre datos enteros o lógicos en GPR/s; la aritmética con signo causa un trap en caso de desbordamiento**
ADD,ADDI,ADDU,ADDUI	Suma, suma inmediato (todos los inmediatos son de 16 bits); con signo y sin signo
SUB,SUBI,SUBU,SUBUI	Resta, resta inmediato; con signo y sin signo
MULT,MULTU,DIV,DIVU	Multiplica y divide, con signo y sin signo; los operandos deben estar en registros de punto flotante; todas las operaciones tienen valores de 32 bits
AND,ANDI	And, and inmediato
OR,ORI,XOR,XORI	Or, or inmediato, or exclusiva, or exclusiva inmediata
LHI	Carga inmediato superior: carga la mitad superior de registro con inmediato
SLL, SRL, SRA, SLLI, SRLI, SRAI	Desplazamientos: ambos inmediatos (S__I) y forma variable (S__); los desplazamientos son desplazamientos lógicos a la izquierda, lógicos a la derecha, aritméticos a la derecha
S___,S___I	Inicialización condicional: «__» puede ser LT, GT, LE, GE, EQ, NE
Control	**Saltos y bifurcaciones condicionales; relativos al PC o mediante registros**
BEQZ,BNEZ	Salto GPR igual/no igual a cero; desplazamiento de 16 bits desde PC+4
BFPT,BFPF	Test de bit de comparación en el registro de estado FP y salto; desplazamiento de 16 bits desde PC+4
J, JR	Bifurcaciones: desplazamiento de 26 bits desde PC (J) o destino en registro (JR)
JAL, JALR	Bifurcación y enlace: guarda PC+4 en R31, el destino es relativo al PC (JAL) o un registro (JALR)

Tipo de instrucción/Código de op.	Significado de la instrucción
TRAP	Transfiere a sistema operativo a una dirección vectorizada; ver Capítulo 5
RFE	Volver al código del usuario desde una excepción; restaurar modo de usuario; ver Capítulo 5
Punto flotante	**Operaciones en punto flotante en formatos DP y SP**
ADDD,ADDF	Suma números DP, SP
SUBD,SUBF	Resta números DP, SP
MULTD,MULTF	Multiplica punto flotante DP, SP
DIVD,DIVF	Divide punto flotante DP, SP
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convierte instrucciones: CVTx2y convierte de tipo x a y, donde x e y pueden ser uno de: I (Entero), D (Doble precisión), o F (Simple precisión). Ambos operandos están en los registros FP
____D,____F	Compara DP y SP: «__» puede ser LT, GT, LE, EQ, NE; pone bit de comparación en registro de estado FP

FIGURA 4.18 Lista completa de las instrucciones en DLX. Los formatos de estas instrucciones se muestran en la Figura 4.19. Esta lista puede encontrarse también en la contraportada posterior.

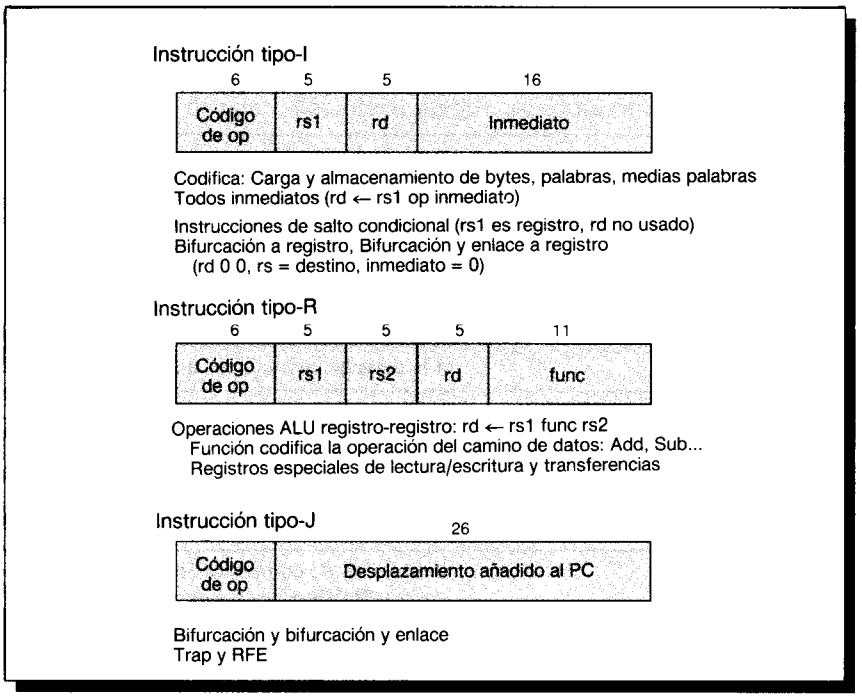


FIGURA 4.19 Formato de las instrucciones para DLX. Todas las instrucciones se codifican en uno de los tres tipos.

Máquina	Registros	Modos de direccionamiento	Operaciones
DLX	32 enteros; 16 DP o 32 SP FP	Desplazamiento de 16 bits; inmediato de 16 bits	Ver Figura 4.18
AMD 29000	192 enteros con cache de pila; 8 DP FP	Solamente registro diferido; inmediato de 8 bits	Trap de multiplicación/división entera para software. Saltos =, ≠ 0 sólo.
HP Precision Architecture	32 GPR	Desplazamiento de 5, 14 y 32 bits; modo escalado (sólo carga); autoincremento; autodecremento	Cada operación ALU puede saltar a la siguiente instrucción. Muchas instrucciones especiales de manipulación de bits. Inmediatos de 32 bits; instrucciones que soportan decimales: multiplicación/división no en instrucción única. Almacenamientos de palabra parcial. Direcciones de 64 bits posibles a través de la segmentación.
Intel i860	32 enteros; 16 DP o 32 SP FP	Desplazamiento de 16 bits, modo indexado; autoincremento; inmediato de 16 bits	Salto compara dos registros para igualdad. Están soportadas las traps condicionales. FP recíproco en lugar de dividir. Algún soporte para cargas y almacenamientos de 128 bits.
MIPS R2000 / R3000	32 enteros; 16 FP	Desplazamiento de 16 bits; inmediato de 16 bits	Carga/almacenamiento de punto flotante transfiere 32 bits a un registro del par. La condición de salto puede comparar dos registros. Multiplicación y división entera en los GPR. Instrucciones especiales para carga/almacenamiento de palabra parcial.
Motorola 88000	32 GPR	Desplazamiento de 16 bits; modo indexado	Instrucciones especiales de manipulación de bits. Los saltos pueden examinar (test) para cero y también bits de test inicializados por comparaciones.
SPARC	Ventanas de registros con 32 registros enteros disponibles por procedimiento 16 DP o 32 SP FP	Desplazamiento de 13 bits e inmediato de 13 bits; modo de direccionamiento indexado	Los saltos utilizan código de condición, inicializados selectivamente por instrucciones. No instrucciones de multiplicación/división entera. No transferencias entre registros enteros y FP.

FIGURA 4.20 Comparación de las características principales de una serie de arquitecturas recientes de carga/almacenamiento. Todas las máquinas tienen un tamaño básico de instrucción de 32 bits, aunque están comtempladas algunas provisiones para acortar o alargar la instrucción. Por ejemplo, Precisión Architecture utiliza instrucciones de dos palabras para inmediatos largos. Las ventanas de registros y caches de pila, que se utilizan en las arquitecturas SPARC y AMD 29000, se explican en el Capítulo 8. La MIPS R2000 se utiliza en la DECstation 3100, la máquina analizada en el Capítulo 2, y utilizada como máquina de carga/almacenamiento en el Capítulo 3. El número de registros de punto flotante de doble precisión se indica si están separados de los registros enteros. En el Apéndice E se da una descripción comparativa detallada de las arquitecturas DLX, MIPS R2000, SPARC, i860, y 88000. Las arquitecturas MIPS y SPARC tienen extensiones que no estaban soportadas por hardware en la primera implementación. Estas se explican en el Apéndice E.

Estas máquinas son todas muy similares. Si no se está convencido, intentar la realización de una tabla como ésta comparando estas máquinas con el VAX o el 8086.

DLX tiene un fuerte parecido con todas las demás máquinas de carga/almacenamiento que aparecen en la Figura 4.20. (Ver Apéndice E para una descripción detallada de las cuatro máquinas de carga/almacenamiento muy parecidas a DLX.) Así, las medidas de la siguiente sección serán aproximaciones razonables del comportamiento de cualquiera de las máquinas. En efecto, algunos estudios sugieren que entre estas máquinas son más significativas las diferencias de los compiladores que las diferencias de las arquitecturas.

4.6 Juntando todo: medidas de utilización del repertorio de instrucciones

En esta sección examinamos el uso dinámico de los cuatro repertorios de instrucciones presentados en este capítulo. Todas las instrucciones responsables del 1,5 por 100 o más de las ejecuciones de las instrucciones, en un conjunto de benchmarks, se incluyen en las medidas de cada arquitectura. En aras de la concisión, los porcentajes fraccionarios se redondean para que todas las entradas en los gráficos, de la frecuencia de códigos de operación, sean como mínimo el 2 por 100.

Para facilitar comparaciones entre medidas dinámicas del repertorio de instrucciones, las medidas se organizan por clase de aplicación. La Figura 4.21 muestra estas clases de aplicación y los programas utilizados para obtener los datos de uso-de-instrucción en cada una de las máquinas explicadas. A veces, compararemos datos para diferentes arquitecturas ejecutando el mismo tipo de aplicación (por ejemplo, un compilador) **pero** diferentes programas. Se advierte al lector que estas comparaciones deben hacerse cuidadosamente y con

Máquinas	Compiladores	Punto flotante	Enteros general	Procesamiento de datos comerciales
VAX	GCC	Spice	TeX	COBOLX
360	PL/I	FORTGO	PLIGO	COBOLGO
8086	Turbo C		Assembler	Lotus 1-2-3
DLX	GCC	Spice	TeX	US Steel

FIGURA 4.21 Programas usados para obtener información sobre mezclas de instrucciones. Hay cuatro tipos de cargas de trabajo, y cada uno tiene un programa de representación —excepto que no hay programa de punto flotante para el 8086. Las entradas a GCC, Spice y TeX utilizadas por el VAX se acortaron intencionadamente porque el proceso de medición necesita mucho tiempo. (Los lectores que obtengan medidas para el 360 o el 8086 corriendo GCC, Spice o TeX y quieran compartir sus datos pueden contactar con el editor.)

limitaciones sustanciales. A pesar de que ambos programas puedan ser el mismo tipo de aplicación, diferencias en el lenguaje de programación, estilo de codificación, compiladores, etc., pueden afectar sustancialmente a los resultados.

En esta sección presentamos las medidas de las mezclas de instrucciones utilizando un diagrama para cada máquina. El diagrama muestra el uso medio de una instrucción en los programas medidos para esa arquitectura. Las medidas individuales, detalladas para cada programa, pueden encontrarse en el Apéndice C. Este apéndice será necesario como referencia para hacer los ejercicios y ejemplos de este capítulo.

Recordar que estas medidas dependen de los benchmarks elegidos y de la tecnología del compilador utilizado. Aunque los autores piensan que las medidas de esta sección son razonablemente indicativas del uso de estas cuatro arquitecturas, otros programas pueden tener un comportamiento diferente de los aquí presentados, y compiladores diferentes pueden dar resultados diferentes. Al hacer un estudio real del repertorio de instrucciones, al arquitecto le gustaría tener un mayor conjunto de «benchmarks», para abarcar el mayor número posible de aplicaciones. También le gustaría considerar el sistema operativo y su utilización del repertorio de instrucciones. Los benchmarks mono-usuario como los medidos aquí no se comportan necesariamente de la misma manera que el sistema operativo.

Medidas de repertorio de instrucciones VAX

En esta sección los datos sobre la utilización del repertorio de instrucciones VAX provienen principalmente de medidas sobre nuestros tres programas de benchmark. Añadimos los datos obtenidos en otro estudio para COBOL cuando explicamos las distribuciones de los códigos de operación. Para estas medidas, Spice y TeX se compilaron con las versiones de optimización global de los compiladores VAX desarrollados originalmente para VMS [llamados VCC y *fort*]. GCC no puede ser compilado por el compilador vcc y por consiguiente utiliza el compilador estándar VAX cc, que realiza solamente optimizaciones locales. Una vez compilados, estos programas se ejecutaron con el bit de traza activado. Esto hace que el programa se detenga en la ejecución de cada instrucción, permitiendo que un programa de medida recolecte datos. Como esto ralentiza el programa por un factor entre *1 000* y *10 000*, se utilizaron entradas más pequeñas para los programas GCC, TeX y Spice.

Utilización de los modos de direccionamiento

Comenzamos examinando los modos de direccionamiento del VAX, ya que la elección de los modos de direccionamiento y operaciones son ortogonales. Primero, descomponemos las referencias en tres clases: modos de direccionamiento de registro, inmediato (incluyendo literal corto), y de memoria. La Figura 4.22 muestra estas tres clases para nuestros benchmarks. En los tres programas, más de la mitad de las referencias de operandos son a registros.

Aproximadamente, una tercera parte de los operadores del VAX son referencias a memoria. ¿Cómo se especifican las posiciones de memoria? Los mo-

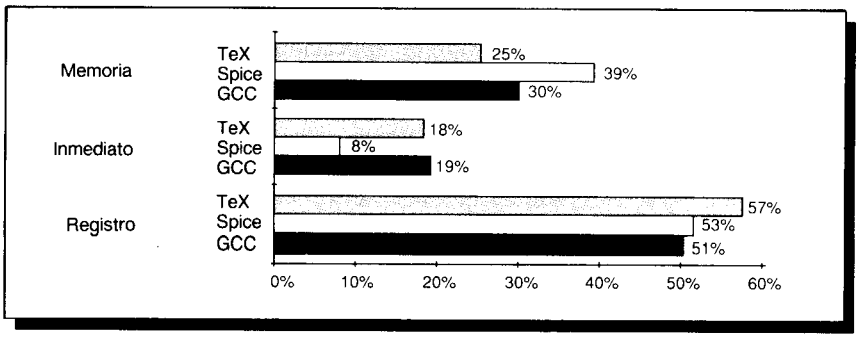


FIGURA 4.22 Descomposición de los tipos básicos de operandos para los tres benchmarks en el VAX. Las frecuencias son muy similares en los programas, excepto por el bajo uso de inmediatos por Spice y su correspondiente mayor uso de operandos de memoria. Esto se debe probablemente a las pocas constantes de punto flotante que se almacenan como inmediatos, pero en su lugar son accedidas desde memoria. Un operando se cuenta por el número de veces que aparece en una instrucción, en vez de por el número de referencias. Por tanto, la instrucción `ADDL2 R1, 45 (R2)` cuenta como una referencia a memoria y una referencia a registro. Los modos de direcciones de memoria de la Figura 4.23 se cuentan de la misma forma. Wiecek [1982] informa que aproximadamente el 90 por 100 de los accesos de operandos son o una lectura o una escritura; y aproximadamente en el 10 por 100 de los accesos ambos escriben y leen el mismo operando (tal como `R1` en la `ADDL2`).

dos de direccionamiento de memoria del VAX caen en tres clases separadas: direccionamiento basado en el PC, direccionamiento escalado, y los demás modos de direccionamiento (a veces denominados modos de direccionamiento generales). El uso principal del direccionamiento basado en el PC es especificar los destinos de los saltos, en lugar de operandos de datos; por ello, no incluimos aquí este modo de direccionamiento. El modo escalado se considera como un modo de direccionamiento separado, y el modo basado (registro base) sobre el que se construye se considera también. La Figura 4.23 muestra el uso de los modos de direccionamiento en los tres programas de benchmark. Sin muchas sorpresas, domina el modo de desplazamiento. Tomados juntos, el desplazamiento y registro diferido, que es esencialmente un caso especial de desplazamiento con un valor de constante cero, constituyen del 70 al 96 por 100 de los modos de direccionamiento que se presentan dinámicamente.

El tamaño de una instrucción VAX es casi siempre de un byte para el código de operación más el número de bytes de los modos de direccionamiento. A partir de estos datos puede estimarse el tamaño medio de una instrucción. Los arquitectos hacen con frecuencia este tipo de estimación cuando no disponen de medidas exactas. Esto es particularmente cierto cuando la recolección de datos es cara. Por ejemplo, para reunir los datos del VAX de este capítulo, fueron necesarios de uno a varios días de tiempo de ejecución, por cada programa.

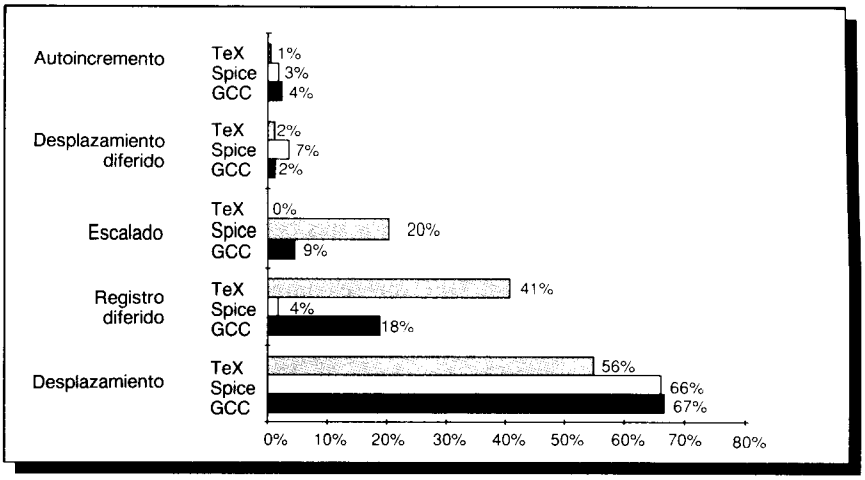


FIGURA 4.23 Uso de los modos de direccionamiento de memoria del VAX, que contabilizan aproximadamente el 31 por 100 de las referencias a operandos en los tres programas. De nuevo Spice destaca a causa de la baja frecuencia de registros diferidos. En Spice, los valores de desplazamiento distintos de cero se presentan con mucha más frecuencia. El uso de arrays en lugar de punteros probablemente lo influencie. De igual forma, Spice utiliza el modo escalado para acceder a los elementos del array. El modo de deplazamiento diferido se utiliza para acceder a los parámetros reales de una subrutina FORTRAN. Recordar que el direccionamiento basado en el PC no se incluye aquí —el uso del direccionamiento basado en PC se puede medir por la frecuencia de los saltos.

Ejemplo

La instrucción media VAX tiene 1,8 operandos. Utilizar este hecho y los datos de los tamaños de los desplazamientos de la Figura 3.13 (Cap. 3) para estimar el tamaño medio de una instrucción VAX. Esta estimación es útil para determinar el ancho de banda de memoria por instrucción, un parámetro de diseño crítico.

Respuesta

A partir de los datos anteriores sabemos que los modos de registro y literal, cada uno de los cuales necesita 1 byte, dominan la mezcla. El modo de direccionamiento más intensamente utilizado, el modo de desplazamiento, puede variar de 2 a 5 bytes —el byte del registro más 1 o más bytes de desplazamiento. En base a la información sobre longitudes de la Figura 3.13 suponemos que el desplazamiento medio es de 1,5 bytes, para un tamaño total de 2,5 bytes para el modo de direccionamiento. Para este ejemplo, suponemos que los modos literal, registro y de desplazamiento constituyen todos los accesos.

Esto significa que hay 1 byte para el código de operación, 1 byte para el modo registro o literal, y aproximadamente 2,5 bytes para el modo de desplazamiento. Utilizando 1,8 operandos por instrucción y las frecuencias medias de acceso de la Figura 4.22, obtenemos

$$1 + 1{,}8 \cdot (0{,}54 + 0{,}15 + 0{,}31 \cdot 2{,}5) \text{ o sea } 3{,}64 \text{ bytes.}$$

Wiecek [1982] midió 3,8 bytes por instrucción. Medidas directas de nuestros tres programas muestran que los tamaños medios son 3,6, 4,9, y 4,2 para GCC, Spice, y TeX, respectivamente.

Mezclas de instrucciones

Ahora, examinamos la distribución de las operaciones de las instrucciones, utilizando nuestros tres benchmarks más el programa COBOLX del estudio publicado por Clark y Levy [1982]. COBOLX es un benchmark interno y sintético de DEC que se compiló con el compilador VAX VMS COBOL y utiliza instrucciones decimales. Sin embargo, los nuevos compiladores de DEC para el VAX evitan utilizar el repertorio de instrucciones decimales, ya que la mayor parte de esa arquitectura está emulada en software —y por tanto es mucho más lenta— en las VAX más modernas basadas en VLSI.

Los datos de esta sección se presentan en forma de diagrama, pero en el Apéndice C se dan tablas detalladas para cada máquina y benchmark. Aquí, los datos hacen énfasis en la frecuencia de instrucciones, pero no siempre coinciden las distribuciones de frecuencia y de tiempo. En la siguiente sección veremos un ejemplo. El Apéndice D contiene un conjunto de medidas detalladas basadas en medidas de distribución de tiempos.

La Figura 4.24 muestra todas las instrucciones responsables, de más del 1,5 por 100, de las ejecuciones dinámicas de instrucciones en todos los benchmarks. Cada barra completa muestra una mezcla media de instrucciones sobre los cuatro programas, y cómo los programas contribuyen a esa mezcla.

GCC y TeX son muy similares en comportamiento; la mayor diferencia es la frecuencia más alta de transferencias de datos de TeX. Spice y COBOLX parecen muy diferentes. Cada uno ejecuta más del 20 por 100 de sus instrucciones utilizando una parte del repertorio de instrucciones que los otros benchmarks apenas utilizan. Tanto COBOLX como Spice realizan muy pocas operaciones aritméticas enteras, utilizando en vez de ello operaciones decimales o en punto flotante. COBOLX utiliza poco las instrucciones de transferencia de datos (4 por 100 frente a un promedio del 20 por 100 de los otros tres programas); en su lugar, el 38 por 100 de las instrucciones que ejecuta son decimales o de cadena.

Las 27 instrucciones de la Figura 4.24 corresponden a una media del 88 por 100 de las instrucciones ejecutadas en los cuatro benchmarks. Sin embargo, la parte final de la distribución es larga y hay muchas instrucciones ejecutadas con una frecuencia del 1/2 al 1 por 100. Por ejemplo, en Spice, las 15 instrucciones más utilizadas constituyen el 90 por 100 de las ejecuciones, y las 26 más utilizadas hasta el 95 por 100. ¡Sin embargo, hay 149 instrucciones VAX diferentes ejecutadas al menos una vez!

Medidas de utilización del repertorio de instrucciones del 360

Las medidas de esta sección están tomadas de las realizadas por Shustek en su tesis Ph.D. [1978]. Este trabajo incluye un estudio de las características dinámicas de siete grandes programas sobre la arquitectura IBM 360. Obtuvo sus

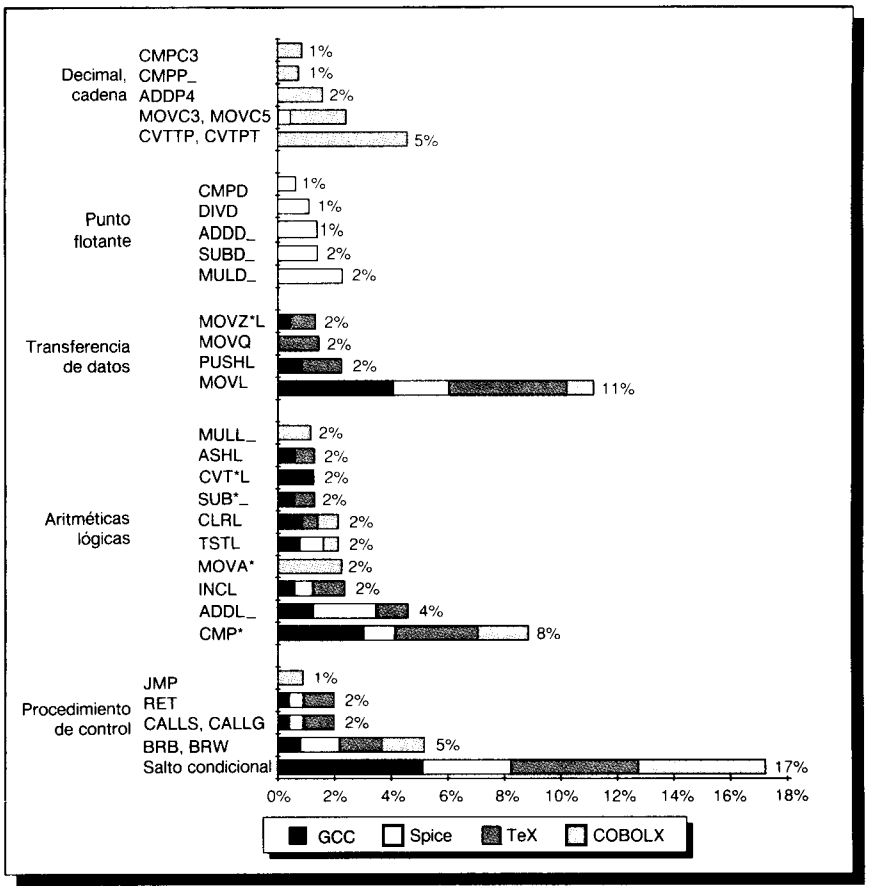


FIGURA 4.24 Las frecuencias de las instrucciones VAX combinadas gráficamente. El tamaño total de cada barra muestra el comportamiento que se vería en una máquina que ejecutase estos cuatro programas con igual frecuencia. Los segmentos de la barra muestran el porcentaje de utilización de esa instrucción que proviene de cada uno de los programas. Esto ilustra por qué algunas partes del repertorio de instrucciones necesitan estar allí para sólo una clase de aplicación. Globalmente, sólo se usa intensamente un pequeño número de instrucciones, aparte de las de control, transferencia de datos e instrucciones aritméticas enteras.

datos construyendo un intérprete para la arquitectura 360. Los cuatro programas descritos en la Figura 4.25 se utilizan en esta sección para examinar las características de utilización del repertorio de instrucciones del 360.

Tipos de instrucción y modos de direccionamiento

La Figura 4.26 muestra la frecuencia de accesos a datos por modos de direccionamiento. El programa COBOL tiene una frecuencia muy alta de accesos a datos. Las transferencias *(moves)* de datos carácter y el uso de datos deci-

males, que siempre residen en memoria, probablemente cuentan para esto. FORTGO tiene, sustancialmente, un número muy bajo de referencias a memoria. Esto puede suceder a causa de la ubicación de las variables en los registros para los bucles internos del programa.

En el 360 solamente hay dos modos de direccionamiento de memoria: registro base + desplazamiento (formato RS, formato SI y formato SS) y registro base + desplazamiento + registro índice (formato RX). Sin embargo, las operaciones disponibles en las instrucciones que direccionan memoria, normalmente, aparecen en un solo formato. Por tanto, probablemente es más útil

Programa	Clase de benchmark	Número de instrucciones	Función del programa
COBOLGO	Business D.P.	3 559 533	Formateador informa uso COBOL
PLIGO	General entero	23 863 497	Contabiliza uso del computador PL/I
FORTGO	Punto flotante	11 719 853	Resolver sistemas lineales FORTRAN
PLIC	Compilador	24 338 101	Compila PL/I

FIGURA 4.25 Cuatro programas utilizados para medir el IBM 360. El sufijo «GO» indica una ejecución de un programa, mientras que el sufijo «C» indica una compilación. Escogemos el compilador de PL/I porque es el mayor y más representativo; también está escrito en PL/I. La tesis de Shustek utilizaba dos ejecuciones FORTRAN. Escogemos LINSYS2 para representar la ejecución FORTRAN, ya que es un programa FORTRAN más típico; nos referiremos a la ejecución de LINSYS2 como FORTGO.

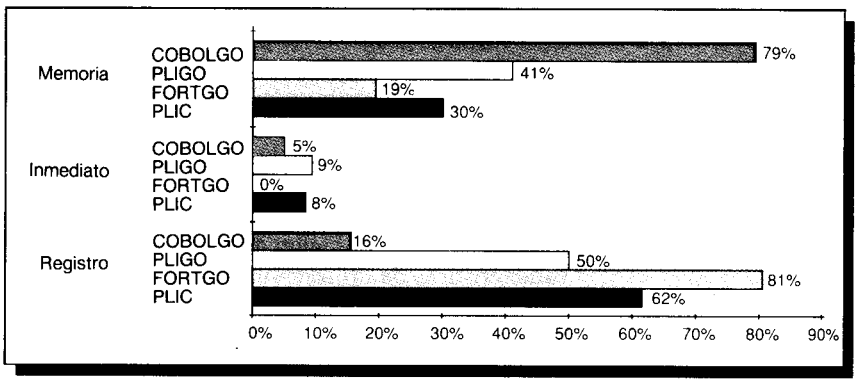


FIGURA 4.26 Distribución de los accesos a operandos hechos por las instrucciones del 360. Soporte limitado para inmediatos es la razón principal de que los inmediatos tengan tan poco uso.

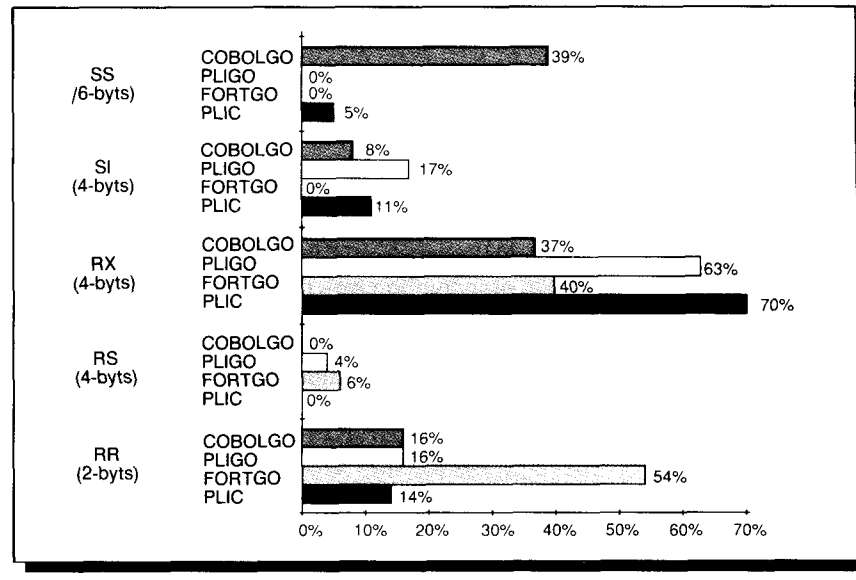


FIGURA 4.27 Porcentaje de instrucciones de 6, 4 y 2 bytes para los cuatro programas del 360. La mayoría de las instrucciones son de 4 bytes, y casi ninguna de 6 bytes, excepto cuando ejecutan COBOLGO.

examinar la utilización del formato de las instrucciones, como se muestra en la Figura 4.27. La mayoría de las instrucciones son de formato RX seguidas de RR. El elevado uso del formato RX no nos debería llevar a concluir que el modelo de direccionamiento de desplazamiento + registro base + registro índice se utiliza ampliamente, ya que en el 85 por 100 de las instrucciones RX el registro índice es cero. COBOL presenta un elevado porcentaje de instrucciones de formato SS, y esto se esperaba porque las instrucciones de cadena y decimales son todas de formato SS. La ejecución FORTRAN muestra un gran porcentaje de formato RR, instrucciones de 2 bytes. Esto tiene sentido en un programa que use intensivamente los registros en sus bucles internos optimizados.

Ejemplo

Dados los datos de la Figura 4.27 calcular la longitud media de instrucción para el programa PLIGO.

Respuesta

La longitud media de instrucción es

$$6 \cdot \%\,\mathrm{SS} + 4 \cdot (\%\,\mathrm{RX} + \%\,\mathrm{RS} + \%\,\mathrm{SI}) + 2 \cdot \%\,\mathrm{RR}$$
$$= 0 + 4 \cdot (0{,}63 + 0{,}04 + 0{,}17) + 2 \cdot 0{,}16 = 3{,}68 \text{ bytes}$$

En los cuatro programas la longitud media medida es de 3,7 bytes.

Mezclas de instrucciones

Ahora, examinaremos los datos para las mezclas de instrucciones. La Figura 4.28 muestra las instrucciones más utilizadas en los cuatro benchmarks del 360. Como ilustra la Figura 4.28, las variaciones entre los programas son muy grandes. El compilador PL/I tiene un número extraordinariamente grande de saltos, mientras que la ejecución de PL/I tiene muy pocos. El uso de operadores aritméticos y lógicos es bastante uniforme con la excepción del programa COBOL, que en su lugar utiliza operaciones decimales.

Comparando estos programas con el VAX, destaca la frecuencia mucho más baja de los saltos —16 por 100 en el 360 frente al 23 por 100 en el VAX. El número de saltos de un programa está en gran parte fijado por el programa, excepto en algunas anomalías arquitectónicas y posibles optimizaciones del compilador (tal como el desenrollamiento de bucles *(loop unrolling)* —expli-

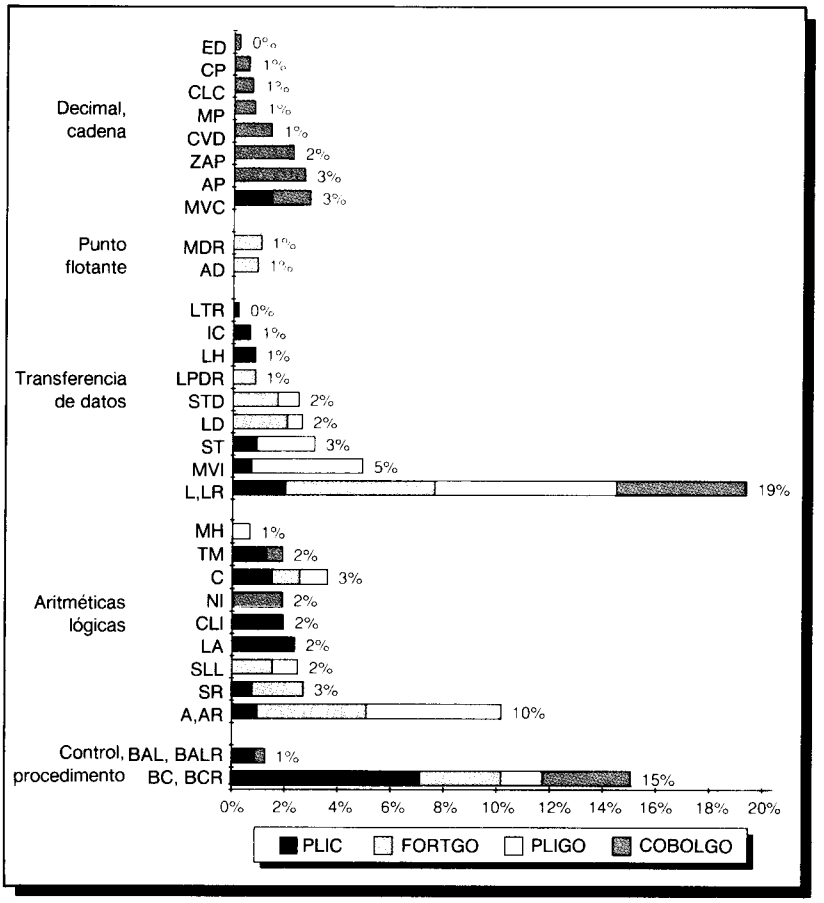


FIGURA 4.28 Datos combinados para los cuatro programas en el 360. Comparar esto con la Figura 4.24, donde están dibujados los datos para el VAX.

cado en el Capítulo 6— pero no utilizado por estos compiladores). Por tanto, el porcentaje de saltos es una medida indirecta de la potencia o densidad de las instrucciones, ya que indica el número de instrucciones que se requieren para cada salto. Podemos esperar que el VAX, con sus modos de direccionamiento más potentes y múltiples operandos de memoria por instrucción, tenga una densidad elevada de instrucciones y una frecuencia mayor de saltos. Vemos, además, la evidencia de mayor densidad de instrucciones del VAX, en la mayor frecuencia de transferencias de datos en el 360 —en el 360 se trasfieren explícitamente más datos en lugar de utilizarlos como operandos de memoria, como en el VAX. Sin embargo, no podemos sacar ninguna conclusión específicamente cuantitativa con respecto a la densidad de instrucciones ya que los programas medidos y los compiladores son diferentes.

También es muy diferente el porcentaje de operaciones de cadenas y carácteres utilizadas por el 360 frente al VAX para las dos aplicaciones COBOL. Finalmente, la ejecución FORTRAN utiliza un número mucho mayor de operaciones enteras en el 360; esto puede deberse a diferencias que surgen cuando el VAX utiliza un modo de direccionamiento, pero el 360 debe utilizar instrucciones explícitas para el cálculo de direcciones.

Como hemos visto, las diferencias en la utilización de instrucciones en el 360 y el VAX son completamente significativas. Las dos arquitecturas siguientes difieren de estas dos primeras aún más espectacularmente.

Medidas de utilización del 8086

Los datos de esta sección fueron tomados de Adams y Zimmerman [1989] en un estudio de siete programas ejecutados en un IBM PC bajo MS DOS 3.1. Obtuvieron los datos ejecutando paso a paso los programas y los recogieron después de la ejecución de cada instrucción, igual que se realizó para la VAX. Los tres programas aquí utilizados, una breve descripción y el número de instrucciones ejecutadas se muestran en la Figura 4.29. Igual que con el VAX y el 360, comenzaremos examinando los accesos a los operandos y modos de direccionamiento y, después, pasaremos a la mezcla de instrucciones.

Programa	Clase de benchmark	Número de instrucciones	Función del programa
Lotus	Comercial	2 904 931	Lotus 1-2-3 calcula cuatro veces una hoja de trabajo de 128 celdas
MASM	Entero general	2 365 711	El Macro Ensamblador de Microsoft ensambla un programa de 500 líneas
Turbo C	Compilador	1 806 143	Turbo C compila Dhrystone

FIGURA 4.29 Tres programas usados para medidas del 8086. Los benchmarks están escritos en una combinación del Ensamblador 8086 y C.

Modos de direccionamiento y longitud de las instrucciones

Nuestra primera medida en el 8086, mostrada en la Figura 4.30, dibuja la procedencia de los operandos. Los inmediatos juegan un papel pequeño, mientras que los accesos a los registros dominan ligeramente sobre los accesos a memoria. Comparado con el VAX, estos programas en el 8086 utilizan una mayor frecuencia de operandos de memoria. El limitado conjunto de registros del 8086 probablemente juega un papel en el incremento del tráfico de memoria, que, sustancialmente, excede el del 360, si ignoramos el programa COBOL (que debe utilizar instrucciones SS) en el 360.

En los programas anteriores, el 41 por 100 de las referencias de operandos son accesos a memoria. La Figura 4.31 muestra la distribución de los modos de direccionamiento para estas referencias a memoria.

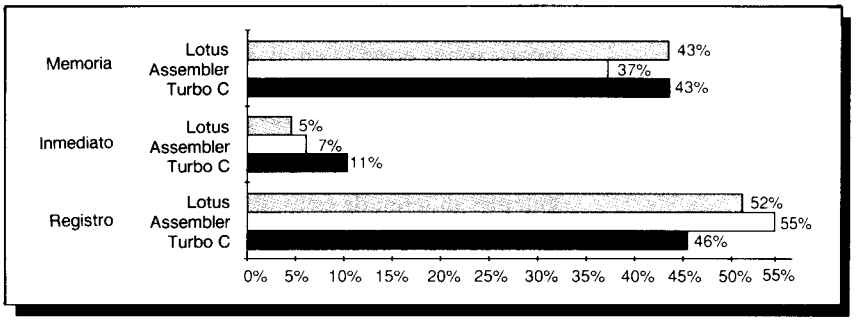


FIGURA 4.30 Tres clases de accesos básicos de operandos en el 8086 y su distribución. El uso implicado del registro acumulador (AX), que se presenta en una serie de instrucciones se cuenta como un acceso a registros.

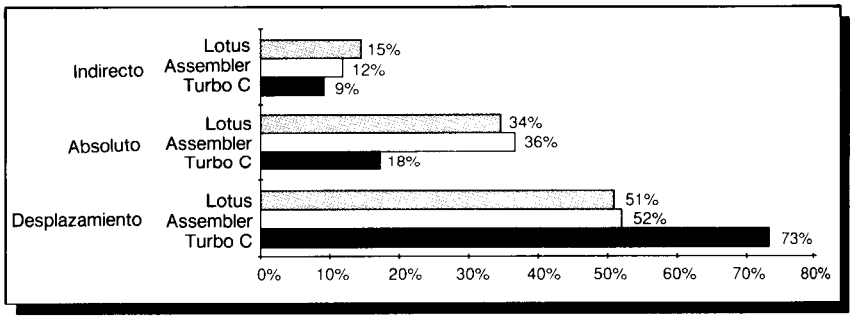


FIGURA 4.31 Los modos de direccionamiento de memoria del 8086 mostrados en este gráfico contabilizan casi todas las referencias de memoria de los tres programas. Los modos de direccionamiento de memoria indexado y basado (registro base) han sido combinados, ya que sus cálculos de direcciones efectivas son los mismos. El modo indirecto de registro está en efecto basado con un desplazamiento cero, equivalente al modo diferido de registro del VAX. Si el indirecto de registro se contabilizase como un modo básico con desplazamiento cero, aproximadamente dos tercios de las referencias de memoria serían modos de desplazamiento. Los otros dos modos restantes no se utilizan esencialmente en los tres programas.

Las instrucciones de longitud variable, uso de registros implícitos, y pequeños tamaños del especificador de registros se combinan para dar una instrucción media muy corta. Para estos tres programas la longitud media de instrucción es aproximadamente 2,5 bytes.

Mezclas de instrucciones en el 8086

Las instrucciones responsables de más del 1,5 por 100 de las ejecuciones, de los tres programas que corren en el 8086, se muestran gráficamente en la Figura 4.32. El subconjunto visualizado del repertorio de instrucciones contabiliza una mayor porporción de todas las ejecuciones de las instrucciones (90 por 100) que en el VAX o en el 360. Como se podía sospechar, las arquitecturas con repertorios de instrucciones más pequeños utilizan un mayor porcentaje de sus códigos de operación.

La principal característica diferenciadora entre los programas, es el cambio de instrucciones de transferencia de datos por instrucciones de control en Lotus. Lotus hace un uso intensivo de la instrucción LOOP, que puede contabilizar ese cambio.

La frecuencia global de las instrucciones de transferencia de datos es mucho mayor en el 8086 que en el VAX. Esta diferencia surge probablemente porque el 8086 tiene menos registros de propósito general. Otras explicaciones posibles incluyen el uso de instrucciones de cadena, que generan una secuen-

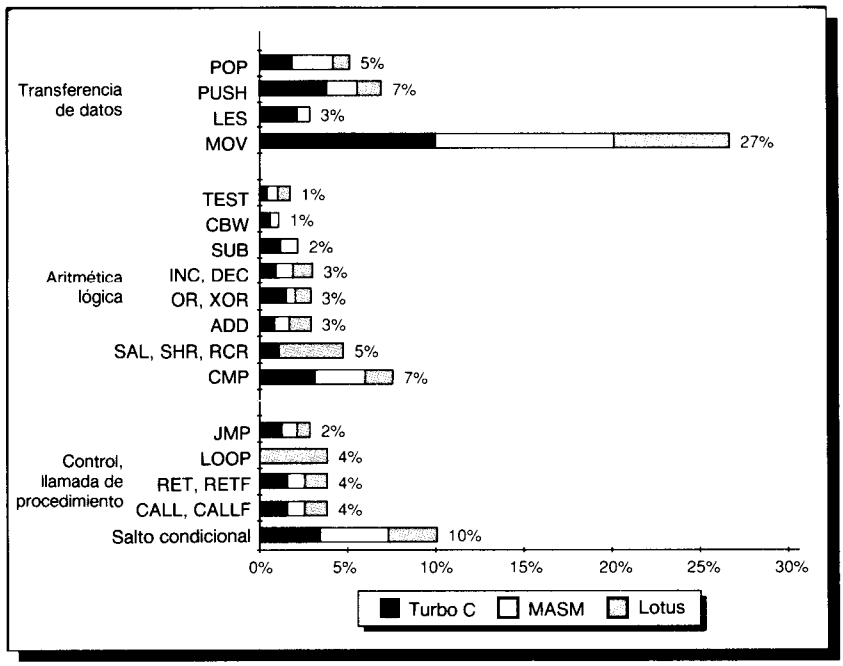


FIGURA 4.32 Distribución de las frecuencias de instrucciones en el 8086 mostradas en el mismo formato utilizado para el VAX y el 360.

cia de instrucciones de transferencia y, transferencias explícitas de datos entre segmentos para facilidad de tratamiento. La frecuencia total de saltos no es muy diferente entre el 8086 y el VAX, aunque la distribución de diferentes tipos de instrucciones de control lo sea. El porcentaje de operaciones aritméticas en el 8086 es mucho más pequeño, debido al menos parcialmente, al mayor número de instrucciones de transferencia *(move)*.

En esta y en las dos secciones precedentes vimos máquinas diseñadas en los años sesenta (el 360) y en los años setenta (el VAX y el 8086). En la siguiente sección hablaremos sobre una máquina típica de las diseñadas en los años ochenta y de su utilización.

Medidas de la utilización del repertorio de instrucciones en DLX

Igual que con las otras arquitecturas que acabamos de examinar, comenzamos nuestro examen de la utilización del repertorio de instrucciones en DLX con medidas de ubicación de operandos y desde allí vamos a las mezclas de instrucciones. Los datos de DLX en este libro se midieron utilizando la arquitectura MIPS R2000/3000 y ajustando los datos para que reflejen las diferencias entre las arquitecturas DLX y MIPS. La tecnología de compiladores MIPS con un nivel de optimización 2, que hace optimización global completa con ubicación de registros, se utilizó para compilar los programas. Un programa especial llamado «pixie» (duende) se utilizó para instrumentar el módulo objeto. El módulo objeto instrumentado produce un fichero de vigilancia *(monitoring file)* que se utiliza para producir estadísticas detalladas de las ejecuciones.

Utilización de los modos de direccionamiento

La utilización de los operandos se muestra en la Figura 4.33. Este dato es muy uniforme en las aplicaciones en DLX. Comparando con el VAX, un porcentaje mucho más elevado de referencias de operandos son a registros: en el VAX, sólo aproximadamente la mitad de las referencias son a registros, mientras que en DLX son, aproximadamente, las tres cuartas partes. Esto ocurre, probablemente, debido al mayor número de registros disponibles en DLX y al mayor énfasis en la ubicación de registros por el compilador DLX.

Como DLX solamente tiene un modo de direccionamiento, no tiene sentido preguntar cómo es la distribución de los modos de direccionamiento. Sin embargo antes observábamos que en el VAX y en el 8086 el modo de direccionamiento diferido, que es equivalente al direccionamiento de desplazamiento con un desplazamiento cero, fue el segundo o tercero más popular. ¿Sería útil añadir este modo a DLX?

Ejemplo Utilizando los datos de los valores de desplazamiento de la Figura 3.13, determinar con qué frecuencia se utilizaría en promedio el modo diferido para los tres programas si el caso de desplazamiento cero fuese un modo especial. En particular, ¿qué porcentajes de referencias a memoria se utilizarían? ¿Qué an-

cho de banda de memoria se ahorraría si tuviésemos una instrucción de 16 bits para este modo de direccionamiento?

Respuesta Las frecuencias de los valores de desplazamiento cero son

GCC: 27%

Spice: 4%

TeX: 17%

La frecuencia media para el valor es entonces (27% + 4% + 17%)/3 = 16%. Por tanto, el modo sería utilizado por el 16 por 100 de las cargas y almacenamientos, que promedian el 32 por 100 de las ejecuciones. La disminución del ancho de banda de las instrucciones sería aproximadamente $\frac{1}{2} \cdot 32\% \cdot 16\%$, o el 3 por 100.

Los otros dos modos de direccionamiento, utilizados con alguna frecuencia, son el escalado en el VAX y el absoluto en el 8086. El modo de direccionamiento escalado se sintetiza en DLX con una suma separada; la presencia de este modo de direccionamiento está influenciada en gran medida por la tecnología del compilador. Los mejores optimizadores utilizan el modo indexado, con menos frecuencia, porque la optimización de la eliminación de variables de inducción elimina la necesidad del direccionamiento indexado y del escalamiento (ver la explicación de la Sección 3.7). El modo directo se sintetiza dedicando un registro que apunte a un área global y accediendo a variables con un desplazamiento desde ese registro. Debido a que sólo las variables escalares (por ejemplo, no estructuras o arrays) necesitan ser accedidas de esta forma, este modo funciona muy bien para la mayoría de los programas.

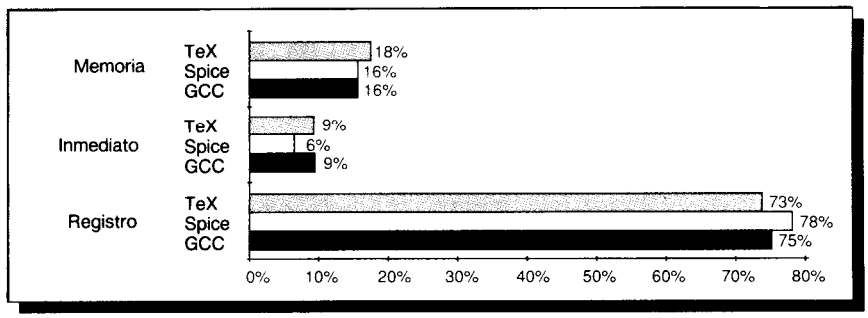


FIGURA 4.33 Distribución de accesos a operandos para los tres benchmarks en DLX. Sólo se incluyen accesos para operandos —no para cálculos de direcciones efectivas—. El hecho que DLX sólo tenga formatos de registro de 3 operandos quizá sea la causa del ligero incremento de la frecuencia de los accesos de operandos a registros, ya que algunas instrucciones probablemente tienen dos únicos operandos de registro y utilizan un registro como fuente y destino. En una máquina como la VAX, tal operación puede utilizar una instrucción de 2 operandos y por tanto ser contabilizada como sólo 2 operandos de registro. Este efecto no se ha medido.

Mezclas de instrucciones en DLX

La Figura 4.34 muestra las mezclas de instrucciones para nuestros tres programas más el benchmark US Steel COBOL —el benchmark COBOL más ampliamente utilizado. El benchmark es un programa sintético de aproximadamente mil líneas. Se incluye aquí porque su comportamiento es sustancialmente diferente de los programas FORTRAN y C. También son interesantes medidas sobre COBOL porque reflejan los cambios, que se presentan en la utilización del repertorio de instrucciones, cuando la aritmética decimal no está directamente soportada por instrucciones decimales. Examinemos primeramente las diferencias entre los programas antes de considerar cómo estas mezclas se comparan con las del VAX.

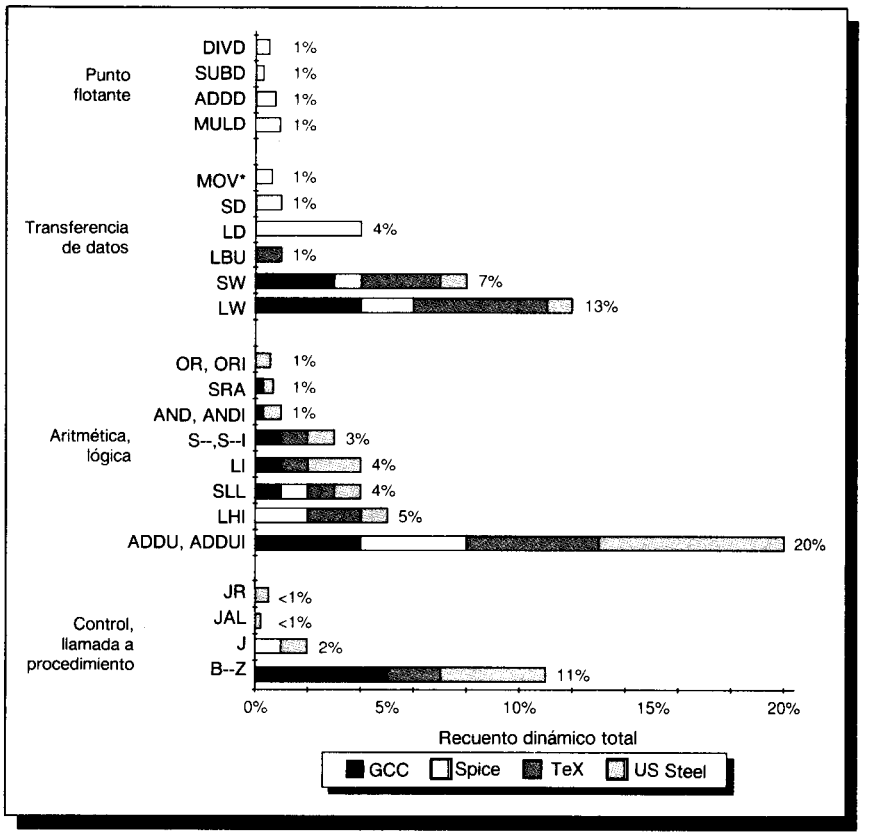


FIGURA 4.34 La mezcla de instrucciones de DLX, visible sobre cuatro programas, descompuesta para mostrar la contribución de cada programa. Lo notable es cómo un pequeño número de instrucciones —salto condicional, suma, carga, y almacenamiento— domina en los cuatro programas. El código de operación `LI` realmente es `ADDUI` con R0 como operando; la alta frecuencia de `ADDU` y `ADDUI` se explica más adelante.

Las diferencias significativas entre estos programas son sorprendentes. Spice y TeX destacan porque tienen una frecuencia de saltos muy baja. El efecto de convertir la aritmética decimal COBOL en aritmética binaria, se ve claramente, en el gran porcentaje de operaciones aritméticas en el US Steel. Todos los desplazamientos, operadores lógicos y cargas inmediatas, que se utilizan para hacer conversiones rápidas decimal-binario, se presentan en frecuencias significativas. La baja frecuencia de transferencias de datos de US Steel ciertamente está afectada por este incremento de las operaciones lógicas y aritméticas. Sólo la frecuencia de llamadas de US Steel es bastante alta para contabilizar más del 1 por 100 de las ejecuciones de instrucciones (la frecuencia de JAL es aproximadamente el 1 por 100 para los otros tres benchmarks).

Estas mezclas difieren enormemente del VAX (o de las demás máquinas de esta sección). Una diferencia es el porcentaje muy alto de las instrucciones ADDU y ADDUI. Estas instrucciones se utilizan para una serie de propósitos donde otras máquinas puedan utilizar una instrucción diferente o una instrucción o modo de direccionamiento más potente. Entre los usos más frecuentes de ADDU y ADDUI están: las copias registro-registro (codificado como ADDU con R0), sintetizar un modo de direccionamiento como el escalado, e incrementar el puntero de pila en una llamada a procedimiento.

Es interesante comparar la frecuencia de saltos entre DLX y el VAX, ya que el número absoluto de saltos debe ser aproximadamente igual (por las razones explicadas antes), y la relación de las frecuencias de saltos debe ser aproximadamente la misma que la relación del número global de instrucciones. Sin embargo, los compiladores pueden afectar al tipo de salto utilizado —salto condicional frente a bifurcación— por lo que necesitamos combinar todos los saltos y bifurcaciones, excepto los utilizados en las llamadas a procedimientos, para realizar una comparación.

Ejemplo

Encontrar la relación de saltos absolutos en el VAX frente a DLX para los tres benchmarks comunes. La relación del número de instrucciones, medidas en la Sección 3.8 es

$$\frac{\text{Instrucciones DLX}}{\text{Instrucciones VAX}} = 2{,}0$$

Utilizar los datos del Apéndice C para porcentajes exactos de saltos.

Respuesta

Del Apéndice C encontramos que la frecuencia media de saltos en DLX es $\frac{19\%+2\%+7\%}{3} = 9{,}3\%$, mientras la media para el VAX es el 17,3 por 100. Por tanto, la relación del número de saltos es

$$\frac{\text{Saltos}_{\text{DLX}}}{\text{Saltos}_{\text{VAX}}} = \frac{9{,}3\% \cdot \text{Instrucciones}_{\text{DLX}}}{17{,}3\% \cdot \text{Instrucciones}_{\text{VAX}}}$$

$$= \frac{9{,}3 \cdot 2{,}0 \cdot \text{Instrucciones}_{\text{VAX}}}{17{,}3 \cdot \text{Instrucciones}_{\text{VAX}}}$$

$$= \frac{18,3}{17,3} = 1,06$$

Por tanto DLX tiene aproximadamente el 6 por 100 más de saltos.

En las instrucciones aritméticas y lógicas, GCC y US Steel son los más diferentes entre el VAX y DLX. Sabemos que US Steel difiere a causa de la ausencia de instrucciones decimales —sería interesante ver si la mezcla de instrucciones en dicho programa sería igual con los nuevos compiladores VAX que evitan las instrucciones decimales. Otra diferencia importante entre las dos máquinas es la frecuencia más baja de las instrucciones de comparación y de test en DLX. El uso de la comparación con cero en la instrucción de salto es responsable de esto. Como las instrucciones «SET» también se utilizan para inicializar variables lógicas, no podemos saber con exactitud el porcentaje de saltos condicionales en DLX, que no necesitan una comparación, pero podemos suponer que está entre el 75 y 80 por 100.

La diferencia en las transferencias de datos se ha explicado ampliamente al final del Capítulo 3 (para una máquina muy próxima a DLX) y en la subsección anterior. Sabemos que el mayor número de registros (como mínimo el doble) y el ubicador de registros más ambicioso significan que la frecuencia de cargas y almacenamientos es menor en DLX que en el VAX.

Ahora hemos visto mezclas de instrucciones para cuatro máquinas diferentes. En el Apéndice D podremos ver cómo difieren estas mezclas cuando examinemos distribuciones de tiempo en lugar de frecuencia de ocurrencias, y en la sección siguiente revisaremos algunas de nuestras observaciones clave y señalaremos algunas pifias adicionales utilizando los datos que hemos examinado en esta y en secciones anteriores.

4.7 Falacias y pifias

Falacias: Existe lo que podríamos llamar **un** *programa típico.*

A mucha gente le gusta creer que hay un sencillo programa «típico» que se puede utilizar para diseñar un repertorio de instrucciones óptimo. Por ejemplo, ver los benchmarks sintéticos explicados en la Sección 2.2. Los datos de este capítulo muestran, claramente, que los programas pueden variar significativamente en la forma que utilizan un repertorio de instrucciones. Por ejemplo, la frecuencia de las instrucciones de flujo de control en DLX variaban del 5 al 23 por 100. Las variaciones son aún mayores en un repertorio de instrucciones que tenga características específicas para soportar una clase de aplicaciones, como por ejemplo, instrucciones decimales o de punto flotante que no son utilizadas por otras aplicaciones. Hay una pifia relacionada.

Pifia: Diseñar una arquitectura basándose en pequeños o grandes benchmarks a partir de un dominio restringido de aplicaciones cuando nos proponemos que la máquina sea de propósito general.

Muchos programas exhiben un comportamiento un tanto parcial, o no utilizan un aspecto particular de una arquitectura. Obviamente, escoger los benchmarks TeX o GCC para diseñar el repertorio de instrucciones, puede dar como resultado una máquina que no ejecute bien un programa como Spice o COBOLX. Un ejemplo más sutil surge cuando se escoge un benchmark representativo, pero sintético. Por ejemplo, Dhrystone (ver Sección 2.2) realiza una llamada a procedimientos, aproximadamente, cada 40 instrucciones en una máquina como DLX —¡el número de llamadas a procedimientos es más de la mitad del número de saltos condicionales! Por comparación, en GCC una llamada se presenta aproximadamente una vez cada 100 instrucciones, y los saltos son 15 veces más frecuentes que las llamadas a los procedimientos.

Falacia: Una arquitectura con defectos no puede tener éxito.

En la literatura se critica con frecuencia al IBM 360 —los saltos no son relativos al PC, y el desplazamiento es muy pequeño en un direccionamiento basado. Sin embargo, la máquina ha tenido un enorme éxito porque hizo adecuadamente algunas cosas nuevas. Primero, la arquitectura tiene un enorme espacio de direcciones. Segundo, es direccionada por bytes y los manipula bien. Tercero, es una máquina de registros de propósito general. Finalmente, es lo bastante sencilla que se puede implementar eficientemente en un amplio rango de rendimiento y coste.

El 8086 proporciona incluso un ejemplo más espectacular. La arquitectura del 8086 es la más extendida hoy día aunque no sea realmente una máquina de registros de propósito general. Además, el espacio segmentado de direcciones del 8086 causa problemas importantes a los programadores y escritores de compiladores. A pesar de estas importantes dificultades, la arquitectura del 8086 —debido a que se ha seleccionado como el microprocesador del IBM PC— ha tenido un éxito enorme.

Falacia: Se puede diseñar una arquitectura sin fallos.

Cualquier diseño de una arquitectura involucra compromisos realizados en el contexto de un conjunto de tecnologías hardware y software. Con el tiempo estas tecnologías cambiarán probablemente, y las decisiones que puedan haber sido correctas en el tiempo que se tomaron pueden parecer erroneas. Por ejemplo, en 1975 los diseñadores del VAX hicieron énfasis en la importancia de la eficiencia del tamaño de código y desestimaron la importancia que la decodificación y segmentación tendrían diez años más tarde. Casi todas las arquitecturas sucumben eventualmente a la carencia de un espacio de direcciones suficiente. Sin embargo, evitar este problema a largo plazo, probablemente, podría significar comprometer la eficiencia de una arquitectura a corto plazo.

Falacia: En mezclas de instrucciones, las distribuciones de tiempo y de frecuencia serán parecidas.

El Apéndice D muestra las distribuciones de tiempo para nuestros programas de benchmark y compara las distribuciones de tiempo y frecuencia. Un sencillo ejemplo, donde son muy diferentes estas distribuciones, está en el programa COBOLGO en el 360. La Figura 4.35 muestra las instrucciones superiores por frecuencia y tiempo. Las dos instrucciones que aparecen más veces

Instrucciones ordenados por frecuencia	Frecuencia	Instrucciones ordenados por distribución de tiempo	Porcentaje de tiempo
L, LR	19 %	ZAP	16 %
BC, BCR	14 %	AP	16 %
AP	11 %	MP	13 %
ZAP	9 %	MVC	9 %
MVC	7 %	CVD	5 %

FIGURA 4.35 Las cinco instrucciones más utilizadas por frecuencia y tiempo para el benchmark COBOLGO ejecutando el 360. También se muestra la frecuencia o porcentaje real de tiempo. Datos adicionales aparecen en el Apéndice D.

son responsables del 33 por 100 de las ejecuciones de las instrucciones en COBOLGO, ¡pero solamente del 4 por 100 del tiempo de ejecución! Recordar que las distribuciones de tiempo dependen de la arquitectura y de la **implementación** utilizada para la medida. Por consiguiente, las distribuciones de tiempo pueden diferir de modelo a modelo, aunque las distribuciones de frecuencia sean las mismas, suponiendo que ni software ni el programa cambien. Esta gran diferencia entre distribuciones de tiempo y frecuencia no existe para arquitecturas más simples de carga/almacenamiento, como DLX.

Pifia: Examinar sólo el peor caso o el comportamiento medio de una instrucción como información para el diseño.

El mejor ejemplo proviene del uso de MVC en un IBM 360. La instrucción puede transferir campos de caracteres solapados, pero esto ocurre menos del 1 por 100 de las veces, y entonces habitualmente para borrar un campo. La longitud media de una transferencia medida por Shustek fue de 10 bytes, pero más de tres cuartos de las transferencias eran de 1 byte o de 4 bytes. Suponer el comportamiento del peor caso (solapamiento de cadenas) o longitud media puede conducirnos a decisiones subóptimas de diseño.

4.8 Observaciones finales

Hemos visto que los repertorios de instrucciones pueden variar mucho según la forma en que se acceda a los operandos y en las operaciones que pueda realizar una sola instrucción. La comparación de la utilización por frecuencia de instrucciones, de los códigos de operación en las arquitecturas que estamos considerando se resume en la Figura 4.36. Esta figura muestra que incluso arquitecturas muy diferentes se comportan de forma análoga en su uso de las clases de instrucciones. Sin embargo, esto nos debe recordar también que el rendimiento puede estar sólo remotamente relacionado con la utilización de

Máquina	Programa	Control (%)	Aritmética lógica (%)	Transferencia de datos (%)	Punto flotante (%)	Decimal, cadena (%)	Totales (%)
VAX	GCC	30	40	19			89
VAX	Spice	18	23	15	23		79
VAX	TeX	30	33	28			91
VAX	COBOLX	25	24	4		38	91
360	PLIC	32	29	17		4	82
360	FORTGO	13	35	40	7		95
360	PLIGO	5	29	56			90
360	COBOLGO	16	9	20		40	85
8086	Turbo C	21	23	49			93
8086	MASM	20	24	46			90
8086	Lotus	32	26	30			88
DLX	GCC	24	35	27			86
DLX	Spice	4	29	35	15		83
DLX	TeX	10	41	33			84
DLX	US Steel	23	49	10			82

FIGURA 4.36 La frecuencia de distribución de instrucciones para cada benchmark descompuesta en cinco clases de instrucciones. Como sólo se han incluido instrucciones con frecuencias mayores que 1,5 por 100, los totales son menores que el 100 por 100.

las instrucciones —las distribuciones de los tiempos de ejecución del Apéndice D, para estas arquitecturas parecen muy diferentes.

Aunque es espectacular la variación de la utilización de las instrucciones en las arquitecturas, es igualmente espectacular en las aplicaciones. Hemos visto que programas en punto flotante, programas COBOL, y programas de sistemas C difieren en la forma que utilizan la máquina. Algunos programas no utilizan grandes segmentos del repertorio de instrucciones. Cuando las características de aplicaciones específicas no son parte del repertorio de instrucciones —por ejemplo, la ausencia de instrucciones decimales en DLX— el impacto es un cambio en la utilización de otras partes de la instrucción. Incluso en dos programas escritos en el mismo lenguaje —GCC y TeX, o PLIC y PLIGO— las diferencias en la utilización de instrucciones pueden ser significativas.

Los datos sobre utilización de las instrucciones son una entrada importante para el arquitecto, pero no indican necesariamente las instrucciones que consumen más tiempo. Los capítulos siguientes ayudarán a explicar por qué surgen diferencias al cuantificar diferencias de CPI entre instrucciones y máquinas.

4.9 Perspectiva histórica y referencias

Aunque gran número de máquinas se ha desarrollado al mismo tiempo que las cuatro máquinas estudiadas en este capítulo, en este apartado nos centramos en estas máquinas y en medidas sobre ellas.

El IBM 360 se introdujo en 1964 con seis modelos y una relación de rendimiento 25:1. Amdahl, Blaauw y Brooks [1964] explicaron la arquitectura del IBM 360 y el concepto de permitir múltiples implementaciones de código objeto compatibles. La noción de arquitectura a nivel lenguaje máquina, como la comprendemos hoy, fue el aspecto más importante del 360. La arquitectura también introdujo algunas innovaciones importantes, muy utilizadas hoy día:

1. Arquitectura de 32 bits
2. Memoria direccionable por bytes con bytes de 8 bits
3. Tamaños de datos de 8, 16, 32, y 64 bits

En 1971, IBM construyó el primer System/370 (modelos 155 y 165), que incluía una serie de extensiones significativas del 360, como explicaron Case y Padegs [1978], que también comentan la historia del System/360. La adición más importante fue la memoria virtual, aunque no estuvo en el 370 hasta 1972 cuando estuvo preparado un sistema operativo de memoria virtual. En 1978, el último 370 era varios cientos de veces más rápido que los primeros 360 construidos unos diez años antes. En 1984, fue necesario abandonar el modelo de direccionamiento de 24 bits construido en el IBM 360 y se introdujo el 370-XA (Arquitectura eXtendida). Aunque los antiguos programas de 24 bits podían estar soportados sin cambios, algunas instrucciones no funcionaban de la misma forma cuando se utilizaba el modelo de direccionamiento de 32 bits (direcciones de 31 bits) porque no podían producir direcciones de 31 bits. Convertir el sistema operativo, la mayor parte escrito en lenguaje ensamblador, fue sin duda la tarea más grande.

Se han realizado algunos estudios del IBM 360 y medidas de instrucciones. La tesis de Shustek [1978] es el estudio más conocido y más completo de la arquitectura 360/370. Hizo algunas observaciones sobre la complejidad del repertorio de instrucciones que no fueron apreciadas completamente hasta unos años más tarde. Otro estudio importante sobre el 360 es el de Toronto, debido a Alexander y Wortman [1975] y realizado en un IBM 360 utilizando 19 programas XPL.

A mediados de los años setenta, DEC cayo en la cuenta que el PDP-11 estaba agotando el espacio de direcciones. El espacio de 16 bits se había extendido de varias formas creativas. Sin embargo, como observaron Strecker y Bell [1976], el pequeño espacio de direcciones fue un problema que no podía ser superado, sino solo pospuesto.

En 1978, DEC introdujo el VAX. Strecker [1978] describió la arquitectura y la denominó «una Extensión de Direcciones Virtuales de la PDP-11» a —a Virtual Addres eXtension of the PDP-11—. Uno de los principales objetivos de DEC fue conservar la base instalada de clientes del PDP-11. Por ello,

los clientes pensaron en el VAX como un sucesor de 32 bits del PDP-11. Un PDP-11 de 32 bits era factible —había tres diseños— pero Strecker consideró que era «demasiado comprometido en términos de eficiencia, funcionalidad y fácil programación». La solución elegida fue diseñar una nueva arquitectura e incluir un modo de compatibilidad PDP-11, que pudiera ejecutar programas PDP-11 sin cambios. Este modo también permitió que se siguiesen utilizando y corriendo los compiladores PDP-11. En muchos aspectos el VAX-11-780 se construyó de forma similar al PDP-11. Entre los más importantes se encuentran:

1. Los tipos y formatos de datos son en su mayor parte equivalentes a los del PDP-11. Los formatos flotantes F y D provienen del PDP-11. Los formatos G y H se añadieron más tarde. El uso del término «palabra» para describir una cantidad de 16 bits se llevó del PDP-11 al VAX.
2. El lenguaje ensamblador se construyó de forma similar al del PDP-11.
3. Se soportaron los mismos buses (Unibus y Massbus).
4. El sistema operativo, VMS, fue «una evolución» del RSX-11M/IAS OS (en contraposición al DECsystem 10/20 OS, que fue un sistema más avanzado).
5. El sistema de ficheros era básicamente el mismo.

El VAX-11/780 fue la primera máquina anunciada de la serie VAX. Es una de las máquinas construidas con más éxito y estudiada intensamente. La piedra angular de la estrategia de DEC fue una única arquitectura, VAX, corriendo un único sistema operativo, VMS. Esta estrategia funcionó bien durante diez años. El gran número de artículos sobre mezclas de instrucciones, medidas de implementación y análisis del VAX, hacen de él un caso ideal de estudio.

Wiecek [1982] escribió sobre el uso de diversas características de la arquitectura corriendo una carga de trabajo que constaba de seis compiladores. Emer hizo una serie de medidas (comentadas por Clark y Levy [1982]) sobre la utilización del repertorio de instrucciones del VAX cuando corría cuatro programas muy diferentes y cuando corría el sistema operativo. Una descripción bien detallada de la arquitectura, incluyendo la gestión de memoria y un examen de diversas implementaciones del VAX puede encontrarse en Levy y Eckhouse [1989].

Los primeros microprocesadores se fabricaron más tarde, en la primera mitad de los años setenta. El Intel 4004 y 8008 fueron máquinas extremadamente simples de estilo acumulador de 4 bits y 8 bits. Morse y cols. [1980] describen la evolución del 8086 a partir del 8080 al final de los años setenta, en un intento de proporcionar una máquina de 16 bits con mejor productividad. En esa época casi toda la programación para los microprocesadores se hacía en lenguaje ensamblador —memoria y compiladores no eran muy abundantes. Intel quería conservar su base de usuarios del 8080, por ello, se diseñó el 8086 para que fuese «compatible» con el 8080. El 8086 **nunca** tuvo el código objeto compatible con el 8080, pero las máquinas eran tan parecidas

que la traducción de los programas en lenguaje ensamblador podía hacerse automáticamente.

A principios de los años ochenta, IBM seleccionó una versión del 8086 con un bus externo de 8 bits, denominada 8088, para utilizarla en el IBM PC. (Escogieron la versión de 8 bits para reducir el coste de la máquina.) Sin embargo, esta elección con el tremendo éxito del IBM PC y sus clónicos (fue posible porque IBM abrió la arquitectura del PC) hizo omnipresente la arquitectura del 8086. Aunque el 68000 fue escogido por el popular Macintosh, el Macintosh nunca fue tan generalizado como el PC (particularmente porque Appel no permitió clónicos), y el 68000 no adquirió la misma influencia en el software que el 8086. El Motorola 68000 puede haber sido más significativo **técnicamente** que el 8086, pero el impacto de la selección por IBM y la estrategia de arquitectura abierta de IBM dominaron, en el mercado, sobre las ventajas del 68000. Como explicamos en la Sección 4.4, el 80186, 80286, 80386 y 80486 han extendido la arquitectura y proporcionado una serie de mejoras en el rendimiento.

Se han publicado numerosas descripciones de la arquitectura 80×86 —Wakerly [1989] es concisa y fácil de comprender. Crawford y Gelsinger [1988] es una descripción minuciosa del 80386. El trabajo de Adams y Zimmerman [1989] representa el primer estudio publicado y detallado, del uso dinámico de la arquitectura del que tenemos conocimiento; los datos del 8086 utilizados en este libro provienen de este estudio.

Las sencillas máquinas de carga y almacenamiento de las que se obtuvo DLX comúnmente se denominan arquitecturas RISC —reduced instruction set computer— *(computador de repertorio de instrucciones reducido).* Las raíces de las arquitecturas RISC vuelven a las máquinas como la 6600, donde Thornton, Cray y otros reconocieron la importancia de la simplicidad del repertorio de instrucciones para construir una máquina rápida. Cray continuó con su tradición de fabricar sencillas máquinas en la CRAY-1. Sin embargo, DLX y sus homólogos cercanos se construyen principalmente sobre el trabajo de tres proyectos de investigación: el procesador RISC de Berkeley, el IBM 801, y el procesador MIPS de Stanford. Estas arquitecturas han logrado un interés industrial enorme debido a la afirmación de tener un rendimiento de dos a cinco veces sobre las demás máquinas que utilizan la misma tecnología.

Empezando a finales de los años setenta, el proyecto IBM fue el primero en arrancar, pero fue el último en hacerse público. La máquina IBM se diseñó como un minicomputador ECL, mientras en los proyectos de la universidad eran ambos microprocesadores basados en MOS. John Cocke es considerado el padre del diseño del 801. Recibió los premios Eckert-Mauchly y Turing en reconocimiento de su contribución. Radin [1982] describe los aspectos notables de la arquitectura 801. El 801 fue un proyecto experimental, pero nunca se diseñó como un producto. De hecho, para restringir costes y complejidad, la máquina se construyó solo con registros de 24 bits.

En 1980, Patterson y cols. en Berkeley comenzaron el proyecto (ver Patterson y Ditzel [1980]) en el que contruyeron dos máquinas llamadas RISC-I y RISC-II. Debido a que el proyecto de IBM no era muy conocido ni estudiado, el papel que jugó el grupo de Berkeley en promocionar la aproximación RISC fue crítico para la aceptación de la tecnología. Además de una sencilla arquitectura de carga/almacenamiento, esta máquina introducía ventajas

de registros —una idea que ha sido adoptada por varias máquinas RISC comerciales (este concepto se explica más adelante en el Capítulo 8). El grupo de Berkeley siguió en la construcción de máquinas RISC para Smalltalk, ver Ungar y cols. [1984], y LISP, ver Taylor y cols. [1987].

En 1981, Hennessy y cols. de Stanford publicaron una descripción de la máquina MISP de Stanford. Segmentación eficiente y planificación de la segmentación asistida por el compilador fueron aspectos claves del diseño original de MISP.

Estas tres primeras máquinas RISC tenían mucho en común. Ambos proyectos de universidad estaban interesados en diseñar una máquina sencilla que se pudiera construir en VLSI en el entorno universitario. Las tres máquinas —801, MISP, y RISC-II— utilizaban una sencilla arquitectura de carga/almacenamiento, instrucciones de formato fijo de 32 bits, y segmentación eficiente. Patterson [1985] describe las tres máquinas y los principios básicos de diseño que han venido a caracterizar lo que es una máquina RISC. Hennessy [1984] da otra visión de las mismas ideas, así como otros aspectos del diseño VLSI de procesadores.

En 1985, Hennessy publicó una explicación de las ventajas del rendimiento RISC y lo relacionó con un CPI sustancialmente más bajo —menos de dos para una máquina RISC y sobre diez para un VAX-11/780 (aunque no con idénticas cargas de trabajo). Un artículo de Emer y Clark [1984] caracterizando el rendimiento del VAX-11/780 contribuyó eficazmente a ayudar a los investigadores RISC a comprender la fuente de la ventaja de rendimiento visto por sus máquinas.

Desde que los proyectos de la universidad terminaron, en el período 1983-1984, la tecnología ha sido ampliamente aprovechada por la industria. Muchos de los primeros computadores (antes de 1986) decían ser máquinas RISC. Sin embargo, con frecuencia estas demandas nacieron más de la ambición comercial que de realidades de ingeniería.

En 1986, la industria de computadores comenzó a anunciar procesadores basados en la tecnología explorada por los tres proyectos de investigación RISC. Moussouris y cols. [1986] describen el procesador entero MIPS R2000; mientras Kane [1987] hace una descripción completa de la arquitectura. Hewlett-Packard convirtió su línea existente de minicomputadores en arquitecturas RISC; la Arquitectura de Precisión HP es descrita por Lee [1989]. IBM nunca convirtió directamente el 801 en producto. En lugar de ello, las ideas fueron adoptadas para una nueva arquitectura que se incorporó en el IBM RT-PC y se describe en una colección de artículos [Waters 1986]. En 1990, IBM anunció una nueva arquitectura RISC (la RS 6000), que es la primera máquina RISC superescalar (ver Capítulo 6). En 1987, Sun Microsystems comenzó a suministrar máquinas basadas en la arquitectura SPARC, una derivación de la máquina RISC-II de Berkeley; SPARC es descrita por Garner y cols. [1988]. Comenzando 1987, los fabricantes de semiconductores empezaron a convertirse en suministradores de microprocesadores RISC. Con el anuncio del AMD 29000, AMD fue el primer fabricante importante de semiconductores en suministrar una máquina RISC. En 1988, Motorola anunció la disponibilidad de su máquina RISC, la 88000.

Antes del movimiento de las arquitecturas RISC, la principal tendencia había estado en las arquitecturas de microcódigo encaminadas a reducir el

desnivel semántico. DEC, con VAX e Intel, con el iAPX 432, estaban entre los líderes en esta aproximación. En 1989, DEC e Intel anunciaron productos RISC —la DECstation 3100 (basada en el MIPS Computer Systems R2000) y el Intel i860, un nuevo microprocesador RISC. Con estos anuncios (y el IBM RS6000), la tecnología RISC ha alcanzado bastante aceptación. En 1990 es difícil encontrar una compañía de computadores sin un producto RISC.

Referencias

ADAMS, T. AND R. ZIMMERMAN [1989]. «An analysis of 8086 instruction set usage in MS DOS programs», *Proc. Third Symposium on Architectural Support for Programming Languages and Systems* (April) Boston, 152-161.

ALEXANDER, W. G. AND D. B. WORTMAN [1975]. «Static and dynamic characteristics of XPL programs», *Computer* 8:11 (November) 41-46.

AMDAHL, G., G. BLAAUW, AND F. BROOKS [1964]. «Architecture of the IBM System/360», *IBM J. of Research and Development* 8:2 (April) 87-101.

CASE, R. AND A. PADEGS [1978]. «Architecture of the IBM System/370», *Comm. ACM* 21:1 (January) 73-96.

CHOW, F., M. HIMELSTEIN, E. KILLIAN, AND L. WEBER [1986]. «Engineering a RISC compiler system», *Proc. COMPCON* (March), San Francisco, 132-137.

CLARK, D. AND H. LEVY [1982]. «Measurement and analysis of instruction set use in the VAX-11/780», *Proc. Ninth Symposium on Computer Architecture* (April), Austin, Tex., 9-17.

CRAWFORD, J. AND P. GELSINGER [1988]. *Programming the 80386,* Sybex Books, Alameda, Calif.

EMER, J. S. AND D. W. CLARK [1984]. «A characterization of processor performance in the VAX-11/780», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301-310.

GARNER, R., A. AGARWAL, F. BRIGGS, E. BROWN, D. HOUGH, B. JOY, S. KLEIMAN, S. MUNCHNIK, M. NAMJOO, D. PATTERSON, J. PENDLETON, AND R. TUCK [1988]. «Scalable processor architecture (SPARC)», *COMPCON, IEEE* (March), San Francisco, 278-283.

HENNESSY, J. [1984]. «VLSI processor architecture», *IEEE Trans. on Computers* C-33:11 (December) 1221-1246.

HENNESSY, J. [1985]. «VLSI RISC processors», *VLSI Systems Design* VI:10 (October) 22-32.

HENNESSY, J., N. JOUPPI, F. BASKETT, AND J. GILL [1981]. «MIPS: A VLSI processor architecture», *Proc. CMU Conf. on VLSI Systems and Computations* (October), Computer Science Press, Rockville, Md.

KANE, G. [1986]. *MIPS R2000 RISC Architecture,* Prentice Hall, Englewood Cliffs, N.J.

LEE, R. [1989]. «Precision architecture», *Computer* 22:1 (January) 78-91.

LEVY, H. AND R. ECKHOUSE [1989]. *Computer Programming and Architecture: The VAX,* Digital Press, Boston.

MORSE, S., B. RAVENAL, S. MAZOR, AND W. POHLMAN [1980]. «Intel Microprocessors—8008 to 8086», *Computer* 13:10 (October).

MOUSSOURIS, J., L. CRUDELE, D. FREITAS, C. HANSEN, E. HUDSON, S. PRZYBYLSKI, T. RIORDAN, AND C. ROWEN [1986]. «A CMOS RISC processor with integrated system functions», *Proc. COMPCON, IEEE* (March), San Francisco.

PATTERON, D. [1985]. «Reduced Instruction Set Computers», *Comm. ACM* 28:1 (January) 8-21.

PATTERSON, D. A. AND D. R. DITZEL [1980]. «The case for the reduced instruction set computer», *Computer Architecture News* 8:6 (October), 25-33.

RADIN, G. [1982]. «The 801 minicomputer», *Proc. Symposium Architectural Support for Programming Languages and Operating Systems* (March), Palo Alto, Calif. 39-47.

SHUSTEK, L. J. [1978]. «Analysis and performance of computer instruction sets», Ph.D. Thesis (May), Stanford Univ., Stanford, Calif.

STRECKER, W. [1978]. «VAX-11/780: A virtual address extension tot the DEC PDP-11 family», *Proc. AFIPS NCC* 47, 967-980.

STRECKER, W. D. AND C. G. BELL [1976]. «Computer structures: What have we learned from the PDP-11?», *Proc. Third Symposium on Computer Architecture.*

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND B. ZORN [1986]. «Evaluation of the SPUR LISP architecture», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo.

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. «Architecture of SOAR: Smalltalk on a RISC», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188-197.

WAKERLY, J. [1989]. *Microcomputer Architecture and Programming,* J. Wiley, New York.

WATERS, F., ED. [1986]. *IBM RT Personal Computer Technology,* IBM, Austin, Tex., SA 23-1057.

WIECEK, C. [1982]. «A case study of the VAX 11 instruction set usage for compilar execution», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177-184.

EJERCICIOS

En estos ejercicios a menudo será necesario conocer la frecuencia de las instrucciones individuales de una mezcla. Las Figuras C.1 a C.4 proporcionan los datos correspondientes a las Figuras 4.24, 4.28, 4.32 y 4.34. Adicionalmente, algunos problemas involucran distribución de los tiempos de ejecución en lugar de distribución de frecuencias. La información sobre distribución de tiempos de ejecución aparece en el Apéndice D; los problemas que requieren datos de los Apéndices C o D incluyen la letra C o D entre ángulos, por ejemplo, <C,D>.

Al hacer estos ejercicios se necesitará trabajar con medidas que puedan no totalizar el 100 por 100. En algunos casos se necesitarán normalizar los datos al total real. Por ejemplo, si se preguntase calcular la frecuencia de las instrucciones MOV_ en las ejecuciones de Spice en la VAX, deberíamos proceder como sigue (utilizando datos de la Figura C.1):

> Frecuencia de MOV_ medida en tabla = 9 % + 6 % = 15 %
>
> Fracción de todas las instrucciones ejecutadas incluidas en la Figura C.1 para Spice = 79 %

Ahora normalizamos el 15 por 100. Esto es equivalente a suponer que el 21 por 100 no medido de la mezcla de instrucciones se comporta igual que la porción medida. Como hay instrucciones MOV_ no medidas ésta es la aproximación más lógica.

$$\text{Frecuencia de MOV_ en Spice sobre VAX} = \frac{15\%}{79\%} = 19\%$$

Sin embargo, si se preguntase calcular la frecuencia de MOVL en Spice, sabemos que es exactamente 9 por 100, ya que tenemos una medida completa para este tipo de instrucción.

4.1 [20/20] <4.2,4.6,C> Suponer que se está en una entrevista en Digital Equipment Corporation con el fin de realizar un trabajo como diseñador de los futuros computadores VAX. Pero, antes de contratarle, quieren hacerle algunas preguntas. Han permitido que traiga sus notas, incluyendo la Sección 4.6 y el Apéndice C.

Recuerde un ejemplo del Capítulo 4 donde se comentaba que la instrucción media de la VAX tenía 1,8 operandos. Recuerde también que los códigos de operación casi siempre tienen 1 byte.

a) [20] Le piden que determine el tamaño medio de una instrucción VAX. Utilice los datos de frecuencia de los modos de direccionamiento de 4.22 y 4.23, la información sobre los tamaños de los desplazamientos de la Figura 3.35 y la longitud de los modos de direccionamiento VAX mostrados en la Figura 4.3. (Esta sería una estimación más precisa que el ejemplo que aparece en la página 183.)

b) [20] Le piden que evalúe el rendimiento de su nueva máquina con un reloj de 100 MHz. Le indican que el CPI medio para todo excepto búsqueda de instrucciones y de operandos, es de 3 ciclos. También le indican que

- cada especificador y acceso a la memoria de datos necesita 2 ciclos adicionales, y
- cada 4 bytes de instrucciones buscadas por la unidad de búsqueda de instrucciones emplea un ciclo.

¿Puede calcular los MIPS nativos efectivos?

4.2 [20/22/22] <4.2,4.3,4.5> Considerar el siguiente fragmento de código C:

```
for (i=1; i<=100; i++)
   {A[i] = B[i] + C;}
```

Suponer que `A` y `B` son arrays de enteros de 32 bits, y `C` e `i` son enteros de 32 bits. Suponer que los valores de todos los datos se encuentran en memoria (en las direcciones 0, 5000, 1500 y 2000 para `A`, `B`, `C` e `i`, respectivamente) excepto cuando están operando.

a) [20] Escribir el código para DLX; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias de datos a memoria se ejecutarán? ¿Cuál es el tamaño del código?

b) [22] Escribir el código para el VAX; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias de datos a memoria se ejecutarán? ¿Cuál es el tamaño del código?

c) [22] Escribir el código para el 360; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias a la memoria de datos se ejecutarán? ¿Cuál es el tamaño del código? Por simplicidad, se puede suponer que el registro R1 contiene la dirección de la primera instrucción del bucle.

4.3 [20/22/22] <4.2,4.3,4.5> Para este ejercicio utilizar la secuencia del código del Ejercicio 4.2, pero poner los datos escalares —el valor de `i` y la dirección de las variables del array (pero no el contenido del array)— en los registros y mantenerlos allí siempre que sea posible.

a) [20] Escribir el código para DLX; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias a la memoria se ejecutarán? ¿Cuál es el tamaño del código?

b) [22] Escribir el código para el VAX; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias a la memoria de datos se ejecutarán? ¿Cuál es el tamaño del código?

c) Escribir el código para el 360; ¿cuántas instrucciones se requieren dinámicamente? ¿Cuántas referencias a la memoria de datos se ejecutarán? ¿Cuál es el tamaño del código? Suponer que R1 está inicializado como en el Ejercicio 4.2, parte C.

4.4 [15] <4.6> Cuando se diseñan sistemas de memoria es útil conocer la frecuencia de las lecturas de memoria frente a las escrituras y también los accesos de las instrucciones frente a los de los datos. Utilizando la información media de mezcla de instrucciones para DLX en el Apéndice C, calcular

- el porcentaje de todos los accesos a memoria que son para datos
- el porcentaje de los accesos a datos que son lecturas
- el porcentaje de todos los accesos a memoria que son lecturas

Ignorar el tamaño del dato al contar los accesos.

4.5 [15] <4.3,4.6> Debido a la falta de saltos relativos al PC, un salto en un 360 requiere, con frecuencia, dos instrucciones. Esto ha sido una crítica importante de la arquitectura. Determinar lo que cuesta esta omisión, suponiendo que siempre se necesita una instrucción extra para un salto condicional en el 360, pero que la instrucción no sería necesaria con saltos relativos al PC. Utilizar los datos medios de la Figura 4.28 para los saltos, determinar cuántas instrucciones más ejecuta el 360 estándar que un 360 con saltos relativos al PC. (Recordar que sólo BC es un salto.)

4.6 [15] <4.2,4.6> Estamos interesados en añadir una instrucción a la arquitectura VAX que compare un operando con cero y salte. Suponer que

- sólo se pueden eliminar las instrucciones que inicialicen el código de condición para un salto condicional;
- el 80 por 100 de los saltos condicionales requiere una instrucción cuyo único propósito sea inicializar la condición, y
- el 90 por 100 de todos los saltos que tengan una instrucción que inicialice la condición (por ejemplo, el 80 por 100 mencionado) están basados en una comparación con 0.

Utilizando los datos medios del VAX de la Figura 4.24, ¿qué porcentaje más de instrucciones ejecutará el VAX estándar en comparación con el VAX al que se le ha añadido la instrucción de compara y salta?

4.7 [18] <4.5,4.6> Calcular el CPI efectivo para DLX. Suponer que hemos hecho las siguientes medidas del CPI medio para instrucciones:

Todas instrucciones R-R	1 ciclo de reloj
Cargas/almacenamientos	1,4 ciclo de reloj
Saltos condicionales	
efectivos	2,0 ciclos de reloj
no efectivos	1,5 ciclos de reloj
Bifurcaciones	1,2 ciclos de reloj

Suponer que el 60 por 100 de los saltos condicionales son efectivos. Promediar las frecuencias de las instrucciones de GCC y TeX para obtener la mezcla de instrucciones.

4.8 [15] <4.2,4.5> En lugar de tener inmediatos soportados por muchos tipos de instrucciones, algunas arquitecturas, como la del 360, reúnen los inmediatos en memoria (en un almacén de literales) y accede a ellos desde allí. Suponer que el VAX no tiene modo de direccionamiento inmediato, pero en lugar de ello ubica inmediatos en memoria y los accede utilizando el modo de direccionamiento de desplazamiento. ¿Cuál sería el incremento en la frecuencia con que se utilizaría el modo de desplazamiento? Utilizar el promedio de las medidas de las Figuras 4.22 y 4.23 para este problema.

4.9 [20/10] <4.5,4.6> Considerar la inclusión de un nuevo modo de direccionamiento índice a DLX. El modo de direccionamiento añade dos registros y un desplazamiento con signo de 11 bits para obtener la dirección efectiva.

Se cambiará nuestro compilador para que las secuencias de código de la forma

```
ADD R1, R1, R2
LW Rd, 0(R1)      (o store)
```

sean sustituidas por una carga (o almacenamiento) utilizando el nuevo modo de direccionamiento. Utilizar las frecuencias medias globales de las instrucciones al evaluar esta adición.

a) [20] Suponer que se puede utilizar el modo de direccionamiento para el 10 por 100 de las cargas y almacenamientos de los desplazamientos (contabilizar para ambos la frecuencia de este tipo de cálculo de direcciones y el desplazamiento más corto). ¿Cuál es la relación de número de instrucciones en el DLX mejorado comparada con el DLX original?

b) [10] Si el nuevo modo de direccionamiento alarga el ciclo de reloj un 5 por 100, ¿qué máquina será más rápida y cuánto?

4.10 [12] <4.2,4.5,D> Suponer que el número medio de instrucciones involucradas en una llamada y retorno en DLX es 8. La frecuencia media de una instrucción JAL de los benchmarks es del 1 por 100. Si todas las instrucciones en DLX tienen el mismo número de ciclos, comparar el porcentaje de ciclos de las llamadas y retornos en DLX con el porcentaje de ciclos de las CALLS y RET en el VAX.

4.11 [22/22] <4.2,4.3,4.6,D> Algunas personas piensan que las instrucciones más empleadas son también las más simples, mientras otros han señalado que las instrucciones que consumen más tiempo, a menudo, no son las más frecuentes.

a) [22] Utilizando los datos de la Figura D.1, calcular el CPI de las cinco instrucciones que consumen más tiempo en el VAX que tiene una frecuencia de ejecución media de, al menos, el 2 por 100. Suponer que el CPI global del VAX es 10.

b) [22] Calcular el CPI para las cinco instrucciones que consumen más tiempo en el 360 que tienen como mínimo un 3 por 100 de frecuencia media, utilizando los datos de la Figura D.2. Suponer que el CPI global del 360 es 4.

4.12 [20/20/10] <4.4,4.6,D> Se le ha contratado para que intente convertir la arquitectura 8086 en otra más orientada a registro-registro. Para hacer esto, necesitará más registros y, por consiguiente, más espacio de codificación, ya que las codificaciones están ajustadas. Suponga que ha determinado que, eliminando las instrucciones PUSH y POP, puede obtener el espacio de codificación necesario. Suponga, que incrementando el número de registros, se reduce en un 25 por 100 la frecuencia de cada una de las instrucciones de referencia a memoria (PUSH, POP, LES y MOV), pero que cada instrucción PUSH o POP restante debe ser sustituida por una secuencia de dos instrucciones. Utilizar los datos medios de las Figuras 4.30-4.32, el CPI medio de 14,1 y la Figura D.3 para responder las siguientes preguntas sobre esta nueva máquina —la RR8086— frente a la 8086.

a) [20] ¿Qué máquina ejecuta más instrucciones y cuántas?

b) [20] Utilizando la información del Apéndice D, determinar qué máquina tiene mayor CPI y en cuánto.

c) [10] Suponiendo que las frecuencias del reloj son idénticas, ¿qué máquina es más rápida y cuánto?

4.13 [25/15] <4.2-4.5> Encontrar un compilador C y compilar el código mostrado en el Ejercicio 4.2 para una máquina de carga/almacenamiento o una de las máquinas estudiadas en este capítulo. Compilar los códigos optimizado y no optimizado.

a) [25] Calcular el número de instrucciones, bytes de instrucción dinámicos buscados y accesos realizados a datos por ambas versiones, la optimizada y la no optimizada.

b) [15] Tratar de mejorar manualmente el código y calcular las mismas medidas que en la parte a para su versión optimizada manualmente.

4.14 [30] <4.6> Si se tiene acceso a un VAX, compilar el código para Spice y tratar de determinar por qué hace un uso mucho menor de inmediatos que programas como GCC y TeX (ver Fig. 4.22).

4.15 [30] <4.6> Si se tiene acceso a una máquina basada en el 8086 compilar algunos programas y examinar la frecuencia de las instrucciones MOV. ¿Cómo corresponde esto a la frecuencia de la Figura 4.32? Examinando el código, ¿se pueden encontrar razones por las que es tan elevada la frecuencia de MOV?

4.16 [30/30] <4.6,4.7> Los pequeños benchmarks sintéticos pueden ser muy erróneos cuando se utilizan para medir mezclas de instrucciones. Esto es particularmente cierto cuando se optimizan estos benchmarks. En estos ejercicios queremos explorar estas diferencias. Estos ejercicios de programación pueden hacerse con un VAX, cualquier máquina de carga/almacenamiento, o utilizando el simulador y compilador DLX.

a) [30] Compilar Whetstone con optimización para un VAX o una máquina de carga/almacenamiento análoga a DLX (por ejemplo, una DECstation o una SPARCstation), o el simulador DLX. Calcular la mezcla de instrucciones para las 20 instrucciones superiores. Comparar las mezclas optimizadas y no optimizadas. Comparar la mezcla optimizada con la mezcla para Spice en la misma máquina o en una similar.

b) [30] Compilar Dhrystone con optimización para un VAX o una máquina de carga/almacenamiento análoga a DLX (por ejemplo, una DECstation o una SPARCstation), o el simulador DLX. Calcular la mezcla de instrucciones para las 20 instrucciones superiores. ¿Cómo comparar las mezclas optimizada y no optimizada? ¿Cómo comparar la mezcla optimizada con la mezcla para TeX en la misma máquina o en una similar?

4.17 [30] <4.6> Muchos fabricantes de computadores incluyen ahora herramientas o simuladores que permiten medir la utilización del repertorio de instrucciones de un programa de usuario. Entre los métodos utilizados está la simulación de la máquina, traps soportados por hardware, y una técnica del compilador que instrumenta el módulo del código objeto insertando contadores. Encontrar un procesador disponible que incluya dicha herramienta. Utilizarlo para medir la mezcla del repertorio de instrucciones para uno de los programas TeX, GCC o Spice. Comparar los resultados con los mostrados en este capítulo.

4.18 [30] <4.5,4.6> DLX tiene solamente tres formatos de operandos para sus operaciones registro-registro. Muchas operaciones deben utilizar el mismo registro como destino y fuente. Podemos introducir un nuevo formato de instrucción en DLX lla-

mado R_2 que sólo tiene dos operandos y un total de 24 bits. Utilizando este tipo de instrucción siempre que una operación sólo tenga dos operandos en registros diferentes, se podría reducir el ancho de banda de las instrucciones necesarias para un programa. Modificar el simulador DLX para contar la frecuencia de las operaciones registro-registro con sólo dos operandos en diferentes registros. Utilizando el benchmark que se proporciona con el simulador, determinar cuánto más ancho de banda de instrucciones requiere DLX que DLX con el formato R_2.

4.19 [35] <D> Determinar un método para medir el CPI de una máquina —preferiblemente una de las máquinas explicadas en este capítulo o una próxima a DLX—. Utilizando los datos de mezcla de instrucciones, escoger las 10 instrucciones superiores y medir su CPI. Comparar la jerarquía de frecuencias con el tiempo empleado. Comparar sus medidas con los números mostrados en el Apéndice D. Intentar explicar cualquier diferencia en la variación del tiempo frente a la frecuencia y entre sus medidas y las del Apéndice D.

4.20 [35] <4.5,4.6> ¿Cuáles son los beneficios de los modos de direccionamiento más potentes? Suponer que los tres modos de direccionamiento del VAX —autoincremento, desplazamiento diferido y escalado— se añaden a DLX. Cambiar el compilador C para incorporar el uso de estos modos. Medir el cambio en el número de instrucciones con estos nuevos modos para algunos programas de benchmark. Comparar las mezclas de instrucciones con las de DLX estándar. Comparar los patrones utilizados con los del VAX que están en la Figura 4.23

4.21 [35/35/30] <4.5,4.6> ¿En cuánto reduce la flexibilidad de las instrucciones memoria-memoria el número de instrucciones comparado con una máquina de carga/almacenamiento? La siguiente tarea de programación le ayudará a descubrir la respuesta:

a) [35] Suponer que DLX tiene un formato de instrucción que permite que uno de los operandos fuente esté en memoria. Modificar el generador de código C para que DLX utilice este nuevo tipo de instrucción. Utilizar diversos programas C para medir la efectividad de su cambio. ¿Cuántas más instrucciones requiere DLX frente a esta nueva máquina que parece ser muy próxima al 360? ¿Con qué frecuencia se utiliza el formato registro-memoria? ¿Cómo difieren las mezclas de instrucciones de las de la Sección 4.6?

b) [35] Suponer que DLX tiene formatos de instrucción que permiten que cualquier operando (o los tres) tenga referencias a memoria. Modificar el generador de código C para que DLX utilice estos nuevos formatos de instrucción. Utilizar diversos programas para medir la utilización de estas instrucciones. ¿Cuántas más instrucciones requiere DLX frente a esta nueva máquina que parece ser muy próxima al VAX? ¿Cómo difieren las mezclas de instrucciones de las de la Sección 4.6? ¿Cuántos operandos de memoria tiene la instrucción media?

c) [30] Diseñar un formato de instrucción para las máquinas descritas en las Partes a y b; comparar el ancho de banda de las instrucciones dinámicas requeridas para estas dos máquinas frente a DLX.

4.22 [40] <4.6> Algunos fabricantes todavía no se han dado cuenta del valor de medir mezclas de repertorios de instrucciones. Quizá usted pueda ayudarles. Escoja una máquina para la cual una herramienta de este tipo no está disponible. Construir una para esa máquina. Si la máquina tiene un modo paso a paso *(single-step)* —como en el VAX o en el 8086— se puede utilizar para crear la herramienta. En otro caso, una conversión del código objeto, como la utilizada en el sistema del compilador MIPS [Chow, 1986] podría ser más apropiada. Si mide la actividad de una máquina utili-

zando los benchmarks de este texto (GCC, Spice y TeX), y está dispuesto a compartir los resultados, por favor contactar con el editor.

4.23 [25] <E> ¿Cómo afectan al rendimiento las variaciones del repertorio de instrucciones entre las máquinas RISC explicadas en el Apéndice E. Escoger como mínimo tres programas pequeños (por ejemplo, uno de clasificación), y codificarlos en DLX y otros dos lenguajes ensambladores. ¿Cuál es la diferencia resultante en el recuento de instrucciones?

4.24 [40] <E> Seleccionar una de las máquinas explicadas en el Apéndice E. Modificar el generador de código DLX y el simulador DLX para generar código y simular la máquina seleccionada. Utilizando benchmarks, medir las diferencias en el recuento de instrucciones entre DLX y la máquina que se ha escogido.

Al analizar las funciones del dispositivo contemplado, se sugieren por sí mismas ciertas distinciones clasificatorias inmediatamente. Primero: como el dispositivo es principalmente un computador tendrá que realizar las operaciones elementales de aritmética más frecuentemente... una parte de aritmética central del dispositivo probablemente tendrá que existir... Segundo: el control lógico del dispositivo, es decir, el secuenciamiento propio de sus operaciones, puede ser realizado de forma más eficiente por un órgano central de control.

John von Neumann, *Primer borrador de un informe sobre el «EDVAC» (1945)*

5.1. Introducción

5.2. Camino de datos del procesador

5.3. Pasos básicos de ejecución

5.4. Control cableado

5.5. Control microprogramado

5.6. Interrupciones y otros enredos

5.7. Juntando todo: control para DLX

5.8. Falacias y pifias

5.9. Observaciones finales

5.10. Perspectiva histórica y referencias

Ejercicios

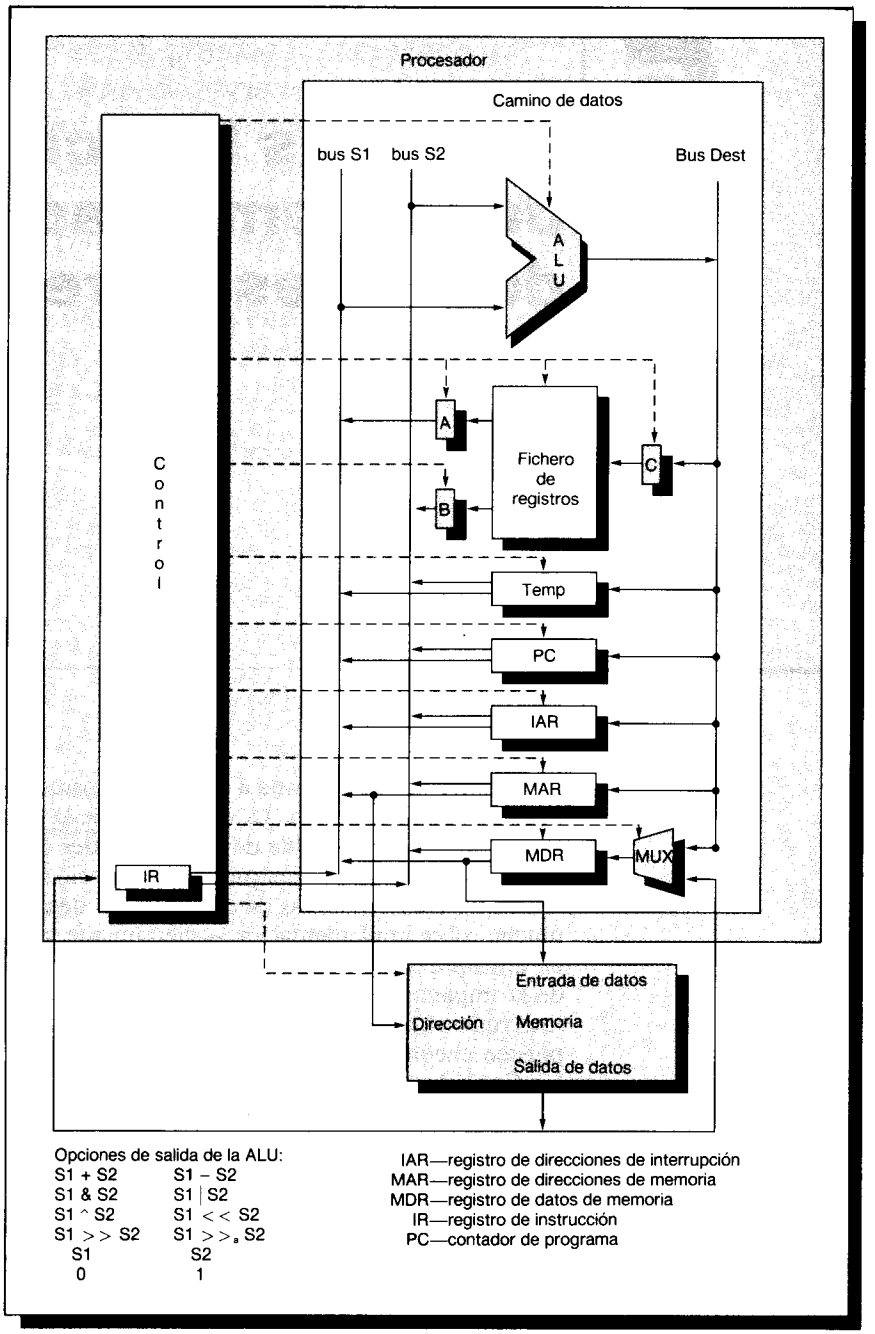
5 Técnicas básicas de implementación de procesadores

5.1. Introducción

La arquitectura da la forma a un edificio, pero la carpintería determina la calidad de su construcción. La carpintería de la computación es la implementación, que es responsable de dos de los tres componentes del rendimiento: CPI (ciclos de reloj por instrucción) y duración del ciclo de reloj.

En las cuatro décadas de construcción de computadores se ha aprendido mucho sobre implementación —ciertamente más de lo que se puede explicar en un capítulo—. Nuestro objetivo, en este capítulo, es dar los fundamentos de la implementación de procesadores, poniendo énfasis en el control y las interrupciones. (En este capítulo se ignora el punto flotante; los lectores lo pueden encontrar en el Apéndice A.) Aunque parte del material es sencillo, los Capítulos 6 y 7 se basan en estos fundamentos y muestran el camino a computadores más rápidos. (Si se trata de una revisión, examinar rápidamente las Secciones 5.1 a 5.3 y echar un vistazo a los ejemplos de la Sección 5.7, que comparan el rendimiento del control cableado frente al microprogramado para DLX.)

Von Neumann dividió el computador en componentes básicos, y estos componentes permanecen todavía: la CPU o *procesador*, es el núcleo del computador y contiene todo excepto memoria, entrada y salida. El procesador se divide además en computación y control.


Procesador
Camino de datos
bus S1
bus S2
Bus Dest
A
L
U
A
B
C
Fichero de registros
Control
Temp
PC
IAR
MAR
MDR
MUX
IR
Entrada de datos
Dirección
Memoria
Salida de datos
Opciones de salida de la ALU:
S1 + S2
S1 – S2
S1 & S2
S1 | S2
S1 ^ S2
S1 << S2
S1 >> S2
S1 >>a S2
S1
S2
0
1
IAR—registro de direcciones de interrupción
MAR—registro de direcciones de memoria
MDR—registro de datos de memoria
IR—registro de instrucción
PC—contador de programa

5.2. Camino de datos del procesador

Hoy día, el órgano «aritmético» de von Neumann se denomina *camino de datos* (datapath). Consta de unidades de ejecución, como p. e. unidades aritmético-lógicas (ALU) o desplazadores, registros, y caminos de comunicación entre ellos, como ilustra la Figura 5.1. Desde la perspectiva del programador, el camino de datos contiene la mayor parte del *estado* del procesador —la información que se debe guardar cuando se suspende la ejecución de un programa y después restaurar para que continúe ejecutándose—. Además de los registros de propósito general visibles al usuario, el estado incluye el contador de programa (PC), el registro de direcciones de interrupción (IAR), y el registro de estado del programa; el último contiene todos los señalizadores de estado para una máquina, como habilitación de interrupciones, códigos de condición, etc.

Como una implementación se crea para una tecnología hardware específica, es la implementación la que impone la duración del ciclo de reloj. La duración del ciclo de reloj está determinada por los circuitos más lentos que operan durante un período de ciclo de reloj —en el procesador, el camino de datos frecuentemente tiene ese honor—. El camino de datos predominará también en el coste del procesador, necesitando normalmente la mitad de los transistores y la mitad del área del procesador. Aunque haga toda la computación, afecte al rendimiento y predomine en el coste, el camino de datos es la parte de diseño más simple del procesador.

Algunos sostienen que la gran cantidad de papel gastado en artículos sobre diseños de ALU y esquemas de arrastre rápidos son responsables de la tala de nuestros bosques, que los artículos sobre diseños de circuitos para registros con

FIGURA 5.1 (Ver página adjunta.) Un procesador típico dividido en control y camino de datos, más memoria. Los caminos para el control están en líneas a trazos y los caminos para la transferencia de datos están en líneas continuas. El procesador utiliza tres buses: S1, S2 y Dest. La operación fundamental del camino de datos es leer operandos del fichero de registros, operar sobre ellos en la ALU, y después volver a almacenar el resultado. Como el fichero de registros no necesita ser leído ni escrito en cada ciclo de reloj, la mayoría de los diseñadores siguen la advertencia de hacer el caso frecuente rápido, descomponiendo esta secuencia en múltiples ciclos de reloj y haciendo más corto el ciclo de reloj. Por ello, en este camino de datos hay cerrojos en las dos salidas del fichero de registros (denominadas A y B) y un cerrojo en la entrada (C). El fichero de registros contiene los 32 registros de propósito general de DLX. (El registro 0 del fichero de registros siempre tiene el valor 0, coincidiendo con la definición de registro 0 en el repertorio de instrucciones de DLX.) El contador de programa (PC) y el registro de dirección de interrupciones (IAR) son también parte del estado de la máquina. Hay también registros, no parte del estado, utilizados en la ejecución de instrucciones: registro de direcciones de memoria (MAR), registro de datos de memoria (MDR), registro de instrucción (IR) y registro temporal (Temp). El registro Temp es un registro disponible para almacenamiento temporal para que el control realice algunas instrucciones de DLX. Observar que el único camino desde los buses S1 y S2 al bus Dest es a través de la ALU.

múltiples puertos de lectura y escritura son, ligeramente, menos culpables. Aunque esto es, seguramente, una exageración, hay numerosas opciones. (Ver el Apéndice A, Sección A.8, para algunos esquemas de arrastre.) Dados los recursos disponibles y los objetivos de coste y rendimiento deseados, es trabajo del diseñador seleccionar el mejor estilo de ALU, el número adecuado de puertos del fichero de registros y después marchar hacia adelante.

5.3 Pasos básicos de ejecución

Antes de explicar el control, revisemos primero los pasos de ejecución de las instrucciones. Para el repertorio de instrucciones de DLX (excluyendo el punto flotante), todas las instrucciones pueden descomponerse en cinco pasos básicos: búsqueda, decodificación, ejecución, acceso a memoria y escritura del resultado. Cada paso puede emplear uno o varios ciclos de reloj en el procesador mostrado en la Figura 5.1. Aquí están los cinco pasos (ver las tablas del libro para revisar la notación del lenguaje de transferencia de registros):

1. Paso de búsqueda de instrucción:

   ```
   MAR ← PC; IR ← M[MAR]
   ```

   Operación. Transfiere el PC (al MAR) y ubica la instrucción de memoria en el registro de instrucción. El PC es transferido al MAR porque éste tiene una conexión con la dirección de memoria de la Figura 5.1, pero el PC no.
2. Paso de búsqueda del registro/decodificación de la instrucción:

   ```
   A ← Rs1; B ← Rs2; PC ← PC + 4
   ```

   Operación. Decodifica la instrucción y accede al fichero de registros para leer los registros. Además, incrementa el PC para que señale a la siguiente instrucción.

La decodificación puede hacerse en paralelo con la lectura de los registros, lo que significa que los valores de dos registros son enviados a los cerrojos A y B **antes** que se decodifique la instrucción. Esto puede verse observando el formato de la instrucción DLX (Fig. 4.19), que muestra que los *registros fuente* tienen siempre la misma posición en una instrucción. Así, los registros pueden ser leídos porque los especificadores de registros no son ambiguos. (Esta técnica se conoce como *decodificación de campo fijo.*) Como la parte inmediata de una instrucción es también idéntica en cada formato DLX, el inmediato de signo extendido también se calcula durante este paso, en caso que sea necesario, para el paso siguiente.

3. Paso de dirección efectiva/ejecución:

   La ALU opera sobre los operandos preparados en el paso anterior, realizando una de tres funciones, dependiendo del tipo de instrucción DLX.

Referencia a memoria:

$$\texttt{MAR} \leftarrow \texttt{A} + (\texttt{IR}_{16})^{16} \# \# \texttt{IR}_{16..31} \; ; \; \texttt{MDR} \leftarrow \texttt{Rd}$$

Operación. La ALU suma los operandos para formar la dirección efectiva, y se carga el MDR.

Instrucción ALU:

$$\texttt{ALUoutput} \leftarrow \texttt{A } op \texttt{ (B or } (\texttt{IR}_{16})^{16} \# \# \texttt{IR}_{16..31})$$

Operación. La ALU realiza la operación especificada por el código de operación sobre el valor de A (Rs1) y sobre el valor de B o el inmediato de signo extendido.

Salto/bifurcación:

$$\texttt{ALUoutput} \leftarrow \texttt{PC} + (\texttt{IR}_{16})^{16} \# \# \texttt{IR}_{16..31}; \; \texttt{cond} \leftarrow \texttt{(A } op \texttt{ 0)}$$

Operación. La ALU suma el PC al valor inmediato de signo extendido (16 bits para saltos y 26 bits para bifurcaciones) para calcular la dirección destino del salto. Para saltos condicionales, se examina un registro, que ha sido leído en el paso anterior, para decidir si se debe insertar esta dirección en el PC. La operación de comparación *op* es el operador relacional determinado por el código de operación, por ejemplo, *op* es «==» para la instrucción BEQZ.

La arquitectura de carga/almacenamiento de DLX significa que la dirección efectiva y los pasos de ejecución se pueden combinar en un solo paso, ya que ninguna instrucción necesita calcular una dirección y, además, realizar una operación sobre los datos. Otras instrucciones enteras no incluidas antes son JAL y TRAP. Estas son análogas a las bifurcaciones, excepto que JAL almacena la dirección de vuelta en R31 y TRAP en el IAR.

4. Paso de completar salto/acceso a memoria: las únicas instrucciones DLX activas en este paso son cargas, almacenamientos, saltos y bifurcaciones.

Referencia a memoria:

$$\texttt{MDR} \leftarrow \texttt{M[MAR]} \text{ o } \texttt{M[MAR]} \leftarrow \texttt{MDR}$$

Operación. Accede a memoria si es necesario. Si la instrucción es una carga, devuelve el dato desde memoria; si es un almacenamiento, entonces escribe el dato en memoria. En cualquier caso la dirección utilizada es la calculada durante el paso anterior.

Salto:

$$\texttt{if (cond) PC} \leftarrow \texttt{ALUoutput (branch)}$$

Operación. Si la instrucción salta, el PC es sustituido por la dirección destino del salto. Para bifurcaciones la condición siempre es cierta.

5. Paso de postescritura (write-back):

```
Rd ← ALUoutput or MDR
```

Operación. Escribir el resultado en el fichero de registros tanto si proviene del sistema de memoria como de la ALU.

Ahora que hemos dado una visión general del trabajo que se debe realizar para ejecutar una instrucción, estamos preparados para examinar las dos técnicas principales de implementación del control.

5.4 Control cableado

Si el diseño del camino de datos es sencillo, entonces alguna parte del diseño del procesador debe ser difícil, y esa parte es el control. Especificar el control es el camino crítico de cualquier proyecto de computadores; y es donde más errores se encuentran cuando se depura un nuevo computador. El control puede simplificarse —la forma más fácil es simplificar el repertorio de instrucciones—, pero ése es el tema de los Capítulos 3 y 4.

Dada una descripción de un repertorio de instrucciones, como la descripción de DLX en el Capítulo 4, y un diseño de un camino de datos, como el de la Figura 5.1, el paso siguiente es definir la unidad de control. La unidad de control indica al camino de datos lo que tiene que hacer, cada ciclo de reloj, durante la ejecución de las instrucciones. Esto se especifica normalmente por un *diagrama de estados finitos.* Cada estado corresponde a un ciclo de reloj, y las operaciones que se van a realizar durante el ciclo de reloj se escriben en ese estado. Cada instrucción tarda varios ciclos de reloj en completarse; el Capítulo 6 muestra cómo solapar la ejecución para reducir los ciclos de reloj por instrucción hasta uno.

La Figura 5.2 muesta parte de un diagrama de estados finitos para los dos primeros pasos de la ejecución de instrucciones en DLX. El primer paso se extiende a los tres estados: el registro de direcciones de memoria es cargado con el contenido del PC durante el primer estado, el registro de instrucción es cargado desde la memoria durante el segundo estado, y el contenido del PC es incrementado en el tercer estado. Este tercer estado también realiza el paso 2, cargando los dos operandos de registro, Rs1 y Rs2, en los registros A y B para utilizarlos en estados posteriores. En la Sección 5.7 se muestra el diagrama completo de estados finitos para DLX.

Convertir un diagrama de estados en hardware es el siguiente paso. Las alternativas para hacer esto dependen de la tecnología de implementación. Una forma de acotar la complejidad del control es mediante el producto

Estados · Entradas de control · Salidas de control

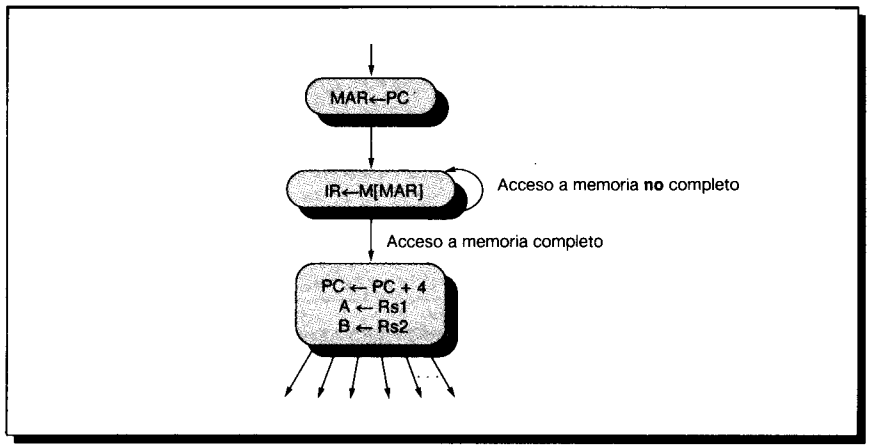


FIGURA 5.2 El nivel superior del diagrama de estados finitos en DLX. Se muestran los dos primeros pasos de la ejecución de una instrucción, búsqueda de la instrucción y decodificación/búsqueda de registros. El segundo estado se repite hasta que la instrucción es obtenida de memoria. Los tres últimos pasos de la ejecución de la instrucción —dirección ejecución/efectiva, acceso de memoria y postescritura— se encuentran en la Sección 5.7.

donde

Estados = el número de estados en un controlador de una máquina de estados finitos;

Entradas de control = el número de señales examinadas por la unidad de control;

Salida de control = el número de salidas de control generadas por el hardware, incluyendo los bits para especificar el siguiente estado.

La Figura 5.3 muestra una organización para el control de DLX. Digamos que el diagrama de estados finitos de DLX contiene 50 estados, por lo que se requieren 6 bits para representar el estado. Por tanto, las entradas de control deben incluir estos 6 bits, algunos bits más (p. e., 3) para seleccionar condiciones desde el camino de datos y la unidad de interfaz de memoria, más los bits de la instrucción. Los especificadores de registros e inmediatos se envían directamente al hardware, así no hay necesidad de enviar los 32 bits de las instrucciones DLX como entradas de control. El código de operación de DLX es de 6 bits, y solamente se utilizan 6 bits del código de operación extendido (el campo «func»), haciendo un total de 12 bits de la instrucción para entradas de control. Dadas esas entradas, el control se puede especificar con una gran tabla. Cada fila de la tabla contiene los valores de las líneas de control, para realizar las operaciones requeridas por ese estado, y suministra el número del estado siguiente. Supondremos que hay 40 líneas de control.

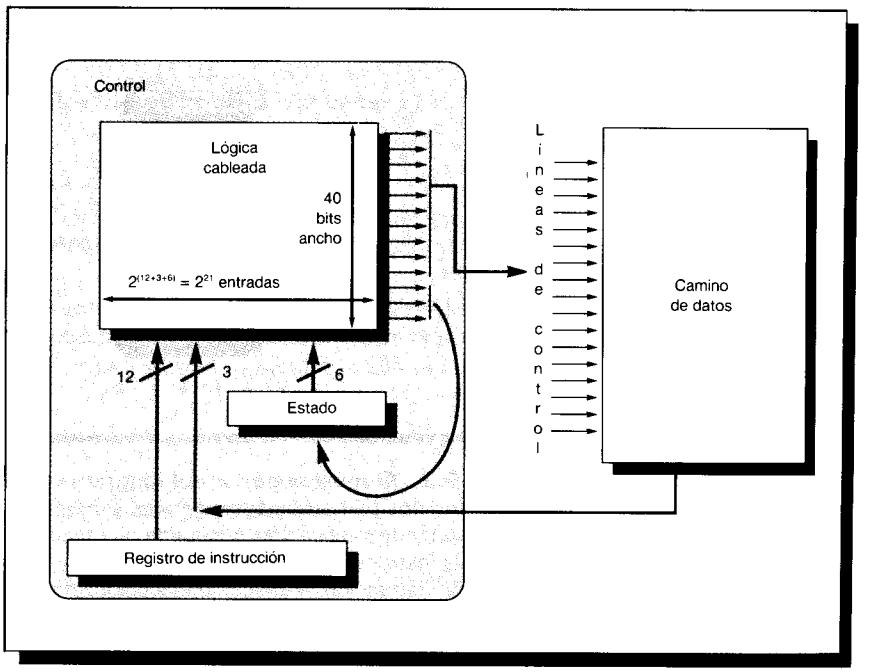


FIGURA 5.3 Control especificado como tabla para un sencillo repertorio de instrucciones. Las entradas de control constan de 6 líneas de entrada para los 50 estados ($\log_2 50 = 5{,}6$), 3 entradas del camino de datos, y 12 bits de instrucción (el código de op de 6 bits más 6 bits del código de op extendido). El número de líneas de control se supone que es 40.

Reducción del coste hardware del control cableado

La implementación más rápida de una tabla es con una *memoria de sólo lectura* (ROM). En este ejemplo se necesitarán 2^{21} palabras, cada una de 40 bits (¡10 MB of ROM!), se necesitará mucho tiempo antes que podamos permitirnos esta cantidad de hardware para el control. Afortunadamente, una parte pequeña de esta tabla tiene información única, por ello su tamaño puede reducirse manteniendo solamente las filas con información única —a costa de una decodificación de direcciones más complicada. Esta construcción hardware se denomina *array de lógica programada* (PLA) y reduce, esencialmente, el hardware de 2^{21} a 50 palabras, aunque incrementando la lógica de decodificación de direcciones. Los programas de diseño asistido por computador pueden reducir, aún más, los requerimientos hardware, minimizando el número de «minterms», que es esencialmente el número de filas únicas. En máquinas reales, a veces es prohibitivo incluso un único PLA, ya que su tamaño crece como el producto de las filas únicas por la suma de las entradas y salidas. En tal caso, se factoriza una gran tabla en varios PLA pequeños, cuyas salidas son multiplexadas para escoger el control correcto.

Aunque parezca mentira, la numeración de los estados del diagrama de estados finitos puede marcar una diferencia en el tamaño del PLA. La idea aquí es tratar de asignar números de estados similares a estados que realicen operaciones similares. Diferenciar los patrones de bits que representan el número del estado solamente en un bit —digamos 010010 y 010011— hace las entradas próximas para la misma salida. Hay también programas de diseño asistido por computador para ayudar a este *problema de asignación de estados.*

Como los bits de la instrucción también son entradas al PLA de control, éstos pueden afectar la complejidad del PLA de la misma manera que la numeración de los estados. Por tanto, debe tenerse cuidado cuando se seleccionen los códigos de operación, ya que pueden afectar el coste del control.

Los lectores interesados en ir más allá de este diseño tienen como referencias muchos textos excelentes sobre diseño lógico.

Rendimiento del control cableado

Cuando se diseña el control detallado para una máquina, se quiere minimizar el CPI medio, el ciclo de reloj, la cantidad de hardware para especificar el control y el tiempo para desarrollar un controlador correcto. Minimizar el CPI significa reducir el número medio de estados en el camino de ejecución de una instrucción, ya que cada ciclo de reloj corresponde a un estado. Normalmente, esto se logra haciendo cambios en el camino de datos para combinar o eliminar estados.

Ejemplo

Cambiamos el hardware para que el PC se pueda utilizar directamente para direccionar memoria sin necesidad de pasar primero por el MAR. ¿Cómo habría que cambiar el diagrama de estados para aprovechar esta mejora y cuál sería el cambio en el rendimiento?

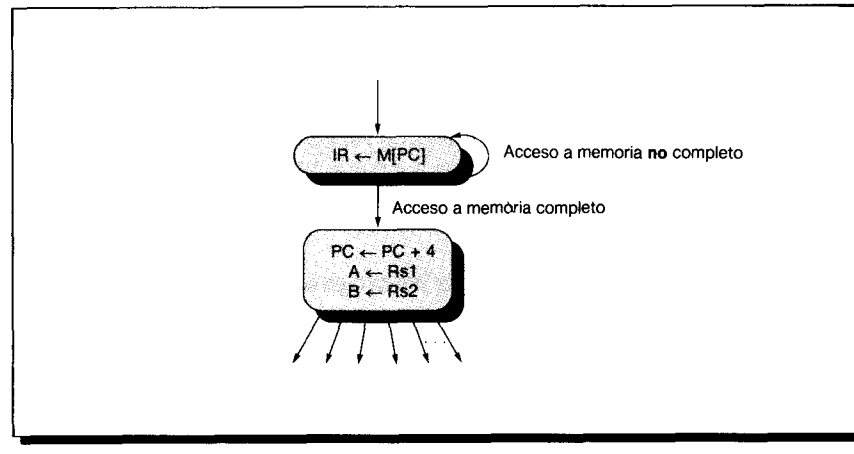


FIGURA 5.4 La Figura 5.2 modificada al eliminar la carga del MAR desde el PC en el primer estado y al utilizar directamente el valor del PC para la dirección de memoria.

Respuesta En la Figura 5.2 vemos que el primer estado copia el PC en el MAR. El cambio de hardware propuesto hace ese estado innecesario, y la Figura 5.4 muestra el diagrama de estados modificado apropiadamente. Este cambio ahorra un ciclo de reloj de cada instrucción. Suponer que el número medio de CPI, originalmente, era 7. Siempre que no haya impacto en la duración del ciclo de reloj, este cambio hará a la máquina un 17 por 100 más rápida.

5.5 Control microprogramado

Después de construir en 1949 el primer computador de programa almacenado, operativo a escala completa, Maurice Wilkes reflexionó sobre el proceso. Las entradas/salidas eran fáciles —los teletipos podían comprarse directamente en la compañía de telégrafos. La memoria y el camino de datos eran altamente repetitivos, y eso hacía las cosas más simples. Pero el control no era fácil ni repetitivo, así que Wilkes trató de descubrir una mejor forma de diseñarlo. Su solución fue convertir la unidad de control en un computador en miniatura, que tuviera una tabla para especificar el control del camino de datos y una segunda tabla para determinar el flujo de control al nivel «micro». Wilkes llamó a esta invención *microprogramación* y antepuso el prefijo «micro» a los términos tradicionales utilizados a nivel de control: microinstrucción, microcódigo, microprograma, etc. (Para evitar confusiones, a veces se utiliza el prefijo «macro» para describir el alto nivel, p. e., macroinstrucción y macroprograma.) Las microinstrucciones especifican todas las señales de control del camino de datos, más la posibilidad de decidir condicionalmente qué microinstrucción se debe ejecutar a continuación. Como sugiere el nombre de «microprogramación», una vez que se diseñan el camino de datos y la memoria para las microinstrucciones, el control se convierte básicamente en una tarea de programación; es decir, la tarea de escribir un intérprete para el repertorio de instrucciones. La invención de la microprogramación posibilitó que el repertorio de instrucciones pudiera cambiarse alterando el contenido de la memoria de control sin tocar el hardware. Como veremos en la Sección 5.10, esta posibilidad jugó un papel importante en la familia 360 —lo que fue una sorpresa para sus diseñadores.

La Figura 5.5 muestra una organización para un sencillo control microprogramado. La estructura de un microprograma es muy similar a la del diagrama de estados, con una microinstrucción para cada estado del diagrama.

ABC de la microprogramación

Aunque al hardware no le importe cómo se agrupen las líneas de control de una microinstrucción, las líneas de control que realizan funciones afines se colocan tradicionalmente juntas para mejor comprensión. Los grupos de las líneas de control afines se denominan *campos* y tienen nombres en un formato de microinstrucción. La Figura 5.6 muestra un formato de microinstrucción con ocho campos, cada nombre de campo refleja su función. La microprogramación puede considerarse la manera de suministrar el patrón de bits

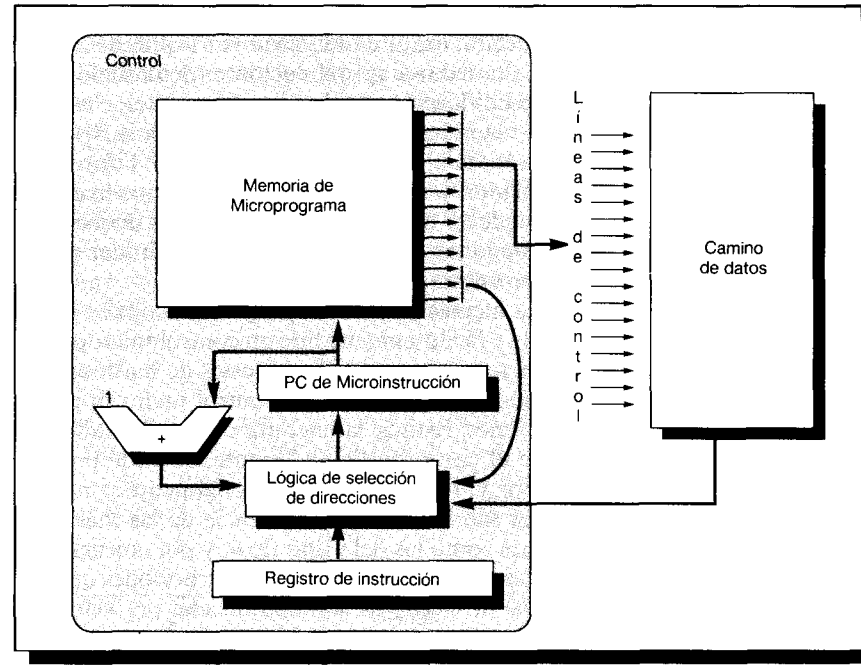


FIGURA 5.5 Una máquina microcodificada básica. De forma distinta a la Figura 5.3, hay un incrementador y lógica especial para seleccionar la siguiente microinstrucción. Hay dos aproximaciones para especificar la siguiente microinstrucción: utilizar un contador de programa de microinstrucciones, como muestra la figura, o incluir la dirección de la siguiente microinstrucción en cada microinstrucción. La memoria del microprograma a veces se denomina ROM porque la mayor parte de las primeras máquinas utilizaron una ROM para almacenar el control.

adecuado a cada campo, muy parecido a la programación de «macroinstrucciones» del lenguaje ensamblador.

Como indica la Figura 5.5, se puede utilizar un contador de programa para proporcionar la siguiente microinstrucción, pero algunos computadores dedican un campo, en cada microinstrucción, para la dirección de la siguiente instrucción. Algunos proporcionan incluso múltiples campos de la dirección siguiente para manipular saltos condicionales.

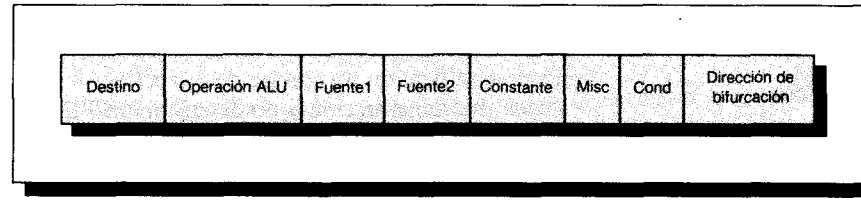


FIGURA 5.6 Microinstrucción ejemplo con ocho campos (utilizada por DLX en la Sección 5.7).

Aunque los saltos condicionales se puedan utilizar para decodificar una instrucción examinando cada vez un bit del código de operación; en la práctica, esta tediosa aproximación es demasiado lenta. El esquema más sencillo de decodificación rápida de una instrucción es colocar el código de operación, de la macroinstrucción, en el medio de la dirección de la siguiente microinstrucción, análogo a una instrucción de bifurcación indexada en lenguaje ensamblador. Una aproximación más refinada es utilizar el código de operación para indexar una tabla que contenga las direcciones de la microinstrucción que proporcionen la siguiente dirección, similar a una tabla de bifurcaciones en código ensamblador.

La memoria del microprograma, o *memoria de control*, es el hardware más visible y fácilmente medido en control microprogramado; por tanto, es el foco de las técnicas para reducir costes de hardware. Las técnicas para recortar el tamaño de la memoria de control incluyen reducir el número de microinstrucciones, reducir la anchura de cada microinstrucción, o ambas cosas. Así como el coste, tradicionalmente, se mide por el tamaño de la memoria de control, el rendimiento tradicionalmente se mide por el CPI. El microprogramador sabio conoce la frecuencia de las macroinstrucciones utilizando estadísticas como las del Capítulo 4, y por consiguiente, conoce dónde y cómo se emplea mejor el tiempo —las instrucciones que demandan la mayor parte del tiempo de ejecución son optimizadas por velocidad, y las otras por espacio.

En cuatro décadas de la historia de la microprogramación ha habido gran variedad de términos y técnicas para microprogramar. En efecto, desde 1968 un «workshop» trata anualmente este tema. Antes de observar algunos ejemplos, recordemos que las técnicas de control —cableado o microprogramado— se juzgan por su impacto en el coste del hardware, duración del ciclo de reloj, CPI, y tiempo de desarrollo. En las dos secciones siguientes estudiaremos cómo pueden disminuirse los costes de hardware reduciendo el tamaño de la memoria de control. Primero examinaremos dos técnicas para reducir la anchura de las microinstrucciones, y después otra para reducir su número.

Reducción del coste de hardware codificando las líneas de control

El enfoque ideal para reducir la memoria de control es escribir primero el microprograma completo en una notación simbólica y después medir cómo se inicializan las líneas de control en cada microinstrucción. Tomando medidas podemos reconocer los bits de control que se pueden codificar en un campo más pequeño. Si, por ejemplo, en una microinstrucción no se ponen a 1 simultáneamente más de una línea de entre 8, entonces, estas ocho líneas se pueden codificar en un campo de 3 bits ($\log_2 8 = 3$). Este cambio ahorra 5 bits en cada microinstrucción y no deteriora el CPI, aunque esto signifique el coste hardware extra de un decodificador 3-a-8 necesario para generar las 8 líneas de control originales. No obstante, ahorrar 5 bits de anchura de la memoria de control habitualmente superará el coste del decodificador.

Esta técnica de reducir la anchura del campo se denomina *codificación*. Para ahorrar espacio adicional, las líneas de control pueden codificarse juntas

cuando se inicializan ocasionalmente en la misma microinstrucción; entonces se requieren dos instrucciones en lugar de una cuando ambas deban inicalizarse. Siempre que esto no ocurra en rutinas críticas, microinstrucciones menos anchas pueden justificar algunas palabras extras de memoria de control.

Hay peligros al codificar. Por ejemplo, si una línea de control está en el camino crítico de temporización, o si el hardware que controla está en el camino crítico, entonces afectará la duración del ciclo de reloj. Un peligro más sutil es que, en una revisión posterior del microcódigo, se puedan encontrar situaciones donde las líneas de control se inicialicen en la misma microinstrucción, bien dañando el rendimiento o requiriendo cambios en el hardware que puedan alargar el ciclo de desarrollo.

Ejemplo

Suponer que queremos codificar los tres campos que especifican un registro en un bus —Destino, Fuente1 y Fuente2— en el formato de microinstrucción de DLX de la Figura 5.6. ¿Cuántos bits de la memoria de control se pueden ahorrar frente a los campos no codificados?

Número	Destino	Fuente1/Fuente2
0	(Ninguno)	A/B
1	C	Temp
2	Temp	PC
3	PC	IAR
4	IAR	MAR
5	MAR	MDR
6	MDR	IR (16-bit inm)
7	—	IR (26-bit inm)
8	—	Constante

FIGURA 5.7 Las fuentes y destinos especificados en los tres campos de la Figura 5.6 de la descripción del camino de datos de la Figura 5.1. A y B no son entradas separadas porque A puede sólo transferir en el bus S1 y B sólo puede transferir en el bus S2 (ver Fig. 5.1). La última entrada de la tercera columna, Constante, es utilizada por el control para especificar una constante necesaria en una operación de la ALU (por ejemplo, 4). Ver Sección 5.7 para su uso.

Respuesta

La Figura 5.7 lista los registros para cada fuente y destino del camino de datos en la Figura 5.1. Observar que el campo destino debe poder especificar que no se modifica nada. Sin codificación, los 3 campos requieren 7 + 9 + 9, ó 25 bits. Como $\log_2 7 \approx 2{,}8$ y $\log_2 9 \approx 3{,}2$, los campos codificados requieren 3 + 4 + 4, o 11 bits. Por tanto, codificar estos tres campos ahorra 14 bits por microinstrucción.

Reducción del coste de hardware con múltiples formatos de microinstrucción

Las microinstrucciones pueden hacerse todavía más cortas si se descomponen en formatos diferentes y se añade un código de operación o *campo de formato* para distinguirlas. El campo de formato asigna a todas las líneas de control, no especificadas, sus valores implícitos, de esta forma no cambia nada en la máquina, y es similar al código de operación de una macroinstrucción.

Reducir los costes del hardware utilizando campos de formato tiene su propio coste de rendimiento —ejecución de más microinstrucciones—. Generalmente, un microprograma utilizando un único formato de microinstrucción puede especificar cualquier combinación de operaciones en un camino de datos y necesitará menos ciclos de reloj que un microprograma compuesto de microinstrucciones restringidas. Las máquinas más delgadas (menos anchas) son más baratas porque los «chips» de memoria son también delgados y altos: se emplean muchos menos «chips» en una memoria de 16K palabras de 24 bits que en una memoria de 4K palabras de 96 bits. (Cuando la memoria de control está en el chip del procesador, esta ventaja hardware desaparece.)

Este enfoque de «delgada pero alta» se denomina con frecuencia *microcódigo vertical*, mientras que la aproximación «ancha pero corta» se denomina *microcódigo horizontal*. Debe notarse que los términos «microcódigo vertical» y «microcódigo horizontal» no tienen definición universal —los diseñadores del 8086 consideraron que su microinstrucción de 21 bits era más horizontal que la de otros computadores monochips de la época. Los términos afines *codificado máximamente y codificado mínimamente* conducen a menor confusión.

La Figura 5.8 representa el tamaño de la memoria de control frente a la longitud de las microinstrucciones para tres familias de computadores. Observar que, para cada familia, el tamaño total es similar, aun cuando las longitudes varíen en un factor de 6. Como regla, las memorias de control mínimamente codificadas utilizan más bits, y el aspecto delgado pero alto de los chips de memoria, significa que las memorias de control máximamente codificadas tienen, naturalmente, más entradas. A veces los diseñadores de las máquinas mínimamente codificadas no tienen opción de chips RAM más cortos, haciendo que máquinas de microinstrucciones anchas finalicen con muchas palabras de memoria de control. Como los costes del hardware no son más bajos si el microcódigo no utiliza todo el espacio de la memoria de control, las máquinas de esta clase pueden acabar con memorias de control mayores que las esperadas en otras implementaciones. Las RAM ECL disponibles para construir el VAX 8800, por ejemplo, resultaron en una memoria de control de 2000 Kbits.

Reducción del coste de hardware añadiendo control cableado para compartir microcódigo

El otro enfoque para reducir memoria de control es reducir el número de microinstrucciones en lugar de su tamaño. Las microsubrutinas proporcionan un

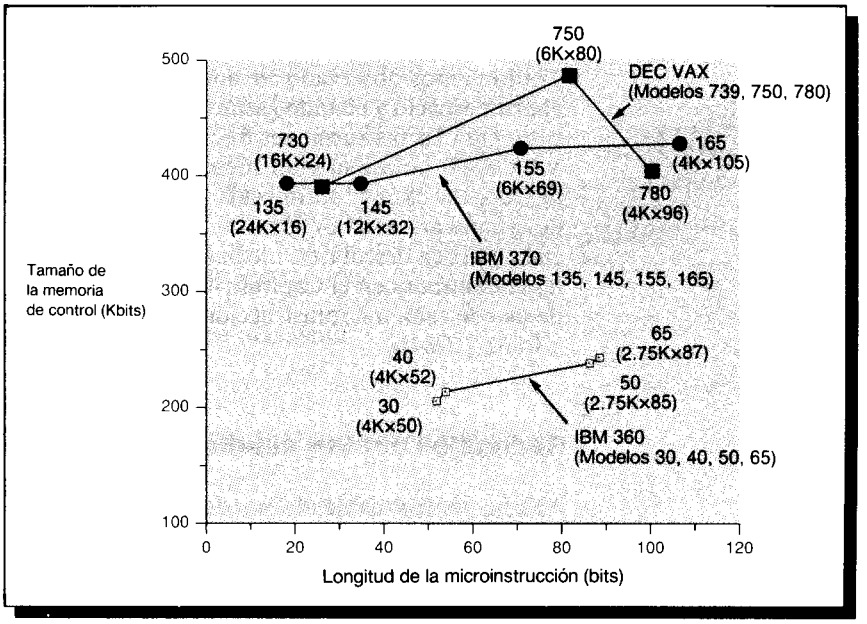


FIGURA 5.8 Tamaño de la memoria de control frente a la longitud de las microinstrucciones para 11 modelos de computadores. Cada punto está identificado por la longitud y anchura de la memoria de control (sin incluir paridad). Los modelos seleccionados de cada familia son los construidos aproximadamente al mismo tiempo: IBM 360, modelos 30, 40, 50 y 65 construidos todos en 1965; IBM 370, modelos 145, 155 y 165 construidos en 1971, siguiendo el 135 en el año siguiente; y el modelo VAX 780 fue construido en 1978 seguido por el 750 en 1980 y el 730 en 1982.

enfoque, así como las rutinas con secuencias «finales» comunes compartiendo código mediante bifurcaciones.

Muchas comparticiones pueden hacerse con asistencia de control cableado. Por ejemplo, muchas microarquitecturas permiten que los bits del registro de instrucción especifiquen el registro correcto. Otra asistencia común es utilizar partes del registro de instrucción para especificar la operación de la ALU. Cada una de estas asistencias está bajo control microprogramado y es invocada con un valor especial en el campo adecuado. El 8086 utiliza ambas técnicas, dando una rutina de 4 líneas para 32 códigos de operación. El inconveniente de añadir control cableado es que puede alargar el ciclo de desarrollo, porque no involucra programación sino que requiere un desarrollo en hardware para el diseño y la depuración.

Esta sección y las dos anteriores explicaban técnicas para reducir costes. Las siguientes secciones presentan tres técnicas para mejorar el rendimiento.

Reducción del CPI con microcódigo de casos especiales

Como hemos observado, el microprogramador prudente sabe cuándo debe ahorrar espacio y cuándo gastarlo. Un ejemplo de esto es dedicar microcódigo extra para las instrucciones frecuentes, reduciendo así el CPI. Por ejemplo, el VAX 8800 utiliza su gran memoria de control para muchas versiones de la instruccion CALLS, optimizada para guardar registros dependiendo del valor de la máscara «salva-registros». Candidatos para microcódigo de casos especiales pueden descubrirse mediante medidas de mezclas de instrucciones, como las encontradas en el Capítulo 4 o en el Apéndice B, o al contar la frecuencia de uso de cada microinstrucción en una implementación existente (ver Emer y Clark [1984]).

Reducción del CPI añadiendo control cableado

Añadir control cableado puede reducir costes, así como mejorar el rendimiento. Por ejemplo, los operandos VAX pueden estar en memoria o en registros, pero las últimas máquinas reducen el CPI al tener un código especial para las transferencias y sumas registro-registro o memoria-registro: `ADDL2 Rn, 10 (Rm)` emplea cinco ciclos o más en la 780, pero muy pocos en el 8600. Otro ejemplo está en la interfaz de memoria, donde la solución obvia es que el microcódigo examine y salte continuamente hasta que la memoria esté lista. Debido al retardo entre el instante que una condición se convierte en verdadera y el instante que se lee la siguiente microinstrucción, este enfoque puede añadir un ciclo extra a cada acceso a memoria. La importancia de la interfaz de memoria está avalada por las estadísticas de la 780 y del 8800 —el 20 por 100 de los ciclos de reloj de la 780 y el 23 por 100 del 8800 están esperando que la memoria esté lista, estas esperas se denominan detenciones *(stalls)*—. Una detención (*stall*) ocurre cuando una instrucción debe aguardar (uno o más ciclos de reloj), esperando que algún recurso esté disponible. En este capítulo se presentan las detenciones solamente cuando se espera memoria; en el siguiente capítulo veremos otras razones de su utilización.

Muchas máquinas aproximan este problema teniendo en la detención hardware una microinstrucción que trate de acceder al registro de la memoria de datos antes que se complete la operación de memoria. (Esto se puede realizar congelando la dirección de la microinstrucción para que se ejecute la misma microinstrucción hasta que se cumpla la condición.) En el instante que la referencia a memoria esté lista, se completa la microinstrucción que necesita el dato, evitando el retardo de reloj extra para acceder a la memoria de control.

Reducción del CPI por paralelismo

A veces el CPI se puede reducir con más operaciones por microinstrucción. Esta técnica, que habitualmente requiere una microinstrucción mayor, incrementa el paralelismo con más operaciones del camino de datos. Es otra carac-

terística de las máquinas denominadas horizontales. Ejemplo de esta ganancia de rendimiento puede verse en el hecho de que los modelos más rápidos de cada familia de la Figura 5.8 también tienen las microinstrucciones más anchas. Sin embargo, hacer más ancha la microinstrucción no garantiza incrementar el rendimiento. Un ejemplo donde no se consigue la ganancia potencial se encuentra en un microprocesador muy similar al 8086, con la excepción que tiene otro bus más en el camino de datos, por lo que requiere seis bits más en su microinstrucción. Esto podía haber reducido la fase de ejecución de tres a dos ciclos de reloj para muchas instrucciones populares del 8086. Desgraciadamente, estas macroinstrucciones populares se agruparon con macroinstrucciones que no podían aprovechar esta optimización, así todas tenían que correr a una frecuencia más lenta.

5.6 Interrupciones y otros enredos

El control es la parte difícil del diseño de procesadores y la parte difícil del control son las *interrupciones* —eventos distintos de los saltos que cambian el flujo normal de la ejecución de las instrucciones. Detectar condiciones de interrupción en una instrucción, con frecuencia, puede estar en el camino crítico de la temporización de una máquina, afectando posiblemente la duración del ciclo de reloj y, por tanto, el rendimiento. Sin prestar atención adecuada a las interrupciones durante el diseño, el añadir interrupciones a una implementación complicada puede incluso embrollar los trabajos, así como hacer el diseño impracticable.

Inventadas para detectar errores aritméticos y eventos de señales en tiempo real, las interrupciones, actualmente, se utilizan para una multitud de difíciles funciones. Aquí se dan 11 ejemplos:

Petición de E/S de dispositivo

Invocar un servicio del sistema operativo desde un programa de usuario

Seguimiento de la ejecución de las instrucciones

Punto de ruptura (interrupción pedida por el programador)

Desbordamiento o desbordamiento a cero aritmético

Fallo de página (Page fault) (no en memoria principal)

Accesos a memoria mal alineados (si se requiere alineación)

Violación de la protección de memoria

Utilización de una instrucción indefinida

Malfunciones del hardware

Fallo de alimentación

La ampliada responsabilidad de las interrupciones ha llevado a la confusa situación de que cada vendedor de computadores invente un término diferente para el mismo evento, como ilustra la Figura 5.9. Intel e IBM todavía llaman

	IBM 360	VAX	Motorola 680x0	Intel 80x86
Petición de dispositivo de E/S	Interrupción de entrada/salida	Interrupción de dispositivo	Excepción (autovector nivel 0...7)	Interrupción vectorizada
Invocar el servicio de sistema operativo desde un programa de usuario	Interrupción de llamada al supervisor	Excepción (trap de cambio/a modo supervisor)	Excepción (instrucción no implementada): en Macintosh	Interrupción (instrucción INT)
Ejecución de traza de instrucción	NA	Excepción (fallo de traza)	Excepción (traza)	Interrupción (trap mono paso)
Punto de ruptura	NA	Excepción (fallo de punto de ruptura)	Excepción (instrucción ilegal o punto de ruptura)	Interrupción (trap de punto de ruptura)
Desbordamiento o desbordamiento a cero aritmético	Interrupción de programa (excepción de desbordamiento o desbordamiento a cero)	Excepción (trap de desbordamiento entero o fallo de desbordamiento a cero flotante)	Excepción (errores de coprocesador de punto flotante)	Interrupción (trap de desbordamiento o excepción de unidad matemática)
Fallo de página (no en memoria principal)	NA (sólo en 370)	Excepción (fallo de traducción no válida)	Excepción (errores de unidad-gestión de memoria)	Interrupción (fallo de página)
Acceso a memoria mal alineados	Interrupción de programa (excepción de especificación)	NA	Excepción (error de dirección)	NA
Violaciones de protección de memoria	Interrupción de programa (excepción de protección)	Excepción (fallo de violación de control de acceso)	Excepción (error de bus)	Interrupción (excepción de protección)
Utilización de instrucciones indefinidas	Interrupción de programa (excepción de operación)	Excepción (fallo de código de operación privilegiado/ reservado)	Excepción (instrucción ilegal o instrucción de punto de ruptura/no implementada)	Interrupción (código de operación inválido)
Malfunciones hardware	Interrupción de comprobación de máquina	Excepción (aborto de comprobación-máquina)	Excepción (error del bus)	NA
Fallo de potencia	Interrupción de comprobación de máquina	Interrupción urgente	NA	Interrupción no enmascarable

FIGURA 5.9 Nombres de 11 clases de interrupciones en cuatro computadores. Cada evento en la IBM 360 y 80x86 se denomina *interrupción*, mientras que en el 680x0 se denomina *excepción*. VAX divide los eventos en *interrupciones* o *excepciones*. Los adjetivos *dispositivo, software* y *urgente* se utilizan con las interrupciones VAX, mientras que las excepciones VAX se subdividen en *fallos* (faults), *traps y abortos*.

Evento	Tiempo entre eventos
Interrupción de E/S	2,7 ms
Interrupción del temporizador de intervalos	10,0 ms
Interrupción software	1,5 ms
Cualquier interrupción	0,9 ms
Cualquier interrupción hardware	2,1 ms

FIGURA 5.10 Frecuencia de diferentes interrupciones en el VAX 8800 ejecutando una carga de trabajo multiusuario en el sistema de tiempo compartido VMS. Los sistemas operativos de tiempo real utilizados en los controladores intercalados pueden tener una mayor frecuencia de interrupciones que los sistemas de tiempo compartido de propósito general. (Coleccionado por Clark, Bannon y Keller [1988].)

a estos eventos *interrupciones*, pero Motorola los llama *excepciones*; y, dependiendo de las circunstancias, DEC los llama *excepciones, fallos, abortos, traps* o *interrupciones*. Para dar una idea de la frecuencia con que ocurren las interrupciones, la Figura 5.10 muestra la frecuencia en el VAX 8800.

Claramente, no hay un convenio consistente para denominar estos eventos. Por tanto, en lugar de imponer uno, revisemos las razones de los diferentes nombres. Los eventos se pueden caracterizar sobre cinco ejes independientes:

1. Síncrono frente a asíncrono. Si el evento ocurre en el mismo lugar cada vez que se ejecuta un programa con los mismos datos y ubicación de memoria, el evento es síncrono. Con la excepción de las malfunciones hardware, los eventos asíncronos están provocados por dispositivos externos al procesador y la memoria.
2. Petición de usuario frente a petición forzada. Si la tarea del usuario lo pide directamente, existe un evento requerido por el usuario.
3. Enmascarable frente a no enmascarable por el usuario. Si se puede enmascarar o inhabilitar por una tarea del usuario, el evento es enmascarable por el usuario.
4. «En instrucciones» frente a «entre instrucciones». Esta clasificación depende si el evento evita que se complete la instrucción cuando se presenta durante su ejecución —sin importar cuándo— o si se reconoce entre instrucciones.
5. Reanudar frente a terminar. Si la ejecución del programa se detiene después de la interrupción, es un evento terminal.

La tarea difícil es implementar las interrupciones que se presentan durante la ejecución de una instrucción cuando ésta se debe reanudar. Se debe invocar otro programa para guardar el estado del programa, corregir la causa de la interrupción, y después restaurar el estado del programa antes que una interrupción pueda ser tratada de nuevo.

	Síncrono frente a asíncrono	Petición de usuario frente a forzada	Enmascarable por el usuario frente a no enmascarable	«En instrucciones» frente a «entre instrucciones»	Reanudar frente a terminar
Petición de dispositivo de E/S	Asíncrono	Forzada	No enmascarable	En	Reanudar
Invocar el servicio del sistema operativo	Síncrono	Petición de usuario	No enmascarable	En	Reanudar
Seguimiento de la ejecución de las instrucciones	Síncrono	Petición de usuario	Enmascarable por el usuario	En	Reanudar
Punto de ruptura	Síncrono	Petición de usuario	Enmascarable por el usuario	En	Reanudar
Desbordamiento de aritmética entera	Síncrono	Forzada	Enmascarable por el usuario	Entre	Terminar
Desbordamiento o desbordamiento a cero de aritmética de punto flotante	Síncrono	Forzada	Enmascarable por el usuario	Entre	Reanudar
Fallo de página	Síncrono	Forzada	No enmascarable	Entre	Reanudar
Accesos a memoria mal alineados	Síncrono	Forzada	Enmascarable por el usuario	Entre	Terminar
Violaciones de protección de memoria	Síncrono	Forzada	No enmascarable	Entre	Terminar
Utilizar instrucciones indefinidas	Síncrono	Forzada	No enmascarable	Entre	Terminar
Malfunciones hardware	Asíncrono	Forzada	No enmascarable	Entre	Terminar
Fallo de alimentación	Asíncrono	Forzada	No enmascarable	Entre	Terminar

FIGURA 5.11 Los eventos de la Figura 5.9 clasificados utilizando cinco categorías.

La Figura 5.11 clasifica los ejemplos de la Figura 5.9 de acuerdo con estas cinco categorías.

Cómo el control comprueba las interrupciones

Integrar interrupciones con control significa modificar el diagrama de estados finitos para examinar las interrupciones. Las interrupciones que se presentan entre instrucciones son examinadas o al comienzo del diagrama de estados finitos —antes de que se decodifique una instrucción— o al final —después de que se termina la ejecución de una instrucción. Las interrupciones que se pueden presentar en una instrucción se detectan generalmente en el estado que causa la acción o en el estado que la sigue. Por ejemplo, la Figura 5.12 muestra la Figura 5.4 modificada para comprobar interrupciones.

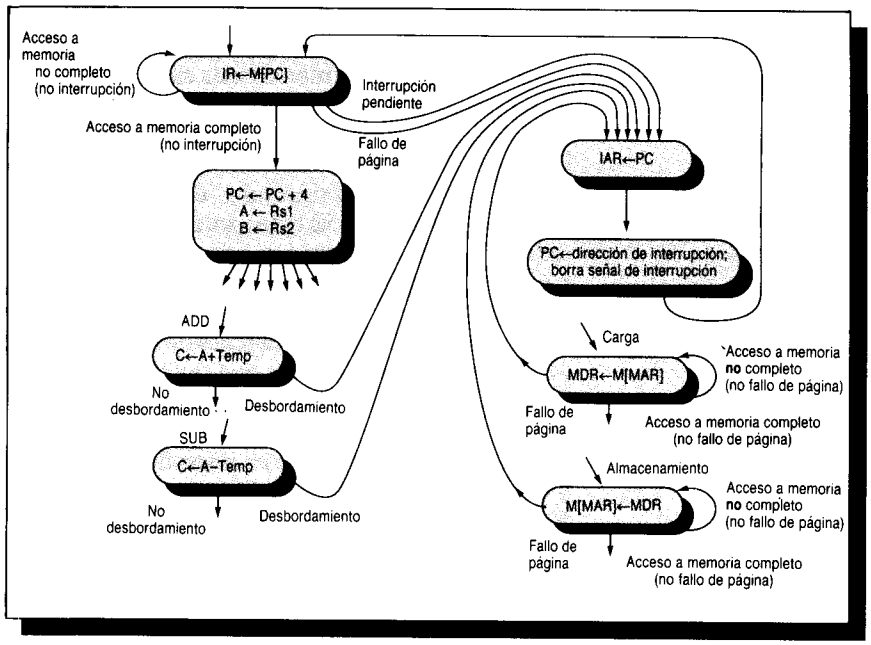


FIGURA 5.12 Vista del nivel superior del diagrama de estados finitos de DLX (Fig. 5.4 en la pág. 221) modificado para comprobar interrupciones. Bien una «interrupción entre» o un fallo de página de instrucción invoca el control que guarda el PC y después lo carga con la dirección de la rutina de interrupción apropiada. La parte inferior de la figura muestra las interrupciones resultantes de los fallos de página de los accesos a datos o desbordamiento aritmético.

Suponemos que DLX transfiere la dirección de vuelta a un nuevo registro visible para el programador; el registro de dirección de retorno de interrupciones. El control carga entonces el PC con la dirección de la rutina de interrupción para esa interrupción.

Qué es difícil de las interrupciones

La terminología conflictiva es confusa, pero ésa no es la que hace difícil la parte del hardware de control. Aun cuando las interrupciones sean poco frecuentes, el hardware debe diseñarse para que se pueda guardar el estado completo de la máquina, incluyendo una indicación del evento causante, y el PC de la instrucción que se va a ejecutar después de servir la interrupción. Esta dificultad está exacerbada por los eventos que ocurren a mitad de la ejecución, muchas instrucciones también requieren que el hardware restaure la máquina al estado inmediatamente anterior al que ocurrió el evento —al principio de la instrucción. Este último requerimiento es tan difícil que los computadores tienen el título de *recomenzable* si pasan esta prueba. El hecho de que supercomputadores y muchos microprocesadores primitivos no logren

esa insignia de honor, ilustra la dificultad de las interrupciones y su coste potencial en cuanto a complejidad hardware y velocidad de ejecución.

Ningún ingeniero merece más admiración que los que construyeron el primer VAX, el primer minicomputador recomenzable de DEC. Las instrucciones de longitud variable significan que el computador puede buscar 50 bytes de una instrucción antes de descubrir que el siguiente byte de la instrucción no está en memoria principal —una situación que requiere tener el PC guardado para apuntar a 50 bytes anteriores—. ¡Imaginar las dificultades de recomenzar una instrucción con seis operandos, cada uno de los cuales puede estar mal alineado y, por tanto, estar parcialmente en memoria y parcialmente en el disco!

Las instrucciones más difíciles de recomenzar son las que modifican algún estado de la máquina antes que se conozca si se va a presentar una interrupción. Los modos de direccionamiento de autoincremento y autodecremento del VAX podían modificar naturalmente registros durante la fase de ejecución del direccionamiento en lugar de en la fase de postescritura *(writeback)*, y así era vulnerable a esta dificultad. Para evitar este problema, los últimos VAX mantienen una cola de la historia de los especificadores de registros y de las operaciones sobre los registros, para que éstas se puedan invertir en una interrupción. Otro enfoque, utilizado en los primeros VAX, era grabar los especificadores y los valores originales de los registros para restaurarlos en caso de interrupción. (La diferencia principal es que solamente se necesitan muy pocos bits para registrar cómo se cambió la dirección, debido al autoincremento o autodecremento en contraposición con el valor completo del registro de 32 bits.)

No son sólo los modos de direccionamiento los que hacen al VAX difícil de recomenzar; las instrucciones de larga ejecución significan que las interrupciones se deben comprobar en mitad de su ejecución para prevenir largas latencias de interrupción. MOVC3, por ejemplo, copia hasta 2^{16} bytes y puede necesitar decenas de milisegundos en terminar la copia —demasiado tiempo de espera para un evento urgente. Por otra parte, aunque hubiese una forma de deshacer la copia en la mitad de la ejecución para que MOVC3 pudiese recomenzar, las interrupciones se presentarían con tanta frecuencia, con respecto a esta instrucción de larga ejecución (ver Fig. 5.10), que MOVC3 arrancaría repetidamente bajo aquellas condiciones. Este esfuerzo desperdiciado de copias incompletas convertiría a MOVC en algo menos que inútil.

DEC dividió el problema para resolverlo. Primero, se extraen de memoria los operandos —dirección fuente, longitud y dirección destino— y se colocan en los registros de propósito general R1, R2 y R3. Si se presenta una interrupción durante esta primera fase, se restauran estos registros, y MOVC3 se recomienza desde el principio. Después de esta primera fase, cada vez que se copia un byte, se decrementa la longitud (R2) y se incrementan las direcciones (R1 y R3). Si se presenta una interrupción durante esta segunda fase, MOVC3 inicializa el bit de *primera parte hecha* (FPD) en la palabra de estado del programa. Cuando la interrupción es atendida y se reejecuta la instrucción, primero comprueba el bit FPD para ver si los operandos ya han sido colocados en los registros. Si es así, la VAX no extrae direcciones ni longitudes de operandos, sino que continúa con los valores actuales de los registros, ya que es todo lo que queda por copiar. Esto permite una respuesta más rápida a las

interrupciones mientras deja que las instrucciones de larga ejecución hagan progresos entre interrupciones.

IBM tenía un problema similar. El 360 incluía la instrucción MVC, que copia hasta 256 bytes de datos. Para las primeras máquinas sin memoria virtual, la máquina esperaba hasta que se completaba la instrucción antes de atender las interrupciones. Con la inclusión de la memoria virtual en el 370, el problema no se podía ignorar más tiempo. Primero, el control trata de acceder a todas las posibles páginas, forzando que todas las posibles interrupciones de fallos de memoria virtual se presenten antes de transferir cualquier dato. Si en esta fase se presenten alguna interrupción, la instrucción era recomenzada. El control ignora entonces las interrupciones hasta que se completa la instrucción. Para permitir copias mayores, el 370 incluye MOVCL, que puede transferir hasta 2^{24} bytes. Los operandos están en registros y son actualizados como parte de la ejecución —como en el VAX, excepto que no hay necesidad de FPD ya que los operandos están siempre en registros. (O, para hablar históricamente, la solución VAX es como el IBM 370, que llegó primero.)

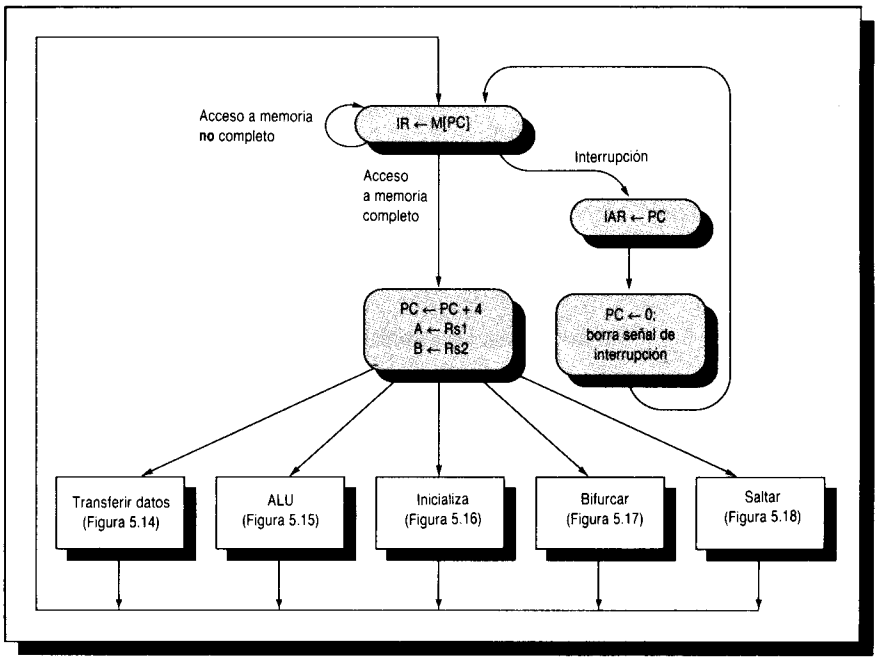


FIGURA 5.13 Vista del nivel superior del diagrama de estados finitos de DLX para instrucciones distintas de punto flotante. Se muestran los dos primeros pasos de la ejecución de la instrucción —búsqueda de la instrucción y de decodificación/búsqueda de datos—. El primer estado se repite hasta que la instrucción es obtenida de memoria o se detecta una interrupción. Si se detecta una interrupción, el PC se guarda en el IAR y el PC se inicializa con la dirección de la rutina de interrupción. Los últimos tres pasos de la ejecución de la instrucción —ejecución/dirección efectiva, acceso a memoria y postescritura— se muestran en las Figuras 5.14 a 5.18.

5.7 Juntando todo: control para DLX

El control para DLX se presenta aquí para enlazar las ideas de las tres secciones anteriores. Comenzamos con un diagrama de estados finitos para representar el control cableado y terminamos con control microprogramado. Se utilizan ambas versiones de control de DLX para demostrar los compromisos para reducir coste o mejorar rendimiento. Debido a que las figuras son demasiado grandes, la comprobación de fallos *(faults)* de página de datos o de desbordamiento aritmético mostrados en la Figura 5.12 no se incluyen en esta sección. (El ejercicio 5.12 los añade.)

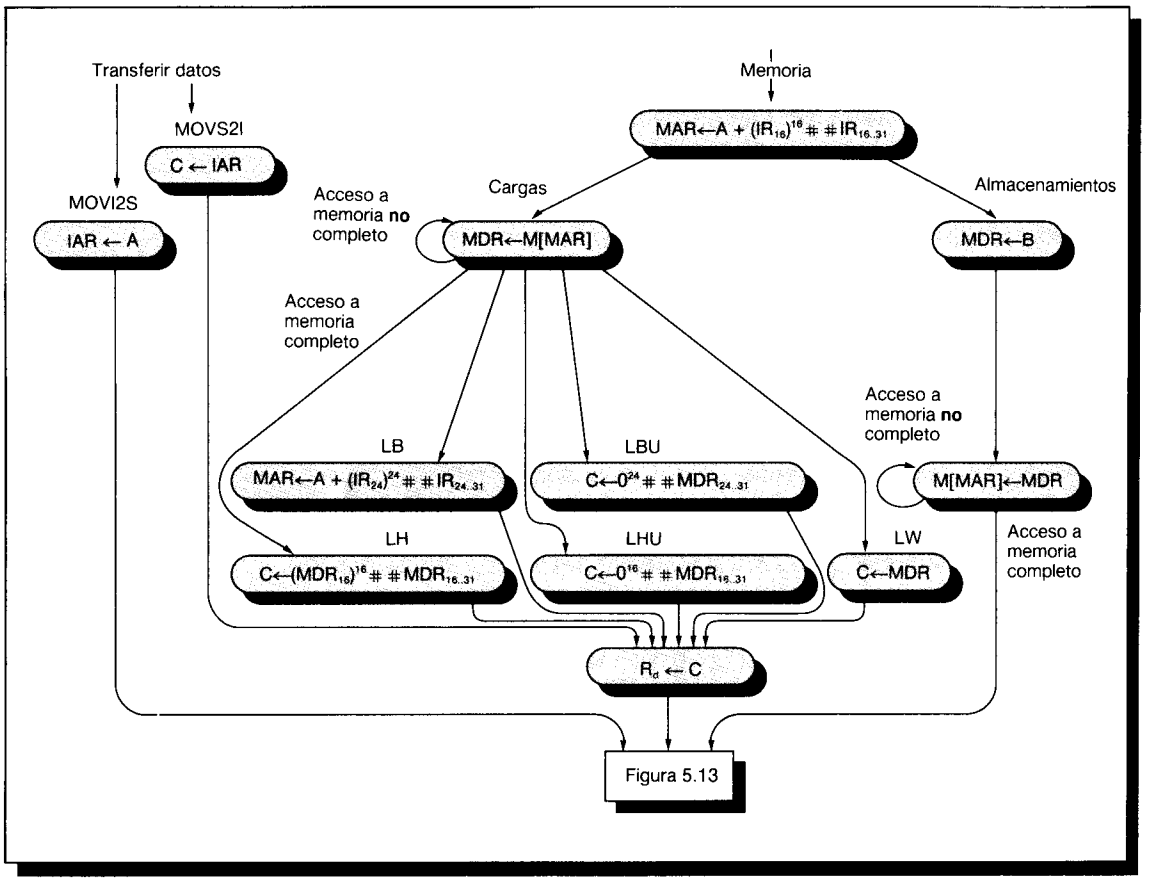


FIGURA 5.14 Estados de cálculo de dirección efectiva, acceso a memoria y postescritura para las instrucciones de acceso a memoria y transferencia de datos de DLX. Para las cargas, el segundo estado se repite hasta que el dato es obtenido de memoria. El estado final de los almacenamientos se repite hasta que se completa la escritura. Mientras la operación de las cinco cargas se muestra en los estados de esta figura, la operación de escritura adecuada depende de los bytes y medias palabras escritas en el sistema de memoria, sin perturbar el resto de la palabra de memoria, y alinear correctamente los bytes y medias palabras (ver Fig. 3.10) sobre los bytes adecuados de memoria. El control de ejecución se completa en la Figura 5.13.

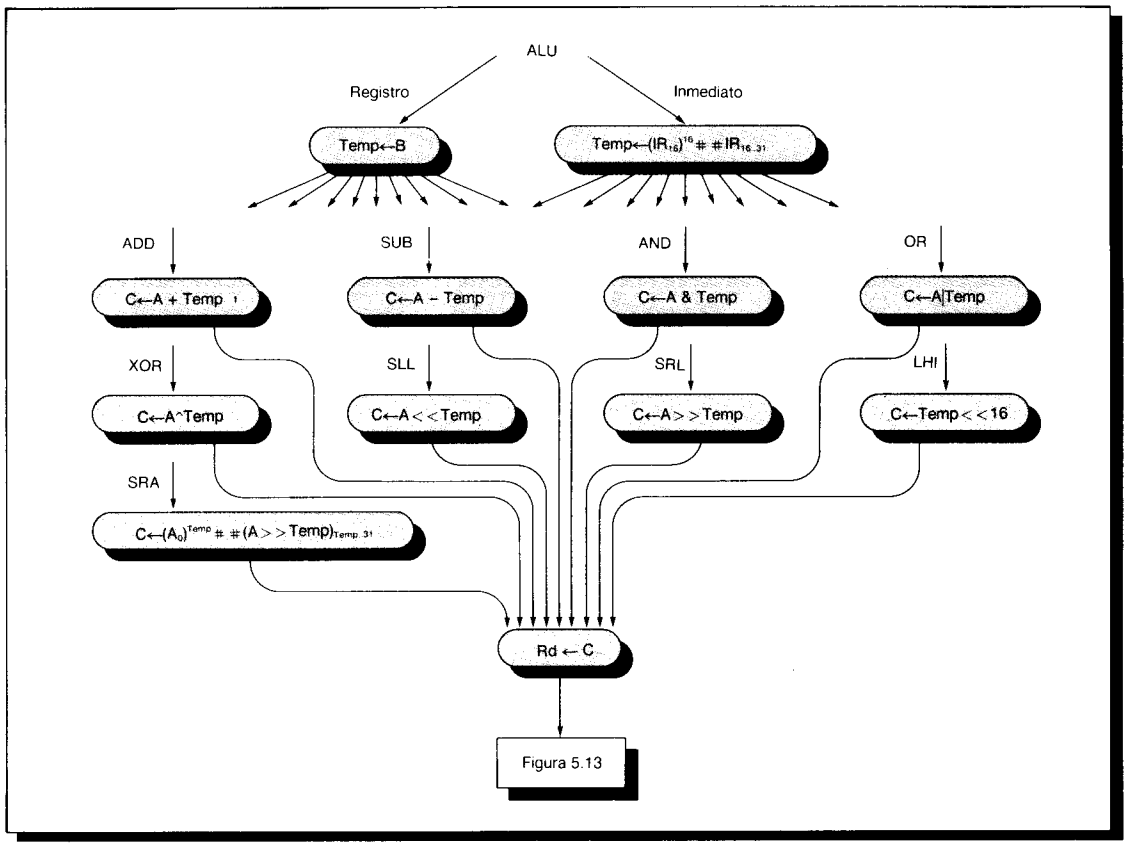


FIGURA 5.15 Los estados de ejecución y postescritura para las instrucciones de la ALU de DLX. Después de poner un registro o el inmediato de 16 bits de signo extendido en Temp, se ejecuta una de las nueve instrucciones, y el resultado (C) se postescribe en el fichero de registros. Sólo SRA y LHI pueden no ser autoexplicativas. La instrucción SRA desplaza a la derecha mientras extiende el signo al operando y LHI carga los 16 bits superiores del registro mientras pone a cero los 16 bits inferiores. (Los operadores de C «<<» y «>>» desplazan a la izquierda y derecha, respectivamente; se rellenan con ceros a menos que los bits estén concatenados explícitamente utilizando # #, p. e., extensión de signo.) Como mencionamos antes, la comprobación del desbordamiento en ADD y SUB no se incluye para simplificar la figura. El control de la ejecución se completa en la Figura 5.13.

FIGURA 5.16 (ver página adjunta). Los estados de ejecución y postescritura para las instrucciones de inicialización (Set) de DLX. Después de poner un registro o el inmediato de 16 bits de signo extendido en Temp, 1 de las 6 instrucciones compara A con Temp y después pone C a 1 o 0, dependiendo que la condición sea cierta o falsa. C se postescribe entonces en el fichero de registros, y después el control de la ejecución se transfiere a la Figura 5.13. Las líneas a trazos de esta figura y de la Figura 5.18 se utilizan para seguir de manera más fácil las líneas de intersección.

FIGURA 5.17 (ver página adjunta). Los estados de ejecución y postescritura para las instrucciones de bifurcación de DLX. Para las instrucciones de bifurcación y enlace, la dirección retorno se coloca primero en C antes de que el nuevo valor se cargue en el PC. Trap la salva en el IAR. Observar que el inmediato de estas instrucciones es 10 bits mayor que el inmediato de 16 bits de las demás instrucciones. Las instrucciones de bifurcación y enlace terminan al escribir la dirección de retorno en R31. Para completar la ejecución, el control se transfiere a la Figura 5.13.

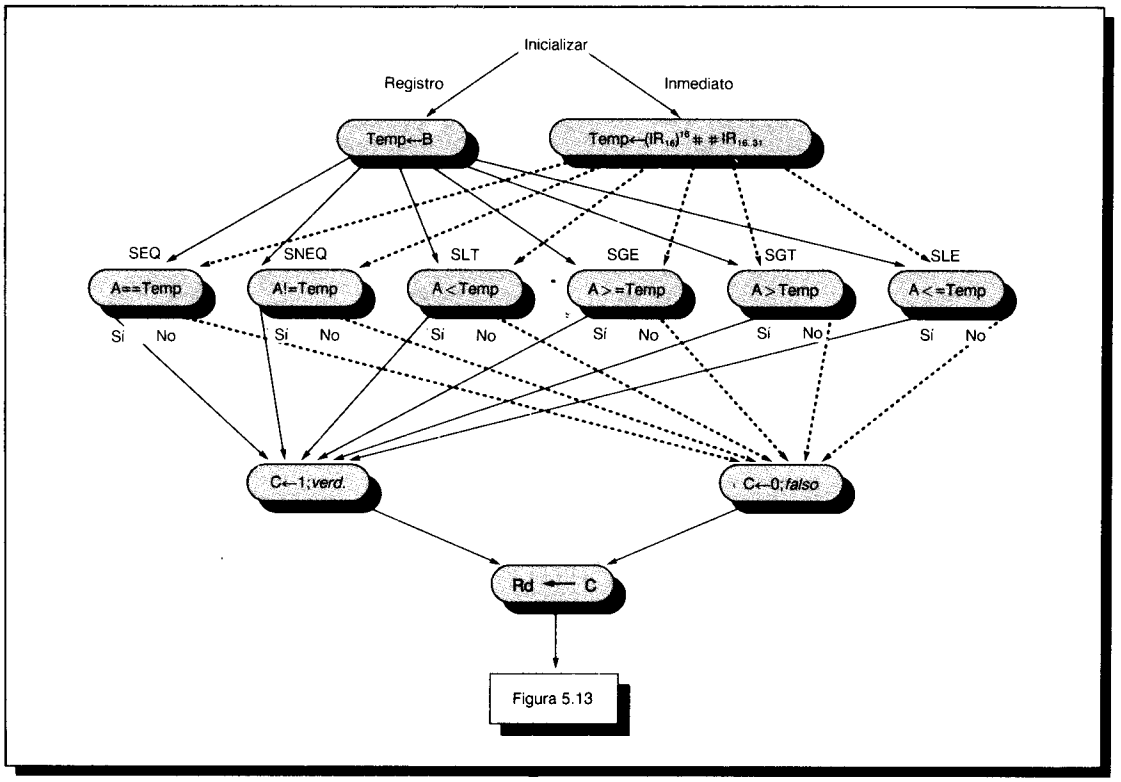


FIGURA 5.16

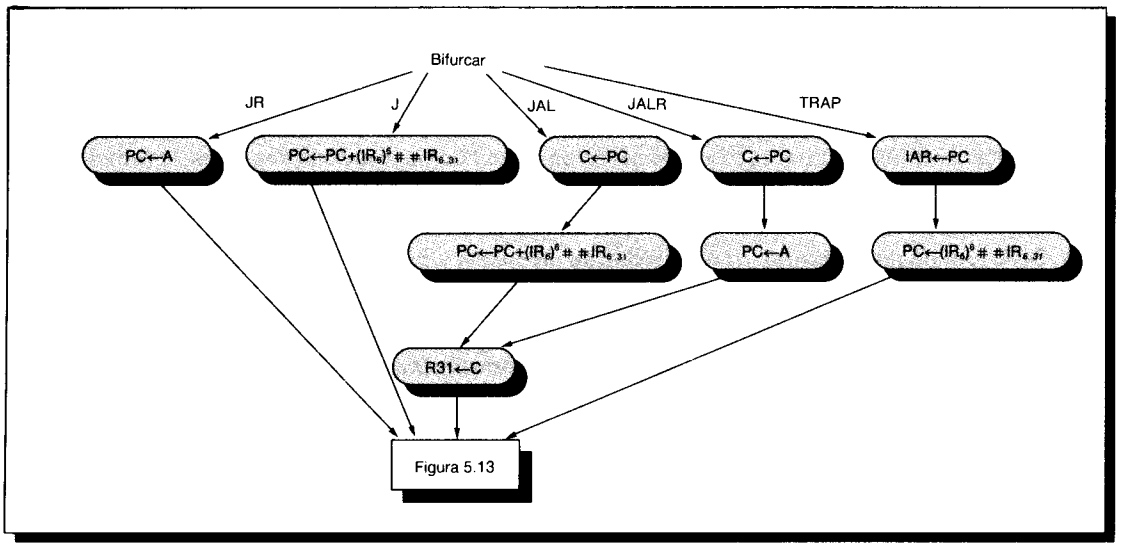


FIGURA 5.17

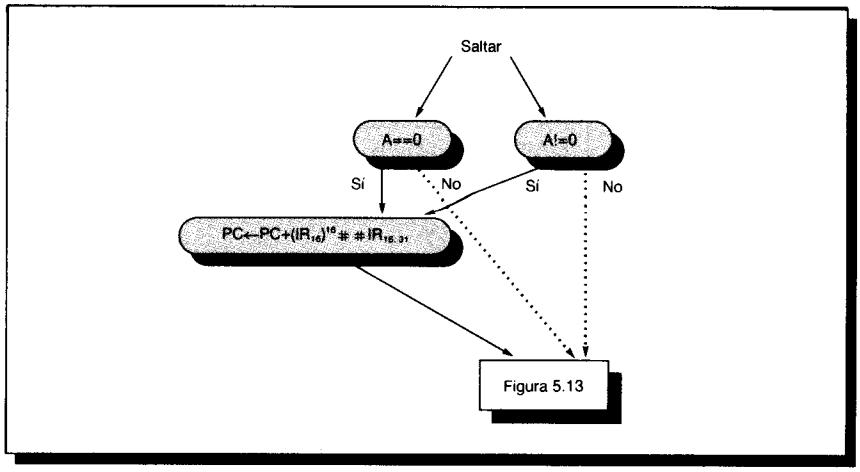


FIGURA 5.18 Los estados de ejecución para las instrucciones de salto de DLX. El PC se carga con la suma del PC y el inmediato sólo si la condición es verdadera. Al completar la ejecución, el control se transfiere a la Figura 5.13.

En lugar de intentar dibujar la máquina de estados finitos DLX en una sola figura mostrando los 52 estados, la Figura 5.13 muestra la parte superior, que contiene 4 estados, más referencias a los restantes estados que se detallan en las Figuras 5.14 (debajo) a 5.18. De forma distinta a la Figura 5.2, la Figura 5.13 aprovecha la modificación del camino de datos que permite al PC direccionar directamente memoria sin ir a través del MAR (Fig. 5.4).

Rendimiento del control cableado para DLX

Como se planteó en la Sección 5.4, el objetivo en los diseños de control es minimizar el CPI, la duración del ciclo de reloj, la cantidad de hardware de control, y el tiempo de desarrollo. El CPI es exactamente el número medio de estados a lo largo del camino de ejecución de una instrucción.

Ejemplo

Supongamos que el control cableado implementa directamente el diagrama de estados finitos de las Figuras 5.13 a 5.18. ¿Cuál es el CPI para DLX ejecutando GCC?

Respuesta

El número de ciclos de reloj para ejecutar cada instrucción DLX se determina, sencillamente, contando los estados de una instrucción. Empezando en la parte superior, cada instrucción emplea como mínimo dos ciclos de reloj en los estados de la Figura 5.13 (ignorando las interrupciones). El número real depende de la media del número de veces que deba repetirse el estado que accede a memoria porque la memoria no esté lista. (Estos ciclos de reloj desperdiciados habitualmente se denominan *ciclos de detención de memoria* o *estados de espera* —memory stall cycles o wait states—.) En las máquinas basadas

Instrucciones DLX	Ciclos mínimos de reloj	Accesos a memoria	Total ciclos de reloj
Cargas	6	2	8
Almacenamientos	5	2	7
ALU	5	1	6
Inicialización	6	1	7
Bifurcaciones	3	1	4
Bifurcación y enlaces	5	1	6
Salto (efectivo)	4	1	5
Salto (no efectivo)	3	1	4

FIGURA 5.19 Ciclos de reloj por instrucción para categorías de DLX que usan el diagrama de estados de las Figuras 5.13 a 5.18. Determinar los ciclos totales de reloj por categoría requiere multiplicar el número de accesos a memoria —incluyendo búsquedas de instrucciones— por el número medio de estados de espera, y sumando este producto al mínimo número de ciclos de reloj. Suponemos un promedio de 1 ciclo de reloj por acceso a memoria. Por ejemplo, las cargas emplean ocho ciclos de reloj si el número medio de estados de espera es uno.

en caches, este valor normalmente es 0 (p. e., no repeticiones, ya que cada acceso de cache tarda un ciclo) cuando el dato se encuentra en la cache, y como mínimo 10 cuando no se encuentra.

El tiempo para la parte restante de la ejecución de la instrucción proviene de las figuras adicionales. Además de los dos ciclos para la búsqueda y decodificación, las instrucciones de carga tardan cuatro ciclos más, más los ciclos de reloj esperando el acceso a los datos, mientras que las instrucciones de almacenamiento tardan tres ciclos de reloj más, más los estados de espera. Las instrucciones de la ALU necesitan tres ciclos de reloj extra, y las instrucciones de inicialización (set) cuatro. La Figura 5.17 muestra que las bifurcaciones necesitan un ciclo de reloj extra mientras que las instrucciones de bifurcación y enlace tres. Los saltos dependen del resultado: los saltos efectivos utilizan dos ciclos de reloj más, mientras los no efectivos necesitan solamente uno. Sumando estos tiempos a la primera parte de la ejecución de la instrucción se obtienen los ciclos de reloj por clase de instrucción DLX mostrados en la Figura 5.19.

Del Capítulo 2, una forma de calcular el CPI es

$$\text{CPI} = \sum_{i=1}^{n} \left(\text{CPI}_i \cdot \frac{\text{I}_i}{\text{Recuento de instrucciones}} \right)$$

Utilizando la mezcla de instrucciones de DLX de la Figura C.4 en el Apéndice C para GCC (normalizada a 100 por 100), el porcentaje de los saltos efectivos a partir de la Figura 3.23, y el número medio de estados de espera

por acceso a memoria se calcula el CPI de DLX para este camino de datos y diagrama de estados.

Cargas	8 · 21 %	=	1,68
Almacenamientos	7 · 12 %	=	0,84
ALU	6 · 37 %	=	2,22
Inicialización	7 · 6 %	=	0,42
Bifurcaciones	4 · 2 %	=	0,08
Bifurcación y enlaces	6 · 0 %	=	0,00
Salto (efectivo)	5 · 12 %	=	0,60
Salto (no efectivo)	4 · 11 %	=	0,44
	CPI Total:		6,28

Por tanto, el CPI de DLX para GCC es aproximadamente 6.3.

Mejorar el rendimiento de DLX cuando el control es cableado

Como mencionamos antes, el rendimiento mejora al reducir el número de estados por los que la instrucción debe pasar durante su ejecución. A veces, el rendimiento se puede mejorar eliminando cálculos intermedios que seleccionan una de varias opciones, bien añadiendo hardware que utiliza la información del código de operación para seleccionar más tarde la opción apropiada, o incrementando sencillamente el número de estados.

Ejemplo

Veamos una mejora del rendimiento de las instrucciones de la ALU eliminando los dos estados superiores de la Figura 5.15, que cargan un registro o un inmediato en Temp. Un enfoque utiliza una nueva opción hardware, que llamaremos «X» (ver Fig. 5.20). La opción X selecciona o el registro B o el inmediato de 16 bits, dependiendo del código de operación en IR. Una segunda aproximación consiste en incrementar simplemente el número de estados de ejecución, para que haya estados separados para instrucciones de la ALU que usan inmediatos frente a instrucciones de la ALU que utilizan registros.

Para cada opción, ¿cuál sería el cambio en el rendimiento y cómo sería el diagrama de estados modificado? Además, ¿cuántos estados se necesitarían en cada opción?

Respuesta

Ambos cambios reducen el tiempo de ejecución de la ALU de cinco a cuatro ciclos de reloj más los estados de espera. De la Figura C.4 las operaciones de la ALU son, aproximadamente, el 37 por 100 de las instrucciones para GCC, bajando el CPI de 6,3 a 5,9, y haciendo la máquina aproximadamente el 7 por

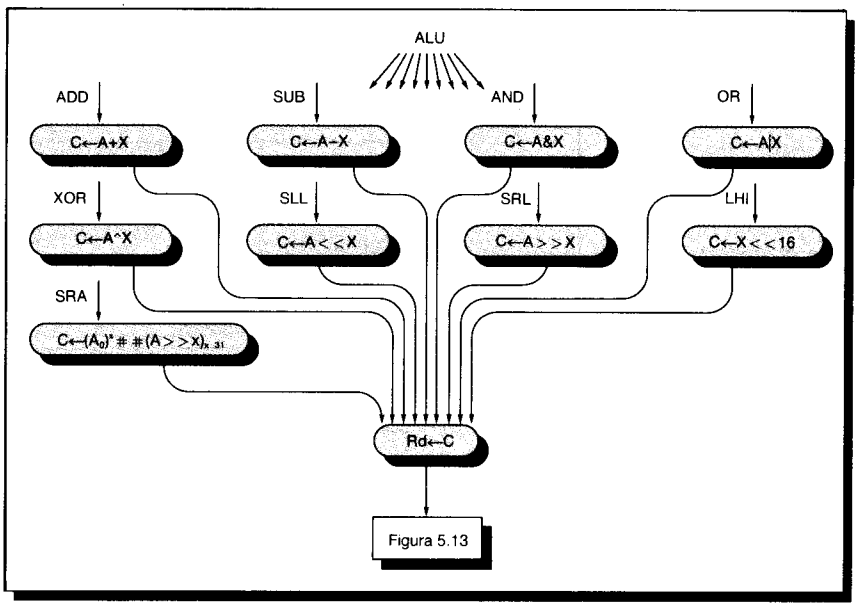


FIGURA 5.20 Figura 5.15 modificada para eliminar los dos estados que cargan Temp. Los estados utilizan la nueva opción X para significar que B o $(IR_{16})^{16}$ # # $IR_{16..31}$ es el operando, dependiendo del código de operación de DLX.

100 más rápida. La Figura 5.20 muestra la Figura 5.15 modificada para utilizar la opción X en lugar de los dos estados que cargan Temp, mientras que la Figura 5.21 sencillamente tiene muchos más estados para conseguir el mismo resultado. El número total de estados son 50 y 58, respectivamente.

El control puede afectar a la duración del ciclo de reloj, bien porque el mismo control emplee más tiempo que las operaciones correspondientes del camino de datos, o porque las operaciones del camino de datos, seleccionadas por el control, alarguen la duración del ciclo de reloj en el peor caso.

Ejemplo

Suponer una máquina con un ciclo de reloj de 10 ns (frecuencia de reloj 100 MHz). Suponer que en una inspección más profunda el diseñador descubre que todos los estados podían ejecutarse en 9 ns, excepto los estados que utilizan el desplazador. ¿Sería inteligente separar aquellos estados, tomando dos ciclos de reloj de 9 ns para los estados de desplazamiento y un ciclo de reloj de 9 ns para los demás?

Respuesta

Suponiendo la mejora del ejemplo anterior, el tiempo de ejecución medio por instrucción para la máquina de 100 MHz es 5,9 · 10 ns o sea 59 ns. El desplazador se utiliza solamente en los estados de cuatro instrucciones: `SSL`, `SRL`, `SRA`, y `LHI` (ver Fig. 5.20). En efecto, cada una de estas instrucciones necesita 6 ciclos de reloj (incluyendo un estado de espera para el acceso a memoria), y

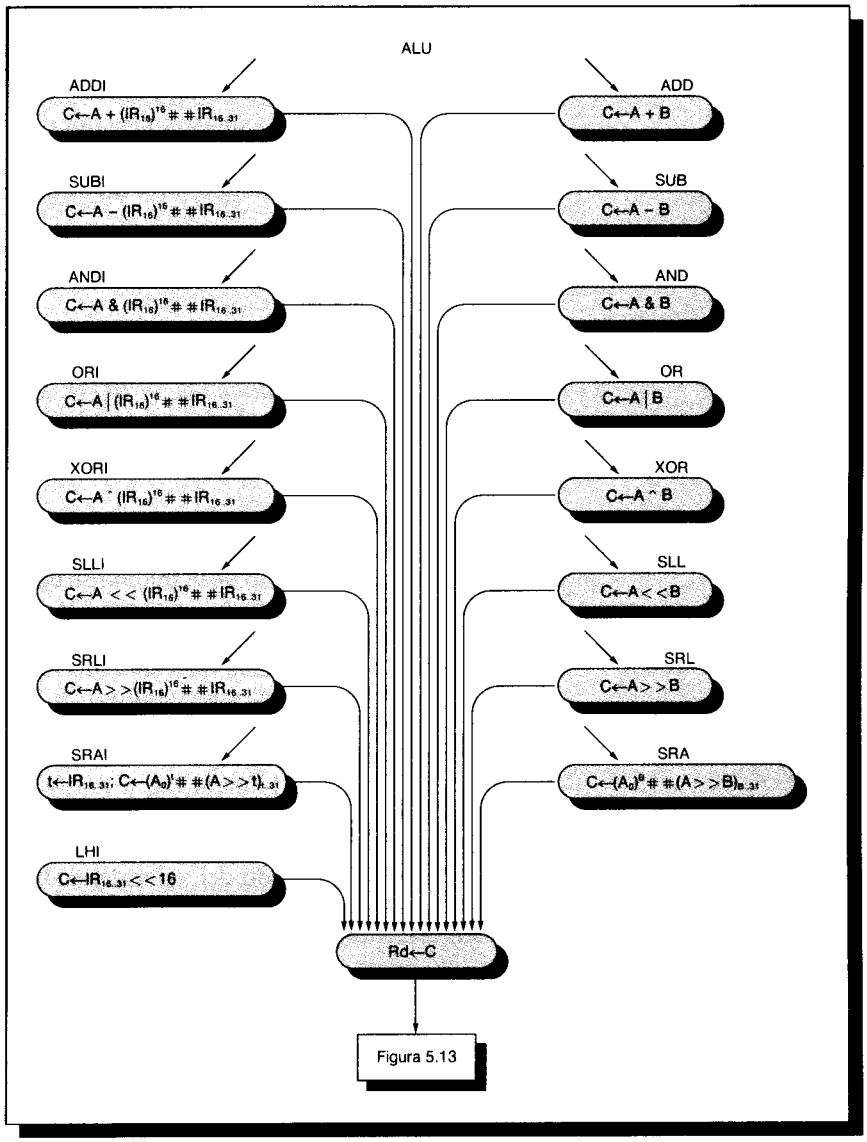


FIGURA 5.21 Figura 5.15 modificada para eliminar los dos estados de carga Temp. De forma distinta a la Figura 5.20, esto no requiere nuevas opciones hardware en el camino de datos, sino simplemente más estados de control.

solamente es necesario dividir uno de los seis ciclos de reloj. Por tanto, el tiempo medio de ejecución de estas instrucciones cambia de 6 · 10 ns, o 60 ns, a 7 · 9 ns, o 63 ns. En la Figura C.4 vemos que estas 4 instrucciones son aproximadamente el 8 por 100 de las instrucciones ejecutadas para GCC (después de la normalización), haciendo que el tiempo de ejecución medio por instrucción sea 92 por 100 · (5,9 · 9 ns) + 8 por 100 · 63 ns, es decir 54 ns. Por tanto,

dividiendo el estado de desplazamiento resulta una máquina que es aproximadamente el 10 por 100 más rápida —una decisión inteligente. (Ver Ejercicio 5.8 para una versión más sofisticada de este ejemplo.)

El control cableado se completa listando las señales de control activadas en cada estado, asignando números a los estados, y, finalmente, generando el PLA. Ahora implementemos el control utilizando microcódigos en una ROM.

Control microcodificado para DLX

Un formato a medida como éste es un esclavo para la arquitectura del hardware y del nivel lenguaje máquina al que sirve. El formato debe lograr un compromiso adecuado entre tamaño de la ROM, tamaño de la circuitería de decodificación de salida de la ROM, y velocidad de ejecución de la máquina.

Jim McKevit y cols. [1977]

Antes de que pueda comenzar la microprogramación, debe determinarse el repertorio de microinstrucciones. El primer paso es listar las posibles entradas para cada campo del formato de la microinstrucción de DLX de la Figura 5.6. La Figura 5.7 los lista para los campos Destino, Fuente1 y Fuente2. La Figura 5.22, más adelante, muestra los valores para los restantes campos.

El secuenciamiento de las microinstrucciones requiere una explicación adicional. El control microprogramado incluye un contador de microprograma para especificar la dirección de la siguiente microinstrucción si no se realiza un salto, como en la Figura 5.5. Además de los saltos que utilizan el campo de la dirección de bifurcación, se emplean tres tablas para decodificar las macroinstrucciones de DLX. Estas tablas están indexadas con los códigos de operación de las instrucciones de DLX, y proporcionan una dirección del microprograma dependiendo del valor del código de operación. Su uso será más claro cuando examinemos el microprograma de DLX.

Siguiendo la indicación del diagrama de estados, el microprograma de DLX está dividido en las Figuras 5.23, 5.25, 5.27, 5.28 y 5.29, con cada sección de microcódigo correspondiendo a una de las Figuras 5.13 a 5.18. El primer estado de la Figura 5.13 se convierte en las dos primera microinstrucciones de la Figura 5.23. La primera microinstrucción (dirección 0) salta a la microinstrucción 3 si hay una interrupción pendiente. La microinstrucción 1 busca una instrucción de memoria, saltando hacia atrás hasta que no se completa el acceso a memoria. La microinstrucción 2 incrementa en 4 el PC, carga A y B, y después realiza la decodificación de primer nivel. La dirección de la siguiente microinstrucción depende entonces de la macroinstrucción que esté en el registro de instrucción. Las direcciones de las microinstrucciones para este primer nivel de decodificación de la macroinstrucción se especifican en la Figura 5.24. (En realidad, la tabla mostrada en esta figura se especifica después de escribir el microprograma, ya que el número de entradas y las correspondientes posiciones no se conocen hasta entonces.)

La Figura 5.25 contiene las instrucciones de almacenamiento y carga de DLX. La microinstrucción 5 calcula la dirección efectiva, y salta a la microinstrucción 9 si la macroinstrucción del IR es una carga. Si no,

Valor	ALU	Misc		Cond	
0	ADD +	Lee Instr	*IR←M[PC]*	---	*Ir a la siguiente microinstrucción secuencial*
1	SUB −	Lee Dato	*MDR←M[MAR]*	Incond	*Bifurca siempre*
2	RSUB $-_r$ (Resta inversa)	Escribe	*M[MAR]←MDR*	¿Int?	*¿Interrupción pendiente* (entre instrucciones)?
3	AND /	AB←RF	*Carga A&B desde Fichero de Reg*	¿Mem?	*¿Acceso a memoria no completo?*
4	OR ^	Rd←C	*Escribe Rd*	¿Cero?	*¿Es cero la salida de la ALU?*
5	XOR <<	R31←C	*Escribe R31 (para Call)*	¿Negativo?	*¿Es menor que cero la salida de ALU?*
6	SLL >>			¿Carga?	*¿Es la macroinstrucción una carga de DLX?*
7	SRL $>>_a$			Decod1 (Fig. 5.24)	*La tabla de direcciones 1 determina la siguiente microinstrucción (utiliza código de operación principal)*
8	SRA *S1*			Decod2 (Fig. 5.26)	*La tabla de direcciones 2 determina la siguiente microinstrucción (utiliza código de operación «func»)*
9	Paso S1 *S2*			Decod3 (Fig. 5.26)	*La tabla de direcciones 3 determina la siguiente microinstrucción (utiliza código de operación principal)*
10	Paso S2 *S2*				

FIGURA 5.22 Las opciones para los tres campos del formato de microinstrucciones de DLX de la Figura 5.6. Los posibles nombres se muestran a la izquierda del nombre del campo, con una explicación de cada campo a la derecha. La microinstrucción real podría contener un patrón de bits correspondiente al número de la primera columna. Combinada con la Figura 5.7, todos los campos están definidos excepto los campos de direcciónes Constantes y Bifurcación, que contienen números proporcionados por el microprogramador. $>>_a$ es una abreviatura para los desplazamientos aritméticos a la derecha y $-_r$ significa resta inversa (B $-_r$A = A − B).

la microinstrucción 6 carga el MDR con el valor que se va a almacenar, y la microinstrucción 7 bifurca a sí misma hasta que la memoria haya terminado de escribir el dato. La microinstrucción 8 entonces bifurca, hacia atrás, a la microinstrucción 0 (Fig. 5.23) para comenzar otra vez el ciclo de ejecución. Si la macroinstrucción era una carga, la microinstrucción 9 se repite hasta que se haya leído el dato. La microinstrucción 10 utiliza entonces la tabla 2 de decodificación (especificada en la Fig. 5.26) para especificar la dirección de la siguiente microinstrucción. De forma distinta a la primera tabla de decodificación, esta tabla es utilizada por otras microinstrucciones. (No hay conflicto de múltiples usos, ya que los códigos de operación de cada instancia son diferentes.)

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
0	Ifetch:							¿Interrupc?	Intrpt	*Comprueba interrupción*
1	Iloop:						Lee Instr	¿Mem?	Iloop	*IR←M[PC]; espera memoria*
2		PC	ADD	PC	Constante	4	AB←RF (Fich. de Reg.)	Decod1		
3	Intrpt:	IAR	Pasa S1	PC						*Interrupción*
4		PC	Pasa S2		Constante	0		Incond	Ifetch	*PC←0 & ir a buscar siguiente instrucción*

FIGURA 5.23 La primera sección del microprograma DLX correspondiente a los estados de la Figura 5.13. La primera columna contiene la dirección absoluta de microinstrucción, seguida por una etiqueta. El resto de los campos contiene valores de las Figuras 5.7 y 5.22 para el formato de microinstrucción de la Figura 5.6. Como ejemplo, la microinstrucción 2 corresponde al segundo estado de la Figura 5.13. Envía la salida de la ALU al PC, dice a la ALU que sume, pone el PC en el bus Fuente 1, y una constante de la microinstrucción (cuyo valor es 4) en el bus Fuente 2. Ademas, A y B se cargan desde el fichero de registros de acuerdo con los especificadores del IR. Finalmente, la dirección de la siguiente microinstrucción que se va a ejecutar proviene de la tabla 1 de códigos (Figura 5.24), que depende del código de operación del registro de instrucción (IR).

Suponer que la instrucción fuese una carga de media palabra. La Figura 5.26 muestra que el resultado de la decodificación 2 sería una bifurcación a la microinstrucción 15. Esta microinstrucción desplaza el contenido del MDR 16 bits a la izquierda y almacena el resultado en Temp. La siguiente microinstrucción desplaza aritméticamente a la derecha Temp 16 bits y coloca el resultado en C. C contiene ahora los 16 bits de más a la derecha del MDR, mientras que los 16 bits superiores contienen la extensión de signo. Esta microinstrucción bifurca a la posición 22, que vuelve a escribir en el especificador de registros destino en IR, y entonces bifurca para buscar la macroinstrucción siguiente comenzando en la posición 0 (Figura 5.23).

Las instrucciones de la ALU se encuentran en la Figura 5.27. Las dos primeras microinstrucciones corresponden a los estados de la parte superior de la Figura 5.15. Después de cargar Temp con el registro o con el inmediato, cada uno utiliza una tabla de decodificación para saltar a la microinstrucción que ejecuta la instrucción de la ALU. Para ahorrar espacio de microcódigo, se utiliza la misma microinstrucción si el operando es un registro o inmediato. Se ejecuta una de las microinstrucciones entre la 25 y 33, almacenando el resultado en C. Entonces bifurca a la microinstrucción 34, que almacena C en el registro especificado en el IR, y vuelve a bifurcar para buscar la siguiente macroinstrucción.

La Figura 5.28 corresponde a los estados de la Figura 5.16, excepto que los dos estados superiores que cargan Temp son las microinstrucciones 23 y 24 de la figura anterior; las tablas de decodificación bifurcarán a las posiciones

25 a 34 de la Figura 5.27, o a las 35 a 45 de la Figura 5.28, dependiendo del código de operación. Las microinstrucciones para Set realizan tests relativos haciendo que la ALU que reste Temp de A y después examine la salida de la ALU para ver si el resultado es negativo o cero. Dependiendo del resultado del test, C se pone a 1 ó a 0 y se escribe de nuevo *(write back)* en el fichero de registros antes de buscar la siguiente macroinstrucción. Los tests para A = Temp, A ≠ Temp, A < Temp y A ≥ Temp son correctos utilizando estas condiciones en la salida de la ALU A − Temp. Por otra parte, A > Temp y A ≤ Temp, no son simples, pero pueden hacerse utilizando la condición opuesta con la resta invertida:

$$(\text{Temp} - \text{A} < 0) = (\text{Temp} < \text{A}) = (\text{A} > \text{Temp})$$

Si el resultado es negativo, entonces A > Temp, en otro caso A ≤ Temp.

La Figura 5.29 contiene el microcódigo final de DLX y corresponde a los estados encontrados en las Figuras 5.17 y 5.18. La microinstrucción 50, correspondiente a la macroinstrucción salta si igual a cero, examina si A es igual a cero. Si lo es, el salto de la macroinstrucción tiene éxito, y la microinstrucción bifurca a la microinstrucción 53. Esta microinstrucción carga el PC con

Códigos de operación (especificados simbólicamente)	Dirección absoluta	Etiqueta	Figura
Memoria	5	Mem:	5.25
Transferencia especial a	20	MovI2S:	5.25
Transferencia especial desde	21	MovS2I:	5.25
S2 = B	23	Reg:	5.27
S2 = Inmediato	24	Imm:	5.27
Salta si igual cero	50	Beq:	5.29
Salta si no igual cero	52	Bne:	5.29
Bifurcación	54	Jump:	5.29
Bifurca a registro	55	JReg:	5.29
Bifurca y enlaza	56	JAL:	5.29
Bifurca y enlaza registro	58	JALR:	5.29
Trap	60	Trap:	5.29

FIGURA 5.24 Códigos de operación y direcciones correspondientes para la tabla de decodificación (Decod1). Los códigos de operación se muestran simbólicamente a la izquierda, seguidos por las direcciones con la dirección absoluta de la microinstrucción, una etiqueta, y la figura donde se puede encontrar el microcódigo. Si esta tabla se implementase con una ROM tendría 64 entradas correspondientes al código de operación de 6 bits de DLX. Como esto produciría claramente muchas entradas redundantes o no especificadas, podría utilizarse un PLA para minimizar hardware.

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
5	Mem:	MAR	ADD	A	inm 16			Carga	Load	*Instrucc. memoria*
6	Store:	MDR	Pasa S2		B					*Almacena*
7	Dloop:						Escribe dato	¿Mem?	Dloop	
8								Incond	Ifetch	*Extrae siguiente*
9	Load:						Lee dato	¿Mem?	Load	*Carga MDR*
10								Decod2		
11	LB:	Temp	SLL	MDR	Constante	24				*Carga byte; desplaza izquierda para eliminar 24 bits superiores*
12		C	SRA	Temp	Constante	24		Incond	Writel	*Desplazamiento aritmético a la derecha para signo extendido*
13	LBU:	Temp	SLL	MDR	Constante	24				*LB sin signo*
14		C	SRL	Temp	Constante	24		Incond	Writel	*SRL lógico*
15	LH:	Temp	SLL	MDR	Constante	16				*Carga media*
16		C	SRA	Temp	Constante	16		Incond	Writel	*SRL aritmético*
17	LHU:	Temp	SLL	MDR	Constante	16				*LH sin signo*
18		C	SRL	Temp	Constante	16		Incond	Writel	*SRL lógico*
19	LW:	C	Pasa S1	MDR				Incond	Writel	*Carga palabra*
20	MovI2S:	IAR	Pasa S1	A				Incond	Ifetch	*Transfiere a especial*
21	MovS2I:	C	Pasa S1	IAR						*Transfiere desde especial*
22	Writel:						Rd←C	Incond	Ifetch	*Postescritura e ir a buscar siguiente instrucción*

FIGURA 5.25 La sección del microprograma DLX para cargas y almacenamientos, correspondiente a los estados de la Figura 5.14. El microcódigo para bytes y medias palabras necesita una microinstrucción extra para alinear los datos (ver Fig. 3.10). Notar que la microinstrucción 5 carga A desde Rd, justo en el caso que la instrucción sea un almacenamiento. La etiqueta Ifetch corresponde a la microinstrucción 0 de la Figura 5.23.

la dirección relativa al PC y después bifurca al microcódigo que busca la nueva macroinstrucción (posición 0). Si A no es igual a cero, falla el salto de la macroinstrucción, por lo que se ejecuta la siguiente microinstrucción en secuencia (51), bifurcando a la posición 0 sin cambiar el PC.

Habitualmente, un estado corresponde a una sola microinstrucción, aunque en algunos de los casos anteriores se necesitaban dos microinstrucciones. Las instrucciones de bifurcación y enlace presentan el caso inverso, con dos estados colapsados en una microinstrucción. Las acciones de los dos últimos estados de bifurcación y enlace de la Figura 5.17 se encuentran en la mi-

Código de operación	Dirección absoluta	Etiqueta	Figura
Carga byte	11	LB:	5.25
Carga byte sin signo	13	LBU:	5.25
Carga media	15	LH:	5.25
Carga media sin signo	17	LHU:	5.25
Carga palabra	19	LW:	5.25
ADD	25	ADD/I:	5.27
SUB	26	SUB/I:	5.27
AND	27	AND/I:	5.27
OR	28	OR/I:	5.27
XOR	29	XOR/I:	5.27
SLL	30	SLL/I:	5.27
SRL	31	SRL/I:	5.27
SRA	32	SRA/I:	5.27
LHI	33	LHI/I:	5.27
Inicializa si igual	35	SEQ/I:	5.28
Inicializa si no igual	37	SNE/I:	5.28
Inicializa si menor que	39	SLT/I:	5.28
Inicializa si mayor o igual que	41	SGE/I:	5.28
Inicializa si mayor que	43	SGT/I:	5.28
Inicializa si menor o igual que	45	SLE/I:	5.28

FIGURA 5.26 Códigos de operación y direcciones correspondientes para las tablas de decodificación 2 y 3 (Decod2 y Decod3). Los códigos de operación se muestran simbólicamente a la izquierda, seguidos por la dirección absoluta de la microinstrucción, la correspondiente etiqueta, y la figura donde se puede encontrar el microcódigo. Como los códigos de operación se muestran simbólicamente y están en el mismo sitio en ambas tablas, se puede utilizar la misma información para especificar las tablas de decodificación 2 y 3. Esta analogía es atribuible a la versión de inmediatos y de registros de las instrucciones de DLX que comparten el mismo microcódigo. Si se implementase una tabla en una ROM, contendría 64 entradas correspondientes al código de operación de 6 bits de DLX. De nuevo, muchas entradas redundantes o no especificadas sugieren el uso de un PLA para minimizar el costo del hardware.

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
23	Reg:	Temp	Pasa S2		B			Decod2		*Fuente2 = reg*
24	Imm:	Temp	Pasa S2		Inm			Decod3		*Fuente2 = inm.*
25	ADD/I:	C	ADD	A	Temp			Incond	Write2	*ADD*
26	SUB/I:	C	SUB	A	Temp			Incond	Write2	*SUB*
27	AND/I:	C	AND	A	Temp			Incond	Write2	*AND*
28	OR/I:	C	OR	A	Temp			Incond	Write2	*OR*
29	XOR/I:	C	XOR	A	Temp			Incond	Write2	*XOR*
30	SLL/I:	C	SLL	A	Temp			Incond	Write2	*SLL*
31	SRL/I:	C	SRL	A	Temp			Incond	Write2	*SRL*
32	SRA/I:	C	SRA	A	Temp			Incond	Write2	*SRA*
33	LHI:	C	SLL	Temp	Constante	16		Incond	Write2	*LHI*
34	Write2:						Rd←C	Incond	Ifetch	*Postescritura & ir a buscar siguiente instrucción*

FIGURA 5.27 Como los dos primeros estados de la Figura 5.15, las microinstrucciones 23 y 24 cargan Temp con un operando y a continuación saltan a la microinstrucción apropiada, dependiendo del código de operación en el IR. Se ejecuta una de las nueve instrucciones siguientes, dejando su resultado en C en la microinstrucción 34. C se vuelve a escribir en el registro especificado en el campo de registro destino de la macroinstrucción DLX que se encuentra en IR.

croinstrucción 57, y, análogamente, para la bifurcación y enlace registro con la microinstrucción 58. Estas microinstrucciones cargan el PC con la dirección de salto relativa al PC y guardan C en R31.

Rendimiento del control microcodificado para DLX

Antes de intentar mejorar el rendimiento o de reducir costes de control, se debe valorar el rendimiento existente. De nuevo, el proceso es contar los ciclos de reloj para cada instrucción, pero esta vez hay una mayor variedad en el rendimiento.

Todas las instrucciones ejecutan las microinstrucciones 0, 1 y 2 de la Figura 5.23, dando una base de 3 ciclos más los estados de espera, dependiendo de la repetición de la microinstrucción 1. Los ciclos de reloj para el resto de las categorías son:

4 para almacenamientos, más estados de espera
5 para cargar palabra, más estados de espera
6 para cargar byte o media palabra (con signo o sin signo), más estados de espera

3 para la ALU
4 para inicializar *(set)*
2 para salto no igual a cero (efectivo)
1 para salto no igual a cero (no efectivo)
1 para bifurcaciones
2 para bifurcación y enlaces

Utilizando la mezcla de instrucciones para GCC de la Figura C.4, y suponiendo una media de 1 estado de espera por cada el acceso a memoria, el CPI es 7,68. Este es más alto que el CPI del control cableado, ya que el test para interrupción emplea otro ciclo de reloj al comienzo, las cargas y almacenamientos son más lentos, y el salto igual a cero es más lento en el caso de que no sea efectivo.

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
35	SEQ/I:		SUB	A	Temp			¿Cero?	Set1	*Inicializa igual*
36		C	Pasa S2		Constante	0		Incond	Write4	*A ≠ T (inicializa a falso)*
37	SNE/I:		SUB	A	Temp			¿Cero?	Set0	*Iniciliza no igual*
38		C	Pasa S2		Constante	1		Incond	Write4	*A ≠ T (inicializa a cierto)*
39	SLT/I:		SUB	A	Temp			¿Negativo?	Set1	*Inicializa menor que*
40		C	Pasa S2		Constante	0		Incond	Write4	*A ≥ T (inicializa a falso)*
41	SGE/I:		SUB	A	Temp			¿Negativo?	Set0	*Inicializa GT o igual*
42		C	Pasa S2		Constante	1		Incond	Write4	*A ≥ T (inicializa a cierto)*
43	SGT/I:		RSUB	A	Temp			¿Negativo?	Set1	*Inicializa mayor que*
44		C	Pasa S2		Constante	0		Incond	Write4	*T ≥ A (inicializa a falso)*
45	SLE/I:		RSUB	A	Temp			¿Negativo?	Set0	*Inicializa LT o igual*
46		C	Pasa S2		Constante	1		Incond	Write4	*T ≥ A (inicializa a cierto)*
47	Set0:	C	Pasa S2		Constante	0		Incond	Write4	*Inicializa a 0 = falso*
48	Set1:	C	Pasa S2		Constante	1				*Inicializa a 1 = cierto*
49	Write4:						Rd←C	Incond	Ifetch	*Postescritura & ir a buscar siguiente instrucción*

FIGURA 5.28 Correspondiente a la Figura 5.16, este microcódigo realiza las instrucciones de inicialización (Set) de DLX. Como en la figura anterior, para ahorrar espacio estas mismas instrucciones se ejecutan o en la versión que utiliza registros o en la que utiliza inmediatos. El microcódigo que se encuentra en las microinstrucciones 43 y 45 es algo peculiar, ya que la resta Temp-A no es igual que el microcódigo anterior. Recordar que A–, Temp = Temp – A (ver Fig. 5.22).

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
50	Beq:		SUB	A	Constante	0		¿0?	Branch	*Instr es salto =0*
51								Incond	Ifetch	*≠0: no realizado*
52	Bne:		SUB	A	Constante	0		¿0?	Ifetch	*Instr es salto ≠0*
53	Branch:	PC	ADD	PC	inm16			Incond	Ifetch	*≠0: realizado*
54	Jump:	PC	ADD	PC	inm26			Incond	Ifetch	*Bifurca*
55	JReg:	PC	Pasa S1	A				Incond	Ifetch	*Bifurca registro*
56	JAL:	C	Pasa S1	PC						*Bifurca y enlaza*
57		PC	ADD	PC	inm26		R31←C	Incond	Ifetch	*Bifurca & guarda PC*
58	JALR:	C	Pasa S1	PC						*Bifurca & enlaza reg*
59		PC	Pasa S1	A			R31←C	Incond	Ifetch	*Bifurca & guarda PC*
60	Trap:	IAR	Pasa S1	PC						*Trap*
61		PC	Pasa S2		inm26			Incond	Ifetch	

FIGURA 5.29 El microcódigo para las instrucciones de salto y bifurcación de DLX, correspondiente a los estados de la Figura 5.17 y 5.18.

Reducción de coste y mejora de rendimiento de DLX cuando el control está microcodificado

El tamaño de una versión completamente no codificada, completamente, de la microinstrucción de DLX se calcula a partir del número de entradas de las Figuras 5.7 y 5.22 más el tamaño de los campos de direcciones de constantes y bifurcaciones. La mayor constante en los campos es 24, que requiere 5 bits, y la dirección más larga es 61, que requiere 6. La Figura 5.30 muestra los campos de las microinstrucciones, las anchuras no codificadas y codificadas. La codificación casi divide por dos el tamaño de la memoria de control.

La microinstrucción puede acortarse más introduciendo formatos múltiples de microinstrucción y combinando campos independientes.

	Dest	Operación ALU	Fuente1	Fuente2	Constante	Misc	Cond	Dirección bifurcación	Total
No codificada	7	11	9	9	5	6	10	6	= 63 bits
Codificada	3	4	4	4	5	3	4	6	= 33 bits

FIGURA 5.30 Anchura de campo en bits de los formatos de microinstrucciones codificadas y no codificadas. Observar que los campos de dirección Contante y Bifurcación no están codificados en este ejemplo, poniendo menores restricciones en el microprograma que utiliza el formato codificado.

Ejemplo

La Figura 5.31 muestra una versión codificada del formato de microinstrucción original de DLX y la versión con dos formatos: uno para las operaciones de la ALU y otro para las operaciones misceláneas y de salto. Se añade un bit para distinguir los dos formatos. La microinstrucción ALU/bifurcación (A/J) realiza las operaciones de la ALU especificadas en la microinstrucción; la dirección de la siguiente microinstrucción está especificada en la dirección de bifurcación. Para la microinstrucción de Transferencia/Misc/Salto (T/M/B), la ALU realiza el Paso S1, mientras que los campos Misc y Cont especifican el resto de las operaciones. El cambio principal en la interpretación de los campos en los nuevos formatos es que la condición de la ALU que se examina en el formato T/M/B referencia a la salida de la ALU de la microinstrucción A/J *previa*, ya que no hay operación ALU en el formato T/M/B. En ambos formatos, los campos de constante y bifurcación se combinan en un simple campo bajo la suposición de que no se utilizan a la vez. (Para el formato A/J, la aparición de una constante en un campo fuente da como resultado la búsqueda de la siguiente microinstrucción.) Los nuevos formatos disminuyen la anchura original de 33 bits a 22, pero el ahorro de tamaño real depende del número de microinstrucciones extras necesarias a causa de las opciones reducidas.

¿Cuál es el incremento en el número de microinstrucciones, comparado con el formato simple, para el microcódigo de la Figura 5.23?

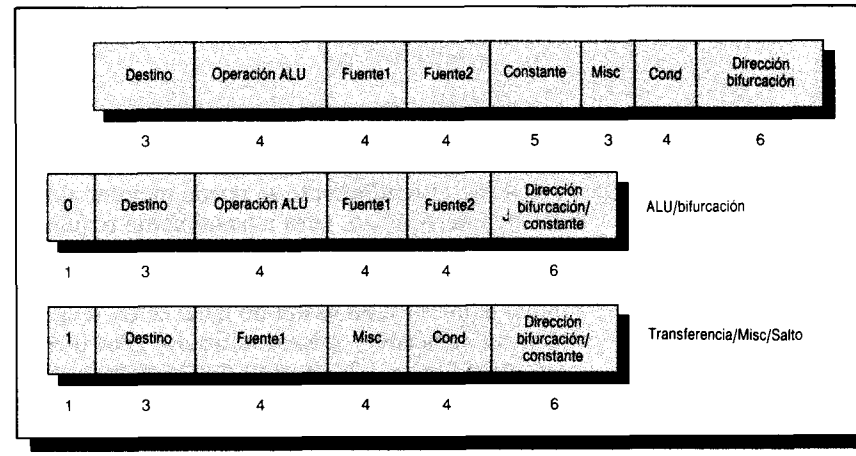


FIGURA 5.31 El formato de microinstrucción original de DLX en la parte superior y, debajo, la versión de doble formato. Observar que el campo Misc se expande de 3 a 4 bits en el T/M/B para hacer los dos formatos de la misma longitud.

Respuesta

La Figura 5.32 muestra el incremento del número de microinstrucciones sobre la Figura 5.23 debido a las restricciones de cada formato. Las cinco microinstrucciones del formato original se expanden a seis en el nuevo formato. La microinstrucción 2 es la única que se expande a dos microinstrucciones para este ejemplo.

Pos	Etiqueta	Tipo	Dest	ALU	S1	S2	Misc	Cond	Const/ bifurcación	Comentario
0	Ifetch:	M/T/B		---		---		¿Interrupc?	Intrpt	*Comprueba interrupción*
1	Iloop:	M/T/B		---		---	Lectura Instr	¿Mem?	Iloop	*IR←M[PC]; espera memoria*
2		A/J	PC	ADD	PC	Constante	---	---	4	*Incrementa PC*
3		M/T/B		---		---	AB←RF (Fich. de Reg.)	Decod1		
4	Intrpt:	A/J	IAR	Pasa S1	PC		---	---	5	*Interrupción*
5		A/J	PC	SUB	Temp	Temp	---	---	Ifetch	*PC←0 (t menús t=0) & ir a buscar siguiente instrucción*

FIGURA 5.32 Versión de la Figura 5.23 utilizando la microinstrucción de formato dual de la Figura 5.31. Observar que las microinstrucciones ALU/Bifurcación comprueban los campos S1 y S2 para un especificador constante, para ver si la siguiente dirección es secuencial (como en la microinstrucción 2); en otro caso van a la dirección de Bifurcación (como en las microinstrucciones 4 y 5). El microprogramador cambió la última microinstrucción para generar un cero, restando un registro de sí mismo en lugar de utilizar la constante 0. Utilizando la constante habría requerido una microinstrucción adicional, ya que este formato va a la siguiente instrucción secuencial si se utiliza una constante. (Ver Fig. 5.31.)

A veces, el rendimiento se puede mejorar al encontrar secuencias más rápidas de microcódigo, pero normalmente requieren cambios en el hardware. La instrucción «salta igual cero» emplea un ciclo de reloj extra cuando el salto no es efectivo con control cableado, pero dos con control microcodificado; aunque la instrucción «salta no igual cero» tenga el mismo rendimiento para el control cableado y microcodificado. ¿Por qué la primera difiere en rendimiento? La Figura 5.29 muestra que la microinstrucción 46 «salta si cero» para buscar la siguiente microinstrucción, lo que es correcto para la macroinstrucción «salta si no igual cero». La microinstrucción 50 también examina el cero para la macroinstrucción «salta si cero» y salta a la microinstrucción que carga el nuevo PC. El caso «no cero» es manipulado por la siguiente microinstrucción (51), que bifurca para buscar la siguiente instrucción —por tanto, un ciclo de reloj para el salto no efectivo sobre «no igual cero» y dos para el salto no efectivo sobre «igual cero». Una solución es sencillamente añadir «no cero» a las condiciones de salto de microcódigo en la Figura 5.22 y cambiar el microcódigo de «salto si igual» a la versión de la Figura 5.33. Como solamente hay diez condiciones de salto, añadir la undécima no requerirá más de los cuatro bits necesarios para una versión codificada de ese campo.

Este cambio hace caer el CPI de 7,68 a 7,63 para el control microcodificado; con todo, éste es todavía mayor que el CPI para el control cableado.

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
50	Beq:		SUB	A	Constante	0		¿no 0?	Ifetch	*Salto =0*
51		PC	ADD	PC	inm16			Incond	Ifetch	*=0: realizado*

FIGURA 5.33 Microcódigo de «salta no igual» de la Figura 5.29 reescrito al utilizar una condición «no cero» en la microinstrucción 44.

Ejemplo

Mejoremos el control microcodificado para que el CPI para GCC sea más próximo al CPI original con control cableado.

Respuesta

La causa principal del rendimiento es el test separado de la Figura 5.23 para las interrupciones. Modificando el hardware, decodifica1 puede matar dos pájaros de un tiro: además de saltar a las microinstrucciones apropiadas correspondientes al código de operación, también salta al microcódigo de interrupción si está pendiente alguna interrupción. La Figura 5.34 muestra el microcódigo revisado. Esta modificación ahorra un ciclo de reloj de cada instrucción, reduciendo el CPI a 6,63.

Pos	Etiqueta	Dest	ALU	S1 (Fuente 1)	S2 (Fuente 2)	C	Misc	Cond	Etiqueta bifurcación	Comentario
0	Ifetch:						Lee Instr	¿Mem?	Ifetch	*IR←M[PC]; espera memoria*
1		PC	ADD	PC	Constante	4	AB←RF (Fich. de Reg.)	Decod1		*También va a interrupción si está pendiente*
2	Intrpt:	IAR	SUB	PC	Constante	4				*Interrupción: deshace incremento PC*
3		PC	Pasa S2		Constante	0		Incond	Ifetch	*PC←0 & buscar y extraer siguiente instrucción*

FIGURA 5.34 Microcódigo revisado que aprovecha un cambio del hardware para hacer que decode1 vaya a la microinstrucción 2 si hay una interrupción pendiente. Esta microinstrucción debe invertir el incremento del PC en la microinstrucción, ya que se guarda el valor correcto.

5.8 Falacias y pifias

Pifia: El microcódigo que implementa una instrucción compleja puede no ser más rápido que el macrocódigo.

Durante algún tiempo, el microcódigo tenía la ventaja de que se buscaba en una memoria mucho más rápida que el macrocódigo. Desde que las caches

empezaron a utilizarse en 1968, el microcódigo no tiene este eje consistente en el tiempo de búsqueda. Sin embargo, el microcódigo todavía tenía la ventaja de utilizar temporalmente registros internos en el cálculo, que pueden ser útiles en máquinas con pocos registros de propósito general. La desventaja del microcódigo es que los algoritmos deben seleccionarse antes que se anuncie la máquina y no se pueden cambiar hasta el siguiente modelo de arquitectura; por otro lado, el macrocódigo puede utilizar mejoras en sus algoritmos en cualquier instante durante la vida de la máquina.

La instrucción «Index» del VAX proporciona un ejemplo: la instrucción comprueba si el índice está entre dos límites, uno de los cuales es habitualmente cero. El microcódigo del VAX-11/780 utiliza dos comparaciones y dos saltos para hacer esto, mientras que el macrocódigo puede realizar la misma comprobación en una comparación y un salto. El macrocódigo comprueba el índice frente al límite superior utilizando comparaciones **sin signo**, en lugar de comparaciones en complemento a dos. Esto trata un índice negativo (menor que cero y así falla la comparación) como si fuera un número muy grande, excediendo entonces el límite superior. (El algoritmo se puede utilizar con límites inferiores distintos de cero restando primero el límite inferior del índice.) Sustituir la instrucción «index» por este macrocódigo del VAX siempre mejora el rendimiento en el VAX-11/780.

Falacia: Si hay espacio en la memoria de control, nuevas instrucciones están libres de coste.

Como la longitud de la memoria de control es habitualmente una potencia de dos, a veces puede haber memoria de control disponible no utilizada para expandir el repertorio de instrucciones. Aquí, la analogía es la de construir una casa y descubrir, al terminarla, que se tiene suficiente terreno y material para añadir una habitación. Sin embargo, esta habitación no sería gratis, ya que tendría los costes de labor y mantenimiento durante la vida de la casa. El intento de añadir instrucciones «gratis» puede ocurrir solamente cuando el repertorio de instrucciones no sea fijo, como probablemente sea el caso en el primer modelo de computador. Como la compatibilidad del repertorio de instrucciones es un requerimiento a largo término, todos los futuros modelos de esta máquina estarán forzados a incluir estas instrucciones «gratis», aunque el espacio esté posteriormente muy solicitado. Esta expansión también ignora el coste de un mayor tiempo de desarrollo para verificar las instrucciones añadidas, así como la posibilidad de los costes de reparación de errores en ellas después que el hardware está construido.

Falacia: Los usuarios encuentran útil la memoria de control escribible.

Los errores en el microcódigo persuadieron a los diseñadores de minicomputadores y grandes computadores que podría ser más prudente utilizar como memorias de control RAM en vez de ROM. Haciendo eso se posibilitaría que los errores del microcódigo se reparasen con discos flexibles ofrecidos a los clientes en lugar de tener un ingeniero de campo poniendo tarjetas y sustituyendo chips. Algunos clientes y algunos fabricantes también decidieron que los usuarios deberían poder escribir el microcódigo; esta oportunidad se co-

noció como *memoria de control escribible* —writable control store— (WCS). En el momento en que se ofreció WCS, el mundo había cambiado para hacer WCS menos atractivo que el originalmente considerado:

- Las herramientas para escribir microcódigo eran mucho más pobres que las de escribir macrocódigo. (Los autores y muchos otros aprovecharon esa posibilidad para diseñar mejores herramientas de microprogramación.)
- Cuando se estaba expandiendo la memoria principal, WCS estaba limitado a microinstrucciones de 1-4KB. (Pocas tareas de programación son más difíciles que forzar el código a una memoria demasiado pequeña.)
- El control microcodificado se adaptó cada vez más al repertorio nativo de macroinstrucciones, haciendo la microprogramación menos útil para tareas distintas que aquellas para las que fue pensada.
- Con el advenimiento del tiempo compartido, los programas podían correr solamente durante milisegundos antes de conmutar a otras tareas. Esto significó que WCS tendría que intercambiarse si lo necesitaba más de un programa, y recargar WCS podía necesitar fácilmente más de algunos milisegundos.
- Tiempo compartido también significa que los programas deben de estar protegidos entre sí. Como, a este bajo nivel, los microprogramas pueden evitar todas las barreras de protección, los microprogramas escritos por los usuarios eran notoriamente poco fiables.
- La demanda creciente de memoria virtual significó que los microprogramas tenían que ser recomenzables —cualquier acceso a memoria podía forzar que se aplazase el cálculo.
- Finalmente, compañías como DEC que ofrecían WCS no proporcionaban soporte a los clientes que quisieran escribir microcódigo.

Muchos clientes pidieron WCS, pero pocos se beneficiaron de él. La desaparición de WCS ha sido por miles de pequeños cortes, y los computadores actuales no disponen WCS.

5.9 Observaciones finales

En su primer artículo [1953], Wilkes identificó las ventajas de la microprogramación que todavía hoy siguen siendo ciertas. Una de estas ventajas es que la microprogramación ayuda a acomodar cambios. Esto puede ocurrir al final del ciclo de desarrollo, donde cambiar simplemente algunos 0 por 1 en la memoria de control puede ahorrar, a veces, rediseños hardware. Una ventaja relativa es que, emulando otros repertorios de instrucciones en microcódigo, se simplifica la compatibilidad del software. La microprogramación también reduce el coste de añadir instrucciones más complejas a una microarquitectura estándar justamente al coste de algunas palabras más de la memoria de control (aunque exista la pifia de que una vez creado un repertorio de instrucciones suponiendo control microprogramado, es difícil construir una máquina

sin usarlo). Esta flexibilidad permite que la construcción hardware comience antes que el repertorio de instrucciones y el microcódigo se haya escrito completamente, debido a que la especificación del control es exactamente una tarea basada en programación. Finalmente, la microprogramación tiene ahora la ventaja adicional de disponer de un gran conjunto de herramientas que se han desarrollado para ayudar a escribir, editar, ensamblar y depurar microcódigo.

La desventaja del microcódigo siempre ha sido el rendimiento. Esto es porque la microprogramación es una esclava de la tecnología de memorias: la duración del ciclo de reloj está limitada por el tiempo de lectura, de las microinstrucciones, de la memoria de control. En los años cincuenta la microprogramación era impracticable, ya que virtualmente la única tecnología disponible para la memoria de control era la misma que se utilizaba para la memoria principal. A finales de los sesenta y principios de los setenta, las memorias semiconductoras estaban disponibles para memorias de control, mientras que la memoria principal se construía con núcleos. El factor de diez en tiempo de ciclo que diferenciaba las dos tecnologías abrió la puerta al microcódigo. La popularidad de la memoria cache en los años setenta cerró otra vez este salto, y las máquinas se construyeron de nuevo con la misma tecnología para la memoria de control y la memoria principal.

Por estas razones, los repertorios de instrucciones inventados desde 1985 no han contado con microcódigo. Aunque no gusta predecir el futuro —y menos escribiéndolo— es opinión de los autores que la microprogramación está limitada a la tecnología de memorias. Si en algún futuro la tecnología ROM llega a ser mucho más rápida que la RAM, o si las caches no son efectivas más tiempo, el microcódigo puede volver a ganar su popularidad.

5.10 Perspectivas históricas y referencias

Las interrupciones nos llevan de nuevo a los pioneros de la industria de computadores Eckert y Mauchly. Las interrupciones se utilizaron primero para señalar el desbordamiento («overflow») aritmético en la UNIVAC I y más tarde para alertar a una UNIVAC 1103 para que comenzase a tomar datos «*online*» en un túnel de viento (ver Codd [1962]). Después del éxito del primer computador comercial, el UNIVAC 1101 en 1953, el primer computador comercial que tenía interrupciones, el 1103, fue sacado a la luz. Las interrupciones las utilizó A. L. Leiner por primera vez para E/S en el National Bureau of Standards DYSEAC [Smotherman, 1989].

Maurice Wilkes aprendió el diseño de computadores en un *workshop* de verano de Eckert y Mauchly y después comenzó a construir el primer computador de programa almacenado operativo a escala completa —el EDSAC—. En esa experiencia se dio cuenta de la dificultad del control. Pensó en un control más centralizado utilizando una matriz de diodos y, después de ver el computador de Whirlwind en Estados Unidos, escribió:

Encontré que, en efecto, tiene un control centralizado basado en el uso de una matriz de diodos. Sin embargo, solamente era capaz de producir una

secuencia fija de 8 pulsos —una secuencia diferente para cada instrucción, pero nunca fija en lo que a una instrucción particular se refería—. No fue, creo, hasta que regresé a Cambridge cuando me di cuenta que la solución era cambiar la unidad de control por un computador en miniatura añadiendo una segunda matriz para determinar el flujo de control al micronivel y proporcionando microinstrucciones condicionales. [Wilkes, 1985, 178]

Wilkes [1953] se adelantó a su época al detectar ese problema. Desgraciadamente, la solución también estaba fuera de su época: para proporcionar el control, la microprogramación necesita una memoria rápida que no estaba disponible en los años cincuenta. Por tanto, las ideas de Wilkes quedaron principalmente como una conjetura académica durante una década, aunque en 1958 él construyó la EDSAC 2 utilizando control microprogramado con una ROM fabricada con núcleos magnéticos.

IBM introdujo en 1964 la microprogramación con la familia IBM 360. Antes de este evento, IBM era como muchas pequeñas firmas que vendían diferentes máquinas con sus propios precios y niveles de rendimiento, pero también con su propio repertorio de instrucciones. (Recordar que se hacía poca programación en lenguajes de alto nivel, de forma que los programas escritos para una máquina IBM no podían correr en otra.) Gene Amdahl, uno de los arquitectos jefe del IBM 360, dijo que los gestores de cada subsidiario aceptaron la familia de computadores 360 solamente porque estaban convencidos de que la microprogramación hacía posible la compatibilidad —si se podía tener el mismo hardware y microprogramarlo con diferentes repertorios de instrucciones, razonaron ellos, entonces también debe ser posible tener diferentes máquinas (hardware) y microprogramarlas para ejecutar el mismo repertorio de instrucciones. Para asegurarse de la viabilidad de la microprogramación, el vicepresidente de ingeniería de IBM visitó incluso subrepticiamente a Wilkes y tuvo una discusión «teórica» sobre los pros y contras del microcódigo. IBM pensó que la idea era tan importante para sus planes que potenció la tecnología de memorias dentro de la compañía para hacer factible la microprogramación.

Stewart Tucker de IBM cargó con la responsabilidad de llevar el software del IBM 7090 al nuevo IBM 360. Pensando en las posibilidades del microcódigo, sugirió expandir la memoria de control para incluir simuladores, o intérpretes, para máquinas más antiguas. Tucker [1967] acuñó para esto el término *emulación*, significando simulación completa a nivel microprogramado. Ocasionalmente, la emulación sobre el 360 era realmente más rápida que el hardware original. La emulación llegó a ser tan popular con los clientes en los primeros años de la 360, que a veces era difícil decir qué repertorio de instrucciones ejecutaba más programas.

Una vez que el gigante de la industria comenzó a utilizar microcódigo, el resto le siguió pronto. Una dificultad al adoptar microcódigo era que la tecnología de memorias que se necesitaba no estaba ampliamente disponible, pero esto se resolvió pronto con las ROM semiconductoras y posteriormente con las RAM. La industria de los microprocesadores siguió la misma historia, los recursos limitados de los primitivos chips forzaban el control cableado. Pero cuando aumentaron los recursos, las ventajas de diseños más simples y fáciles de cambiar persuadieron a muchos para utilizar la microprogramación.

Con la creciente popularidad de la microprogramación vinieron repertorios de instrucciones más sofisticados, incluyendo la memoria virtual. La microprogramación puede haber ayudado a la expansión de la memoria virtual, ya que el microcódigo hizo más fácil enfrentarse con las dificultades que surgían de realizar la correspondencia entre direcciones y recomenzar instrucciones. El modelo 138 de la IBM 370, por ejemplo, implementó completamente la memoria virtual en microcódigo sin soporte hardware.

Durante años, muchas microarquitecturas se hicieron cada vez más dedicadas para soportar el repertorio de instrucciones pensado ya que la reprogramación para un repertorio de instrucciones diferente fallaba a la hora de dar un rendimiento satisfactorio. Con el paso del tiempo llegaron memorias de control mucho mayores, y fue posible considerar una máquina tan elaborada como el VAX. Para ofrecer un VAX en un único chip en 1984, DEC redujo las instrucciones interpretadas por el microcódigo, ejecutando algunas instrucciones por software: el 20 por 100 de las instrucciones VAX eran responsables del 60 por 100 del microcódigo; sin embargo, sólo se ejecutaban el 0,2 por 100 del tiempo. La Figura 5.35 muestra la reducción de la memoria de control al reducir el repertorio de instrucciones. (El VAX está tan ligada al microcódigo que nos aventuramos a predecir que sería imposible construir un repertorio completo de instrucciones VAX sin microcódigo.) La microarquitectura de uno de los subconjuntos VAX más simples, el MicroVAX-1, se describe en Levy y Eckhouse [1989].

	Repertorio de instrucciones completo (VAX VLSI)	Subconjunto del repertorio de instrucciones (Micro VAX 32)
% instrucciones implementadas	100 %	80 %
Tamaño de memoria de control (bits)	480 K	64 K
Número de chips en procesador	9	2
% rendimiento de VAX-11/780	100 %	90 %

FIGURA 5.35 Interrumpiendo algunas instrucciones VAX y modos de direccionamiento, el control de almacenamiento se redujo casi un octavo. El segundo chip del subconjunto VAX es para punto flotante.

Mientras se estaba escribiendo este libro, un precedente legal relativo al microcódigo hizo aparición. La cuestión bajo litigio de *NEC frente a Intel* era que si el microcódigo es como la literatura, y por esa razón merece protección de derechos de copia (Intel), o si es como el hardware, que puede ser patentado pero no tiene derechos de copia (NEC). La importancia de esta cuestión radica en el hecho de que aunque es trivial obtener un derecho de copia, obtener una patente puede llevar mucho tiempo. Un programa se puede conseguir con derechos de copia; de ahí la pregunta que sigue: ¿qué es y no es un programa? Aquí hay una definición legislada:

Un «programa de computador» es un conjunto de sentencias o instrucciones para ser utilizadas directa o indirectamente en un computador con el fin de lograr un cierto resultado.

Después de años de preparación y juicios, un juez declaró que un microprograma era un programa. Los abogados de la parte que perdió impugnaron esta decisión por motivos de parcialidad. Habían descubierto que a través de un club de inversiones, el juez era propietario de 80 dólares de acciones pertenecientes al defendido para el que falló a favor. (La suma tentadora realmente fue de sólo 80 dólares; ¡altamente frustrante para uno de los autores que actuó como un testigo experto en el caso!) El caso fue visto nuevamente, y el nuevo juez dictó que «microcódigo... cae de lleno en la definición de un "programa de computador"...» [Gray, 1989, 4]. Por supuesto, el hecho que dos jueces en dos juicios diferentes tomen la misma decisión no significa que el caso esté cerrado —hay todavía niveles superiores de apelación a los que acudir.

Referencias

CLARK, D. W., P. J. BANNON, AND J. B. KELLER [1988]. «Measuring VAX 8800 performance with a histogram hardware monitor», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, Hawaii, 176-185.

CODD, E. F. [1962]. «Multiprogramming», in F. L. Alt and M. Rubinoff, *Advances in Computers*, vol. 3, Academic Press, New York, 82.

EMER, J. S. AND D. W. CLARK [1984]. «A characterization of processor performance in the VAX-11/780», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301-310.

GRAY, W. P. [1989]. Memorandum of Decision, núm. C-84-20799-WPG, U. S. District Court for the Northern District of California (February 7, 1989).

LEVY, H. M. AND R. H. ECKHOUSE, JR. [1989]. *Computer Programming and Architecture: The VAX*, 2nd ed., Digital Press, Bedford, Mass. 358-372.

MCKEVITT, J., ET AL. [1977]. *8086 Design Report*, internal memorandum.

PATTERSON, D. A. [1983]. «Microprogramming», *Scientific American* 248:3 (March), 36-43.

REIGEL, E. W., U. FABER, AND D. A. FISCHER [1972]. «The Interpreter —a microprogrammable building block system», *Proc. AFIPS 1972 Spring Joint Computer Conf.*, 40, 705-723.

SMOTHERMAN, M. [1989]. «A sequencing-based taxonomy of I/O systems and review of historical machines», *Computer Architecture News*, 17:5 (September), 5-15.

TUCKER, S. G. [1967]. «Microprogram control for the System/360», *IBM Systems Journal*, 6:4, 222-241.

WILKES, M. V. [1953]. «The best way to design an automatic calculating machine», in *Manchester University Computer Inaugural Conf.*, 1951, Ferranti, Ltd., London. (Not published until 1953.) Reprinted in «The Genesis of Microprogramming», in *Annals of the History of Computing*, 8:116.

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer*, The MIT Press, Cambridge, Mass.

WILKES, M. V., AND J. B. STRINGER [1953]. «Microprogramming and the design of the control circuits in an electronic digital computer», *Proc. Cambridge Philosophical Society*, 49:230-238. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 158-163, and in «The Genesis of Microprogramming» in *Annals of the History of Computing*, 8:116.

EJERCICIOS

Si los temas sobre diagramas de estados finitos y microprogramación se tratan a nivel de revisión, quizá desee pasar por alto las cuestiones 5.5 a 5.14.

5.1 [15/10/15/15] <5.5>Una técnica que trata de obtener lo mejor de las microarquitecturas verticales y horizontales es una memoria de control de *dos niveles*, como ilustra la Figura 5.36. Trata de combinar tamaños pequeños de memoria de control con instrucciones largas. Para evitar confusiones, el nivel inferior utiliza el prefijo *nano-*, dando lugar a los términos «nanoinstrucción», «nanocódigo», etc. Esta técnica se utilizó en el Motorola 68000, 68010 y 68020, pero se originó en la D-máquina de Burroughs [Reigel, Faber y Fischer, 1972]. La idea es que el primer nivel tiene muchas instrucciones verticales que apuntan a las pocas instrucciones horizontales únicas del segundo nivel. La D-máquina de Burroughs fue un computador de propósito general que ofrecía memoria de control escribible. Sus microinstrucciones eran de 16 bits, de los cuales 12 especificaban una nanodirección, y las nanoinstrucciones eran de 56 bits. Un intérprete del repertorio de instrucciones utilizaba 1124 microinstrucciones y 123 nanoinstrucciones.

a) [15] <5.5> ¿Cuál es la fórmula general que muestra cuándo un esquema de memoria de control de dos niveles como la D-máquina de Burroughs utiliza menos bits que una memoria de control de un nivel? Suponer que hay M microinstrucciones de a bits y N nanoinstrucciones de b bits de ancho.

b) [10] ¿Consiguió la memoria de control de dos niveles de la D-máquina reducir el tamaño de memoria de control frente a una memoria de control de un nivel para el intérprete?

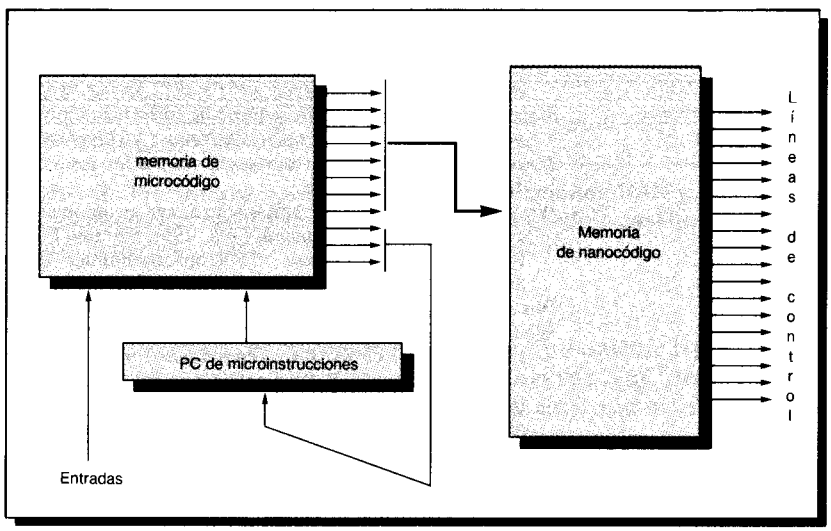


FIGURA 5.36 Implementación microprogramada a dos niveles mostrando las relaciones entre microcódigo y nanocódigo.

c) [15] Una vez que se optimizó el código para mejorar el CPI en un 10 por 100, el código resultante tenía 940 microinstrucciones y 161 nanoinstrucciones. ¿Consiguió la memoria de control de dos niveles de la D-máquina al reducir el tamaño de la memoria de control frente a una memoria de control para el intérprete **optimizado**?

d) [15] ¿Incrementó o decrementó la optimización el número total de bits necesarios para especificar el control? ¿Por qué decreció el número de instrucciones y aumentó el de nanoinstrucciones?

5.2 [15] <5.5, 5.6> Una ventaja del microcódigo es que puede manipular casos raros sin el gasto de invocar al sistema operativo antes de ejecutar la rutina de trap. Suponer una máquina con un CPI de 1.5 y con un sistema operativo que emplea 100 ciclos de reloj en un trap antes de que pueda ejecutar el código adecuado. Suponer que el código del trap necesita 10 ciclos de reloj, tanto si es microcódigo como macrocódigo. Para una instrucción que se presenta el 5 por 100 del tiempo, ¿qué porcentaje del tiempo debe generar un trap antes que la implementación en microcódigo sea globalmente un 1 por 100 más rápido que una implementación en macrocódigo?

5.3 [20/20/30] <4.2, 5.5, 5.6> Exploremos el impacto de reducir una arquitectura como la descrita en la Figura 5.35. Suponer que se suprimiera del VAX la instrucción `MOVC3`.

a) [20] Escribir el macrocódigo VAX que sustituye a `MOVC3`.

b) [20] Suponer que los operandos están colocados en los registros R0, R1 y R2 después de un trap. Utilizando los datos para COBOLX de la Figura C.1 en el Apéndice C sobre utilización de instrucciones (suponiendo que todas las `MOVC_` son `MOVC3`) y suponiendo que la `MOVC3` media transfiere 15 bytes, ¿cuál sería el cambio de porcentaje en el recuento de instrucciones si `MOVC3` no fuese interpretado por microcódigo? (Ignorar el coste de los traps para esta instruccion.)

c) [30] Si se tiene acceso a un VAX, mida la velocidad de `MOVC3` frente a una versión macrocódigo de la rutina de la parte a. Suponiendo que el gasto de un trap sea de 20 ciclos de reloj, ¿cuál es el impacto en el rendimiento de los traps software para `MOVC3`?

5.4 [15] <5.6> Suponer que tenemos una máquina con una duración del ciclo de reloj de 10 ns y un CPI base de 5. Debido a las posibilidades de interrupciones debemos tener registros extras que contengan copia de valores de los registros al comienzo de la instrucción. Estos registros habitualmente se denominan *registros sombra* (*shadow*). Suponer que la instrucción media tiene dos operandos de registro que se deben restaurar en una interrupción. La frecuencia de interrupciones es de 100 interrupciones por segundo, y el coste de la interrupción es 30 ciclos más el tiempo de restaurar los registros sombra, cada uno de los cuales necesita 10 ciclos. ¿Cuál es el CPI efectivo después de contabilizar las interrupciones? ¿Cuál es la pérdida de rendimiento de las interrupciones?

5.5-5.7 Dado el diseño del procesador y el diagrama de estados finitos para DLX como se modificó al final de la parte del control cableado de la Sección 5.7, explorar el impacto en el rendimiento de los siguientes cambios. En cada caso mostrar la porción modificada de la máquina de estados finitos, describir los cambios para el procesador (si es necesario), el cambio en el número de estados, y calcular el cambio del CPI utilizando las estadísticas de mezcla de instrucciones DLX de la Figura C.4 para GCC. Mostrar las razones para el cambio.

5.5 [12] <5.7> Igual que el cambio de las instrucciones de la ALU del segundo ejemplo de la Sección 5.7 y mostrado en las Figuras 5.20 y 5.21, eliminar los estados que cargan Temp para las instrucciones Set (Inicializar) de la Figura 5.16, primero añadiendo la opción «X» y, después, incrementando el número de estados.

5.6 [15] <5.7> Suponer que se optimizó la interfaz de memoria para que no fuese necesario cargar el MAR antes de un acceso a memoria, ni transferir el dato al MDR para una lectura o escritura. En lugar de ello, cualquier registro en el bus S1 puede especificar la dirección, cualquier registro en el S2 puede suministrar el dato de una escritura, y cualquier registro en el bus Dest puede recibir el dato de una lectura.

5.7 [22] <5.7> La mayoría de los computadores solapan la búsqueda de la siguiente instrucción con la ejecución de la instrucción actual. Proponer un esquema que solape todas las búsquedas de instrucción excepto bifurcaciones, saltos y almacenamientos. Reorganice la máquina de estados finitos para que la instrucción esté ya buscada, e incluso parcialmente decodificada.

5.8 [15] <5.7> El ejemplo de la Sección 5.7 supone todo, excepto el desplazador, que puede escalarse a 9 ns. Por desgracia, el sistema de memoria raramente puede escalarse tan fácilmente como la CPU. Replantear el análisis de este ejemplo, pero esta vez suponer que el número medio de estados de espera de memoria es 2, en un ciclo de reloj de 9 ns, frente a uno en 10 ns además de ralentizar los desplazamientos.

5.9-5.14 Estas cuestiones están dirigidas al uso del control microcodificado de DLX, como se muestra en las Figuras 5.23, 5.25 y 5.27-5.29. En cada caso mostrar la parte modificada del microcódigo; describir los cambios en el procesador (si es necesario), los campos de la microinstrucción (si es necesario) y el cambio en el número de microinstrucciones; y calcular el cambio del CPI utilizando las estadísticas de mezcla de instrucciones DLX del Apéndice C para GCC. Mostrar las razones del cambio.

5.9 [15] <5.7> Igual que en el cambio de las instrucciones de la ALU del segundo ejemplo de la Sección 5.7, eliminar las microinstrucciones que cargan Temp para las instrucciones Set de la Figura 5.28, añadiendo primero la opción «X» y después incrementando el número de microinstrucciones.

5.10 [25] <5.7> Continuando el ejemplo de la Figura 5.32, reescribir el microcódigo de la Figura 5.29, utilizando las microinstrucciones de formato dual de la Figura 5.31. ¿Cuál es la frecuencia relativa de cada tipo de microinstrucción? ¿Cuál es el ahorro en tamaño de memoria de control frente al formato original DLX? ¿Cuál es el cambio del CPI?

5.11 [20] <3.4, 5.7> Cargar byte y Cargar media palabra necesita un ciclo de reloj más que Cargar palabra debido a la alineación de los datos (ver Fig. 3.10 y Fig. 5.25). Proponer un cambio que elimine el ciclo extra para estas instrucciones. ¿Cómo afecta este cambio al CPI de GCC? ¿Cómo afecta al CPI de TeX?

5.12 [20] <5.6, 5.7> Cambiar el microcódigo para realizar los siguientes tests de interrupción: fallo *(fault)* de página, desbordamiento («overflow») o desbordamiento a cero («underflow») aritmético, accesos a memoria mal alineados, y utilización de instrucciones indefinidas. Hacer los cambios que sean necesarios en la microarquitectura y el formato de microinstrucción. ¿Cuál es el cambio en tamaño y rendimiento al realizar estos tests?

5.13 [20] <5.7> El diseñador de computadores debe ser cuidadoso de no confeccionar su diseño muy dirigido a un único programa particular. Reevaluar el impacto del rendimiento de todas las mejoras de rendimiento del ejemplo de los Ejercicios 5.9 a 5.12 utilizando esta vez el dato del promedio de la mezcla de instrucciones de la Figura C.4. ¿Cómo afectan los programas a las evaluaciones?

5.14 [20] <5.6, 5.7> Empezando con el microcódigo en las Figuras 5.27 y 5.34, revisar el microcódigo para que la siguiente macroinstrucción se extraiga tan pronto como sea posible durante las instrucciones de la ALU. Suponer un sistema de memoria «perfecto», empleando un ciclo de reloj por referencia a memoria. Aunque técnicamente esta mejora acelera las instrucciones que **siguen** a las instrucciones de la ALU, la forma más fácil de contabilizar este rendimiento mayor es como instrucciones ALU más rápidas. ¿Cuánto más rápidas son las instrucciones de la ALU? ¿Cómo afecta al rendimiento global según las estadísticas de GCC?

5.15 [30] <4, 5.6> Si se tiene acceso a una máquina que utiliza uno de los repertorios de instrucciones del Capítulo 4, determinar la latencia de interrupciones en el peor caso para esa implementación de la arquitectura. Asegurarse de que se está midiendo la latencia bruta de la máquina y **no** el coste del sistema operativo.

5.16 [30] <5.6> Algunas veces, los arquitectos de computadores se han visto forzados a soportar instrucciones que nunca se publicaron en el manual original del repertorio de instrucciones. Esta situación surge porque algunos programas que se crean inadvertidamente inicializan campos de instrucciones no utilizados con valores diferentes a los que el arquitecto esperaba, lo cual causa estragos cuando el arquitecto trata de utilizar aquellos valores para extender el repertorio de instrucciones. IBM resolvió ese problema en el Sistema 370 generando un trap por cada posible campo indefinido. Intentar ejecutar en un computador instrucciones con campos indefinidos para ver qué ocurre. ¿Computan algo útil sus nuevas instrucciones? Si es así, ¿utilizaría estas nuevas instrucciones en los programas?

5.17 [35] <5.4, 5.5, 5.7> Tomar el camino de datos de la Figura 5.1 y construir un simulador que pueda realizar cualquiera de las operaciones necesarias para implementar el repertorio de instrucciones de DLX. Implementar ahora el repertorio de instrucciones de DLX utilizando:

Control microprogramado, y
Control cableado.

Para el control cableado, ver si se pueden encontrar programas de minimización de PLA y de asignación de estados para reducir el coste del control. A partir de estos dos diseños, determinar el rendimiento de cada implementación y el coste en términos de puertas o de área de silicio.

5.18 [35] <2.2, 5.5, 5.7> Las analogías entre las microinstrucciones y las macroinstrucciones de DLX sugieren que el rendimiento se pueda aumentar escribiendo un programa que traduzca del macrocódigo DLX a microcódigo DLX. (Esta es la intuición que inspiró WCS.) Escribir el programa y valorarlo con un benchmark. ¿Cuál es la expansión del tamaño del código resultante?

5.19 [50] <2.2, 4.4, 5.10> Se han realizado intentos recientes para ejecutar el software existente en máquinas de control cableado construyendo simuladores afinados manualmente para máquinas populares. Escribir un simulador para el repertorio de instrucciones del 8086. Ejecutar algunos programas existentes del IBM PC, y ver lo rápido que es su simulador con respecto a un 8086 de 8 MHz.

5.20 [Discusión] <4, 5.5, 5.10> Hipótesis: si la primera implementación de una arquitectura usa microprogramación, afecta a la arquitectura a nivel lenguaje máquina. ¿Por qué puede ser esto cierto? Observando los ejemplos del Capítulo 4 o de otra parte, dar soporte o evidencia contradictoria de máquinas reales. ¿Qué máquinas usarán siempre microcódigo? ¿Por qué? ¿Qué máquinas no utilizarán nunca microcódigo? ¿Por qué? ¿Qué implementación del control piensa que tenía el arquitecto en mente durante el diseño de una arquitectura a nivel lenguaje máquina?

5.21 [Discusión] <5.5, 5.10> Wilkes inventó la microprogramación para simplificar la construcción del control. Desde 1980 ha habido una explosión de software de diseño asistido por computador cuyo objetivo es también simplificar la construcción del control. Hipótesis: los avances en software de diseño asistido por computador han hecho innecesaria la microprogramación. Encontrar evidencias que soporten y refuten la hipótesis.

5.22 [Discusión] <5.10> Las instrucciones de DLX y las microinstrucciones de DLX tienen muchas analogías. ¿Qué haría difícil a un compilador producir microcódigo para DLX en lugar de macrocódigo? ¿Qué cambios en la microarquitectura harían al microcódigo de DLX más útil para esta aplicación?

Es suficiente un problema de triple cauce.

Sir Arthur Conan Doyle, *Las Aventuras de Sherlock Holmes*

6.1 ¿Qué es la segmentación?

6.2 Segmentación básica para DLX

6.3 Haciendo que funcione la segmentación

6.4 El principal obstáculo de la segmentación: riesgos de la segmentación

6.5 Qué hace difícil de implementar la segmentación

6.6 Extensión de la segmentación de DLX para manipular operaciones multiciclo

6.7 Segmentación avanzada: planificación dinámica de la segmentación

6.8 Segmentación avanzada: aprovechando más el paralelismo de nivel de instrucción

6.9 Juntando todo: un VAX segmentado

6.10 Falacias y pifias

6.11 Observaciones finales

6.12 Perspectiva histórica y referencias

Ejercicios

6 Segmentación *(Pipelining)*

6.1 ¿Qué es la segmentación?

La segmentación *(pipelining)* es una técnica de implementación por la cual se solapa la ejecución de múltiples instrucciones. Hoy día, la segmentación es la técnica de implementación clave utilizada para hacer CPU rápidas.

La segmentación es como una línea de ensamblaje: cada etapa de la segmentación completa una parte de la instrucción. Como en una línea de ensamblaje de automóviles, el trabajo que va a realizar en una instrucción se descompone en partes más pequeñas, cada una de las cuales necesita una fracción del tiempo necesario para completar la instrucción completa. Cada uno de estos pasos se define como *etapa de la segmentación* o *segmento.* Las etapas están conectadas, cada una a la siguiente, para formar una especie de cauce —las instrucciones entran por un extremo, son procesadas a través de las etapas y salen por el otro extremo.

La productividad de la segmentación está determinada por la frecuencia con que una instrucción salga del cauce. Como las etapas están conectadas entre sí, todas las etapas deben estar listas para proceder al mismo tiempo. El tiempo requerido para desplazar una instrucción, un paso, a lo largo del cauce es un ciclo máquina. La duración de un ciclo máquina está determinada por el tiempo que necesita la etapa más lenta (porque todas las etapas progresan a la vez). Con frecuencia, el ciclo máquina es un ciclo de reloj (a veces dos, o raramente más), aunque el reloj puede tener múltiples fases.

El objetivo del diseñador es equilibrar la duración de las etapas de la segmentación. Si las etapas están perfectamente equilibradas, entonces el tiempo

por instrucción de la máquina segmentada —suponiendo condiciones ideales (p. e., no atascos)— es igual a

$$\frac{\text{Tiempo por instrucción en la máquina no segmentada}}{\text{Número de etapas de la segmentación}}$$

Bajo estas condiciones, la mejora de velocidad debida a la segmentación es igual al número de etapas. Sin embargo, habitualmente, las etapas no están perfectamente equilibradas; además, la segmentación involucra algún gasto. Así, el tiempo por instrucción en la máquina segmentada no tendrá su valor mínimo posible, aunque pueda estar próximo (digamos en un 10 por 100).

La segmentación consigue una reducción en el tiempo de ejecución medio por instrucción. Esta reducción se puede obtener decrementando la duración del ciclo de reloj de la máquina segmentada o disminuyendo el número de ciclos de reloj por instrucción, o haciendo ambas cosas. Normalmente, el mayor impacto está en el número de ciclos de reloj por instrucción, aunque el ciclo de reloj es, con frecuencia, más corto en una máquina segmentada (especialmente en supercomputadores segmentados). En las secciones de segmentación segmentada avanzada de este capítulo veremos qué profundidad de segmentación se puede utilizar para decrementar el ciclo de reloj y mantener un CPI bajo.

La segmentación es una técnica de implementación, que explota el paralelismo entre las instrucciones de un flujo secuencial. Tiene la ventaja sustancial que, de forma distinta a algunas técnicas de aumento de velocidad (ver Caps. 7 y 10), no es visible al programador. En este capítulo cubriremos primero el concepto de segmentación, utilizando DLX y una versión simplificada de su segmentación. Después, examinaremos los problemas que introduce la segmentación y el rendimiento que se puede alcanzar bajo situaciones típicas. Más tarde, examinaremos en el capítulo las técnicas avanzadas que se pueden utilizar para superar las dificultades que se encuentran en máquinas segmentales y que puedan bajar el rendimiento alcanzable con la segmentación.

Utilizaremos extensiones de DLX porque su simplicidad hace más fácil demostrar los principios de la segmentación. Se aplican los mismos principios a repertorios de instrucciones más complejos, aunque las segmentaciones correspondientes sean más complejas. Veremos un ejemplo de esta segmentación en la sección Juntando todo.

6.2 Segmentación básica para DLX

Recordar que en el Capítulo 5 (Sección 5.3) explicamos cómo podía implementarse DLX con cinco pasos básicos de ejecución:

1. IF-búsqueda de la instrucción
2. ID-decodificación de la instrucción y búsqueda de registros
3. EX-ejecución y cálculo de direcciones efectivas

Número de instrucción	Número de reloj 1	2	3	4	5	6	7	8	9
Instrucción i	IF	ID	EX	MEM	WB				
Instrucción $i + 1$		IF	ID	EX	MEM	WB			
Instrucción $i + 2$			IF	ID	EX	MEM	WB		
Instrucción $i + 3$				IF	ID	EX	MEM	WB	
Instrucción $i + 4$					IF	ID	EX	MEM	WB

FIGURA 6.1 Segmentación sencilla de DLX. En cada ciclo de reloj se busca otra instrucción y comienza sus cinco pasos de ejecución. Si se comienza una instrucción en cada ciclo de reloj, el rendimiento será cinco veces el de una máquina sin segmentación.

4. MEM-acceso a memoria
5. WB-postescritura

Podemos segmentar DLX buscando sencillamente una nueva instrucción **en cada ciclo de reloj.** Cada uno de los pasos anteriores se convierte en una *etapa de la segmentación* —un paso de la segmentación— dando como resultado el patrón de ejecución mostrado en la Figura 6.1. Aunque cada instrucción necesita cinco ciclos de reloj, durante cada ciclo de reloj el hardware está ejecutando alguna parte de cinco instrucciones diferentes.

La segmentación incrementa la productividad de instrucciones de la CPU —el número de instrucciones completadas por unidad de tiempo—, pero no reduce el tiempo de ejecución de una instrucción individual. En efecto, habitualmente incrementa ligeramente el tiempo de ejecución de cada instrucción debido al gasto en el control de la segmentación. El incremento en la productividad de instrucciones significa que un programa corre más rápido y tiene menor tiempo total de ejecución, ¡aun cuando ninguna instrucción se ejecute con más rapidez!

El hecho de que el tiempo de ejecución de cada instrucción permanezca inalterado, pone límites a la profundidad práctica de la segmentación, como veremos en la siguiente sección. Otras consideraciones de diseño limitan la frecuencia de reloj que puede alcanzarse mediante una segmentación más profunda. La consideración más importante es el efecto combinado del retardo de los cerrojos y el sesgo del reloj *(clock skew).* Los cerrojos son necesarios entre etapas del cauce, sumando al tiempo de preparación *(setup)* el retardo a través de los cerrojos en cada período de reloj. El sesgo de reloj también contribuye al límite inferior del ciclo de reloj. Una vez que el ciclo de reloj sea tan pequeño como la suma del sesgo de reloj y el gasto de los cerrojos, no es útil una mayor segmentación.

Ejemplo

Considerar una máquina no segmentada con cinco pasos de ejecución cuyas duraciones son 50 ns, 50 ns, 60 ns, 50 ns, y 50 ns. Suponer que, debido al tiempo de preparación y sesgo de reloj, segmentar la máquina añade 5 ns de gasto a cada etapa de ejecución. Ignorando cualquier impacto de latencia,

¿cuánta velocidad se ganará con la segmentación en la frecuencia de ejecución de las instrucciones?

Respuesta

La Figura 6.2 muestra el patrón de ejecución en la máquina no segmentada y en la segmentada.

El tiempo de ejecución medio por instrucción en la máquina no segmentada es

Tiempo de ejecución medio de instruccion =

$$50 + 50 + 60 + 50 + 50 \text{ ns} = 260 \text{ ns}$$

En la implementación segmentada, el reloj debe ir a la velocidad de la etapa más lenta, más el gasto, que será 60 + 5 = 65 ns; éste es el tiempo de ejecución medio por instrucción. Por tanto, la aceleración obtenida de la segmentación es

$$\text{Aceleración} = \frac{\text{Tiempo medio de instrucción sin segmentación}}{\text{Tiempo medio de instrucción con segmentación}}$$

$$= \frac{260}{65} = 4 \text{ veces}$$

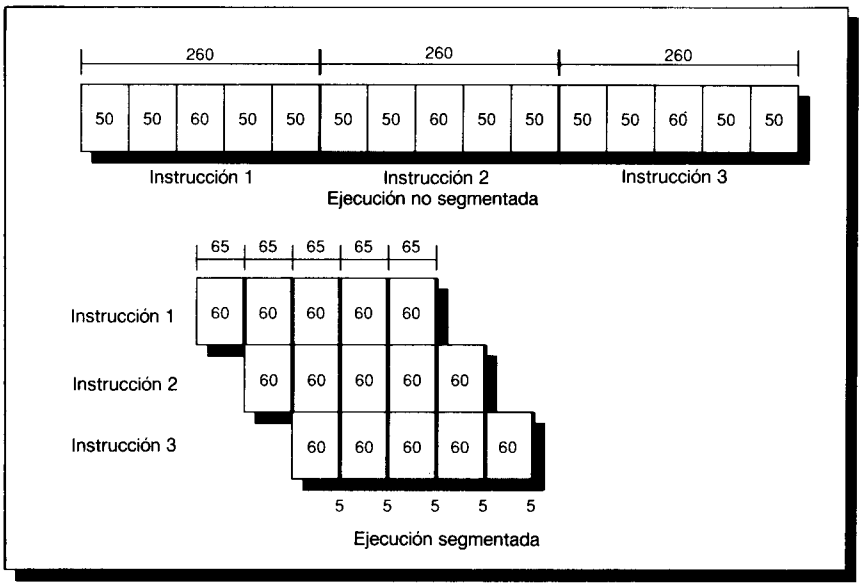


FIGURA 6.2 Patrón de ejecución para tres instrucciones mostradas para las versiones segmentada y no segmentada. En la versión no segmentada, las tres instrucciones se ejecutan secuencialmente. En la versión segmentada, las áreas sombreadas representan el gasto de 5 ns por etapa de cauce. La duración de las etapas debe ser la misma para todas: 60 ns más el gasto de 5 ns. La latencia de una instrucción aumenta, de 260 ns en la máquina no segmentada, a 325 ns en la máquina segmentada.

El gasto de 5 ns establece esencialmente un límite en la efectividad de la segmentación. Si el gasto no está afectado por los cambios del ciclo de reloj, la Ley de Amdahl nos dice que el gasto limita la velocidad.

Debido a que los cerrojos de un diseño segmentado pueden tener un impacto significativo sobre la velocidad de reloj, los diseñadores han investigado cerrojos que permitan la frecuencia de reloj más alta posible. El cerrojo de Earle (inventado por J. G. Earle [1965]) tiene tres propiedades que lo hacen especialmente útil en las máquinas segmentadas. Primero, es relativamente insensible al sesgo de reloj. Segundo, el retardo a través del cerrojo, es siempre un retardo constante de dos puertas, evitando la introducción de sesgos cuando pasan los datos a través del cerrojo. Finalmente, se pueden hacer dos niveles de lógica, en el cerrojo, sin incrementar su tiempo de retardo. Esto significa que dos niveles de lógica de la segmentación pueden solaparse con el cerrojo; así puede ocultarse la mayoría del gasto del cerrojo. En este capítulo no se analizarán los diseños segmentados a este nivel de detalle. El lector interesado deberá ver Kunkel y Smith [1986].

Las dos secciones siguientes añadirán refinamientos y plantearán algunos problemas que pueden presentarse en esta segmentación. En esta explicación (hasta el último segmento de la Sección 6.5) nos centraremos en la segmentación para la parte entera de DLX. Las complicaciones que surgen en la segmentación del punto flotante serán tratadas en la Sección 6.6.

6.3 Haciendo que funcione la segmentación

Su instinto es correcto si encuentra difícil pensar que la segmentación es tan simple como esto, porque no lo es. En ésta y en las tres secciones siguientes, haremos «real» nuestra segmentación de DLX tratando los problemas que introduce la segmentación.

Para comenzar, tenemos que determinar qué ocurre en cada ciclo de reloj de la máquina y asegurarnos que el solapamiento de las instrucciones no utiliza los recursos más allá de sus posibilidades. Por ejemplo, a una simple ALU no se le puede pedir que calcule una dirección efectiva y realice una operación de resta al mismo tiempo. Como veremos, la simplicidad del repertorio de instrucciones de DLX hace relativamente fácil la evaluación de recursos.

Las operaciones que se presentan durante la ejecución de las instrucciones, que se explicaron en la Sección 5.3 del Capítulo 5, se modifican para que se ejecuten de forma segmentada como se muestra en la Figura 6.3. La figura lista las principales unidades funcionales de nuestra implementación de DLX, las etapas de la segmentación y lo que tiene que ocurrir en cada etapa. El eje vertical está etiquetado con las etapas de la segmentación, mientras que el horizontal muestra los principales recursos. Cada intersección muestra lo que ocurre para ese recurso en esa etapa. En la Figura 6.4 mostramos información similar, utilizando el tipo de instrucción como eje horizontal. La combinación de instrucciones que pueden estar siendo ejecutadas, en cualquier instante, es arbitraria. Así, el combinado necesita de todos los tipos de instrucciones en cualquier etapa de la segmentación para determinar qué recursos se necesitan en esa etapa.

Etapa	Unidad PC	Memoria	Camino de datos
IF	PC←PC+4	IR←Mem[PC];	
ID	PC1←PC	IR1←IR	A←Rs1; B←Rs2;
EX			DMAR←A + $(IR1_{16})^{16}$##$IR1_{16..31}$; o ALUoutput←A *op* (B or $(IR1_{16})^{16}$##$IR1_{16..31}$); o ALUoutput←PC1 + $(IR1_{16})^{16}$##$IR1_{16..31}$; cond←(Rs1 *op* 0); SMDR←B;
MEM	if (cond) PC←ALUoutput	LMDR←Mem[DMAR] o Mem[DMAR]←SMDR	ALUoutput1←ALUoutput
WB			Rd←ALUoutput1 o LMDR

FIGURA 6.3 La tabla muestra las principales unidades funcionales y lo que puede ocurrir en cada etapa de la segmentación en cada unidad. En algunas etapas no pueden ocurrir todas las acciones listadas, porque se aplican bajo diferentes hipótesis sobre la instrucción. Por ejemplo, hay tres operaciones de la ALU durante la etapa EX. La primera se presenta sólo en una carga o almacenamiento; la segunda en operaciones de la ALU (en la que la entrada es B o los 16 bits inferiores de IR, de acuerdo con que la instrucción sea de registro-registro o registro-inmediato); la tercera operación se presenta sólo en los saltos. Por simplicidad, hemos mostrado sólo el caso de salto —las bifurcaciones añaden un desplazamiento de 26 bits al PC. Las variables `ALUoutput1`, `PC1`, e `IR1` guardan valores para usarlos en etapas posteriores de la segmentación. Diseñar el sistema de memoria para soportar una carga o almacenamiento de datos en cada ciclo de reloj es un reto; ver el Capítulo 8 para una discusión en profundidad. Este tipo de tabla y el de la Figura 6.4 está fundamentalmente basado en las tablas de reserva de la segmentación de Davidson [1971].

Cada etapa de la segmentación se activa en cada ciclo de reloj. Esto requiere que todas las operaciones de una etapa se completen en un reloj y que cualquier combinación de operaciones se puede presentar a la vez. Aquí están las implicaciones más importantes para el camino de datos, como se especificó en el Capítulo 5:

1. El PC se debe incrementar en cada ciclo de reloj, lo que debe hacerse en IF en lugar de en ID. Esto requerirá un incrementador adicional, ya que la ALU está ocupada en cada ciclo y no se puede utilizar para incrementar el PC.

2. En cada reloj se debe buscar una nueva instrucción —esto se hace también en IF.

3. En cada ciclo de reloj se necesita una nueva palabra de datos —esto se hace en MEM.

4. Debe haber un MDR separado para cargas (LMDR) y almacenamientos (SMDR), ya que cuando son instrucciones consecutivas se solapan en el tiempo.

5. Se necesitan tres cerrojos adicionales que contengan los valores que se necesitarán más tarde, pero que se puedan modificar con una instrucción posterior. Los valores almacenados son: la instrucción, la salida de la ALU y el siguiente PC.

Probablemente, el impacto más grande de la segmentación sobre los recursos de la máquina esté en el sistema de memoria. Aunque el tiempo de acceso a memoria no haya cambiado, el ancho de banda máxima *(peak)* de memoria se debe incrementar cinco veces con respecto a la máquina no segmentada porque, en la máquina, encauzada se requieren dos accesos a memoria en cada reloj frente a dos accesos cada cinco ciclos de reloj, en una máquina no segmentada, con el mismo número de pasos por instrucción. Para proporcionar dos accesos a memoria cada ciclo, la mayoría de las máquinas utilizarán caches separadas para instrucciones y datos (ver Cap. 8, Secc. 8.3).

Durante la etapa EX, la ALU puede utilizarse para tres funciones diferentes: un cálculo efectivo de una dirección de dato, un cálculo de una dirección de salto, o una operación de la ALU. Afortunadamente, las instrucciones DLX son simples; la instrucción en EX hace como máximo una de estas funciones; así no surgen conflictos.

La segmentación que tenemos ahora para DLX funcionaría bastante bien si cada instrucción fuera independiente de las demás instrucciones que se ejecutan al mismo tiempo. En realidad, las instrucciones pueden ser dependientes unas de otras; éste es el tema de la siguiente sección.

Etapa	Instrucción de la ALU	Carga o almacenamiento	Instrucción de salto
IF	IR←Mem[PC]; PC←PC+4;	IR←Mem[PC]; PC←PC+4;	IF←Mem[PC]; PC←PC+4;
ID	A←Rs1; B←Rs2; PC←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR
EX	ALUoutput←A *op* B; o ALUoutput←A *op* $((IR1_{16})^{16}\#\#IR1_{16..31})$;	DMAR←A+ $((IR1_{16})^{16}\#\#IR1_{16..31})$; SMDR←B;	ALUoutput←PC1 + $((IR1_{16})^{16}\#\#IR1_{16..31})$; cond←(Rs1 *op* 0);
MEM	ALUoutput1←ALUoutput	LMDR←Mem[DMAR] ; o Mem[DMAR]←SMDR;	if (cond) PC←ALUoutput;
WB	Rd←ALUoutput1;	Rd←LMDR;	

FIGURA 6.4 Eventos en cada etapa de la segmentación de DLX. Como la instrucción todavía no está codificada, las dos primeras etapas de la segmentación son siempre idénticas. Observar que era crítico poder buscar los registros antes de decodificar la instrucción; en cualquier otro caso se necesitaría una etapa adicional. Debido al formato de instrucción fija, los campos de los registros se decodifican siempre y se accede a los registros; el PC y los campos inmediatos se pueden enviar también a la ALU. Al comienzo de la operación de la ALU las entradas correctas son multiplexadas, según el código de operación. Con esta organización todas las operaciones dependientes de instrucciones se presentan en la etapa EX o más tarde. Igual que en la Figura 6.3, incluimos los casos de saltos, pero no de bifurcaciones, que tendrán un desplazamiento de 26 bits además de un desplazamiento de 16 bits.

6.4 El principal obstáculo de la segmentación: riesgos de la segmentación

Hay situaciones, llamadas *riesgos* (*hazards*), que impiden que se ejecute la siguiente instrucción del flujo de instrucciones durante su ciclo de reloj designado. Los riesgos reducen el rendimiento de la velocidad ideal lograda por la segmentación. Hay tres clases de riesgos:

1. *Riesgos estructurales* surgen de conflictos de los recursos, cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones en ejecuciones solapadas simultáneamente.
2. *Riesgos por dependencias de datos* surgen cuando una instrucción depende de los resultados de una instrucción anterior, de forma que, ambas, podrían llegar a ejecutarse de forma solapada.
3. *Riesgos de control* surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.

Los riesgos en la segmentación pueden hacer necesario detenerla. La diferencia principal entre detenciones en una máquina segmentada y en una no segmentada (como las que vimos en DLX en el Capítulo 5) se presenta porque hay múltiples instrucciones ejecutándose a la vez. Una detención en una máquina segmentada requiere, con frecuencia, que prosigan algunas instrucciones, mientras se retardan otras. Normalmente, cuando una instrucción está detenida, todas las instrucciones posteriores a esta instrucción también se detienen. Las instrucciones anteriores a la instrucción detenida pueden continuar, pero no se buscan instrucciones nuevas durante la detención. Veremos algunos ejemplos de cómo operan las detenciones en esta sección —¡no se preocupe, no son tan complejas como puede parecer!

Una detención hace que el rendimiento de la segmentación se degrade con relación al rendimiento ideal. Veamos una sencilla ecuación para encontrar la aceleración real de la segmentación, comenzando con la fórmula de la sección anterior.

$$\text{Aceleración de la segmentación} = \frac{\text{Tiempo medio de instrucción sin segmentación}}{\text{Tiempo medio de instrucción con segmentación}} =$$

$$= \frac{\text{CPI sin segmentación} \cdot \text{Ciclo de reloj sin segmentación}}{\text{CPI con segmentación} \cdot \text{Ciclo de reloj con segmentación}} =$$

$$= \frac{\text{Ciclo de reloj sin segmentación}}{\text{Ciclo de reloj con segmentación}} \cdot \frac{\text{CPI sin segmentación}}{\text{CPI con segmentación}}$$

Recordar que la segmentación se puede considerar como una disminución del CPI o de la duración del ciclo de reloj; tratémosla como una disminución del CPI. El CPI ideal en una máquina segmentada es habitualmente

$$\text{CPI Ideal} = \frac{\text{CPI sin segmentación}}{\text{Profundidad de la segmentación}}$$

Reorganizando esto y sustituyendo en la ecuación de la aceleración:

$$\text{Aceleración} = \frac{\text{Ciclo de reloj sin segmentación}}{\text{Ciclo de reloj con segmentación}} \cdot \frac{\text{CPI Ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI con segmentación}}$$

Si nos limitamos a las detenciones de la segmentación

$$\text{CPI con segmentación} = \text{CPI Ideal} + \text{Ciclo de reloj de detención de la segmentación por instrucción}$$

Podemos sustituir y obtener:

$$\text{Aceleración} = \frac{\text{Ciclo de reloj sin segmentación}}{\text{Ciclo de reloj con segmentación}} \cdot \frac{\text{CPI Ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI Ideal} + \text{Ciclos de detención de la segmentación}}$$

Aunque esto da una fórmula general para la velocidad de la segmentación (ignorando detenciones distintas a las de la segmentación), en muchas instancias se puede utilizar una ecuación más simple. Con frecuencia, preferimos ignorar el incremento potencial en la frecuencia de reloj debido al gasto de la segmentación. Esto hace iguales las frecuencias de reloj y nos permite despreciar el primer término. Ahora se puede utilizar una fórmula más simple:

$$\text{Aceleración de la segmentación} = \frac{\text{CPI Ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI Ideal} + \text{Ciclos de detención de la segmentación}}$$

Aunque utilicemos esta forma más simple para evaluar la segmentación de DLX, un diseñador debe tener cuidado de no descontar el impacto potencial sobre la frecuencia de reloj al evaluar las estrategias de segmentación.

Riesgos estructurales

Cuando se segmenta una máquina, la ejecución solapada de las instrucciones requiere la segmentación de unidades funcionales y duplicación de recursos para permitir todas las posibles combinaciones de instrucciones. Si alguna combinación de instrucciones no se puede acomodar debido a conflictos de

Instrucción	Número de ciclo de reloj								
	1	2	3	4	5	6	7	8	9
Instrucción de carga	IF	ID	EX	MEM	WB				
Instrucción $i + 1$		IF	ID	EX	MEM	WB			
Instrucción $i + 2$			IF	ID	EX	MEM	WB		
Instrucción $i + 3$				detención	IF	ID	EX	MEM	WB
Instrucción $i + 4$						IF	ID	EX	MEM

FIGURA 6.5 Detención durante un riesgo estructural —una carga con un puerto de memoria—. Con sólo un puerto de memoria, el procesador no puede iniciar una búsqueda de datos y una búsqueda de instrucción en el mismo ciclo. Una instrucción de carga roba efectivamente un ciclo de búsqueda de instrucción haciendo que la segmentación se detenga —ninguna instrucción se inicia en el ciclo de reloj 4 (que, normalmente, sería la instrucción $i + 3$). Como la instrucción que se está buscando está detenida, las demás instrucciones pueden proceder normalmente. El ciclo de detención continuará pasando a través de las diferentes etapas de la segmentación.

recursos, se dice que la máquina tiene un *riesgo estructural*. Las instancias más comunes de riesgos estructurales surgen cuando alguna unidad funcional no está completamente segmentada. Entonces, el procesador no puede iniciar, secuencialmente, ninguna secuencia de instrucciones, en las que todas utilicen esa unidad funcional. También aparecen riesgos estructurales cuando algunos recursos no se han duplicado lo suficiente, para que permitan la ejecución de todas las combinaciones de instrucciones. Por ejemplo, una máquina puede tener solamente un puerto de escritura en el fichero de registros, pero bajo ciertas circunstancias, puede haber necesidad de realizar dos escrituras en un ciclo de reloj. Esto generará un riesgo estructural. Cuando una secuencia de instrucciones encuentre este riesgo, el procesador detendrá una de las instrucciones hasta que la unidad requerida esté disponible.

Muchas máquinas segmentadas comparten un único puerto de memoria para datos e instrucciones. Como consecuencia, cuando una instrucción contenga una referencia a la memoria de datos, la segmentación debe detenerse durante un ciclo de reloj. La máquina no puede buscar la siguiente instrucción debido a que la referencia al dato está utilizando el puerto de memoria. La Figura 6.5 muestra que una segmentación con un puerto de memoria es análoga a una detención durante una carga. Veremos otro tipo de detenciones cuando hablemos de riesgos de datos.

Ejemplo

Supongamos que las referencias a datos constituyen el 30 por 100 de la mezcla y que el CPI ideal de la máquina segmentada, ignorando los riesgos estructurales, es 1,2. Sin considerar ninguna otra pérdida de rendimiento, ¿cuántas veces es más rápida la máquina ideal sin los riesgos estructurales de memoria, frente a la máquina con los riesgos?

Respuesta

La máquina ideal será más rápida según la relación de la aceleración de la máquina ideal sobre la máquina real. Como las frecuencias de reloj no están afectadas, podemos utilizar lo siguiente para la aceleración:

$$\text{Aceleración de la segmentación} = \frac{\text{CPI Ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI Ideal} + \text{Ciclos de detención de la segmentación}}$$

Como la máquina ideal no tiene detenciones, la aceleración es sencillamente

$$\frac{1{,}2 \cdot \text{Profundidad de la segmentación}}{1{,}2}$$

La aceleración de la máquina real es:

$$\frac{1{,}2 \cdot \text{Profundidad de la segmentación}}{1{,}2 + 0{,}3 \cdot 1} = \frac{1{,}2 \cdot \text{Profundidad de la segmentación}}{1{,}5}$$

$$\frac{\text{Aceleración}_{\text{ideal}}}{\text{Aceleración}_{\text{real}}} = \frac{\left(\frac{1{,}2 \cdot \text{Profundidad de la segmentación}}{1{,}2}\right)}{\left(\frac{1{,}2 \cdot \text{Profundidad de la segmentación}}{1{,}5}\right)} =$$

$$= \frac{1{,}5}{1{,}2} = 1{,}25$$

Por tanto, la máquina sin riesgos estructurales es el 25 por 100 más rápida.

Si los demás factores son iguales, una máquina sin riesgos estructurales tendrá siempre un CPI más bajo. Entonces, ¿por qué permite un diseñador riesgos estructurales? Hay dos razones: para reducir el coste y para reducir la latencia de la unidad. La segmentación de todas las unidades funcionales puede ser muy costosa. Máquinas que soportan referencias a memoria de un ciclo de reloj requieren un ancho de banda total de memoria de dos veces este valor y, con frecuencia, tienen mayor ancho de banda en las patillas. De la misma forma, para segmentar completamente un multiplicador de punto flotante se necesitan muchas puertas. Si los riesgos estructurales no se presentan con frecuencia, puede no merecer la pena el coste de evitarlos. También es posible, habitualmente, diseñar una unidad no segmentada, o que no esté segmentada completamente, con un retardo total menor que el de una unidad completamente segmentada. Por ejemplo, la unidad de punto flotante del CDC 7600 y del MIPS R2010 optaron por una latencia menor (menos ciclos por operación) en lugar de una segmentación completa. Como veremos pronto, reducir la latencia tiene otros beneficios de rendimiento y, frecuentemente, pueden superar las desventajas de los riesgos estructurales.

Ejemplo

Muchas máquinas recientes no tienen unidades de punto flotante completamente segmentadas. Por ejemplo, supongamos que tenemos una implementación de DLX con una latencia de 5 ciclos de reloj para multiplicar en punto flotante, pero sin segmentación. ¿Tendrá este riesgo estructural un impacto

grande o pequeño sobre el rendimiento durante la ejecución de Spice en DLX? Por simplicidad, suponer que las multiplicaciones de punto flotante están distribuidas uniformemente.

Respuesta Los datos de la Figura C.4 muestran que la multiplicación en punto flotante tiene una frecuencia del 5 por 100 en Spice. La segmentación propuesta por nosotros puede manejar hasta una frecuencia del 20 por 100 de multiplicaciones en punto flotante —una cada cinco ciclos de reloj. Esto significa que el beneficio del rendimiento de la multiplicación, en punto flotante completamente segmentada, probablemente es bajo mientras las multiplicaciones en punto flotante no estén apiñadas sino distribuidas uniformemente. Si estuviesen apiñadas, el impacto podría ser mucho mayor.

Riesgos por dependencias de datos

Un efecto importante de la segmentación es cambiar la temporización relativa de las instrucciones al solapar su ejecución. Esto introduce riesgos por dependencias de datos y de control. Los riesgos por dependencias de datos se presentan cuando el orden de acceso a los operandos los cambia la segmentación, con relación al orden normal que se sigue en las instrucciones que se ejecutan secuencialmente. Considerar la ejecución encauzada de las instrucciones:

```
ADD    R1,R2,R3
SUB    R4,R1,R5
```

La instrucción SUB tiene una fuente, R1, que es el destino de la instrucción ADD. Como muestra la Figura 6.6, la instrucción ADD escribe el valor de R1 en la etapa WB, pero la instrucción SUB lee el valor durante su etapa ID. Este problema se denomina *riesgo por dependencias de datos*. A menos que se tomen precauciones para prevenirlo, la instrucción SUB leerá el valor erróneo y tratará de utilizarlo. En efecto, el valor utilizado por la instrucción SUB no es aún determinístico: aunque nos pueda parecer lógico suponer que SUB utilizará siempre el valor de R1, que fue asignado por una instrucción anterior a ADD, esto no siempre es así. Si se presentase una interrupción entre las instrucciones ADD y SUB, se completaría la etapa WB de ADD, y el valor de R1 en ese punto **sería** el resultado de ADD. Este comportamiento impredecible es obviamente inaceptable

Instrucción	Ciclo de reloj					
	1	2	3	4	5	6
Instrucción ADD	IF	ID	EX	MEM	WB_dato escrito aquí	
Instrucción SUB		IF	ID_dato leído aquí	EX	MEM	WB

FIGURA 6.6 La instrucción ADD escribe en un registro que es un operando fuente para la instrucción SUB. Pero ADD no **termina** de escribir el dato en el fichero de registros hasta tres ciclos de reloj después que SUB **comienza** a leerlo.

El problema planteado en este ejemplo puede resolverse con una simple técnica hardware llamada *adelantamiento (forwarding)* (también llamada *desvío [bypassing]* y a veces *cortocircuito*). Esta técnica funciona como sigue: el resultado de la ALU siempre realimenta sus cerrojos de entrada. Si el hardware de adelantamiento detecta que la operación previa de la ALU ha escrito en un registro, correspondiente a una fuente para la operación actual de la ALU, la lógica de control selecciona el resultado adelantado como entrada de la ALU en lugar del valor leído en el fichero de registros. Observar que con el adelantamiento, si SUB es detenida, se completará ADD, y no se activará el desvío, haciendo que se utilice el valor del registro. Esto también es cierto para el caso de una interrupción entre las dos instrucciones.

En la segmentación de DLX, debemos pasar resultados no sólo a la instrucción inmediatamente siguiente, sino también a la instrucción después de ésa. Para la instrucción tres líneas más abajo, se solapan las etapas ID y WB; sin embargo, como la escritura no se termina hasta el fin de WB, debemos continuar adelantando el resultado. La Figura 6.7 muestra un repertorio de instrucciones y las operaciones de adelantamiento que se pueden presentar.

Es deseable reducir el número de instrucciones que deban ser desviadas, ya que cada nivel requiere hardware especial. Recordando que el fichero de registros es accedido dos veces en un ciclo de reloj, es posible hacer las escrituras en el registro en la primera mitad de WB y las lecturas en la segunda

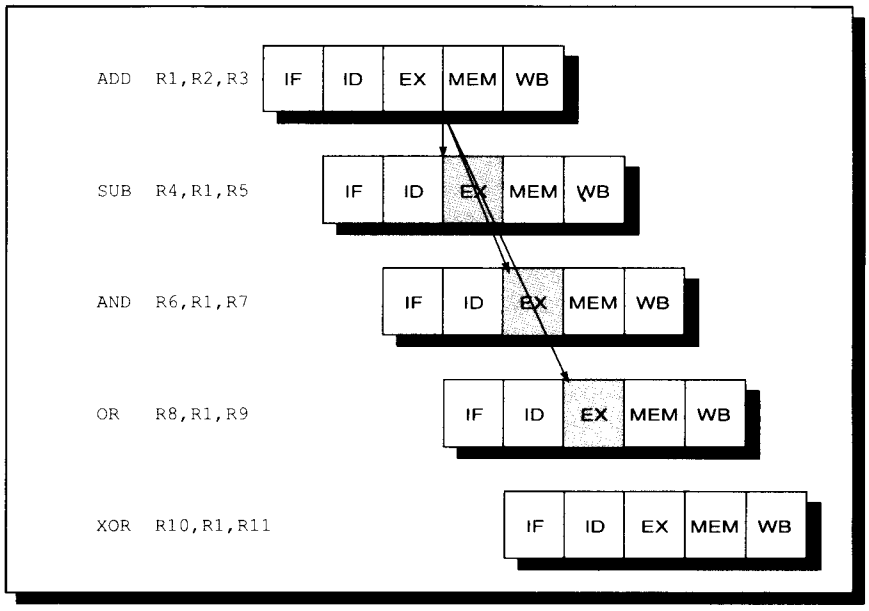


FIGURA 6.7 Un conjunto de instrucciones en el procesador segmentado que necesitan adelantar resultados. La instrucción ADD inicializa R1, y las cuatro instrucciones siguientes lo utilizan. El valor de R1 debe ser desviado para las instrucciones SUB, AND y OR. En el instante que la instrucción XOR va a leer R1 en la fase ID, la instrucción ADD ha completado WB, y el valor está disponible.

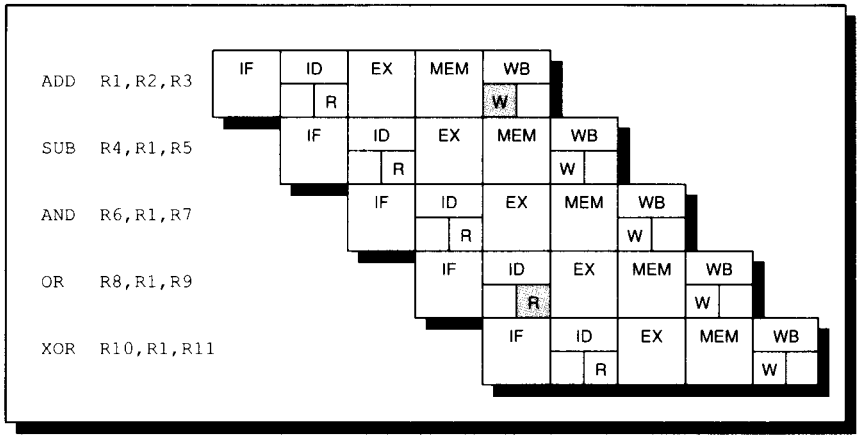


FIGURA 6.8 La misma secuencia de instrucciones que la de la Figura 6.7, con lecturas y escrituras de registros en las mitades opuestas de las etapas ID y WB. Las instrucciones SUB y AND necesitan todavía que el valor de R1 sea desviado, y esto ocurrirá cuando entren en su etapa EX. Sin embargo, durante la instrucción OR, que también usa R1, se ha completado la escritura de R1, y no se necesita adelantamiento. XOR depende de ADD, pero el valor de R1 de ADD se escribe siempre, en el ciclo, antes que XOR alcance su etapa ID y lo lea.

mitad de ID. Esto elimina la necesidad de desviarse para una tercera instrucción, como muestra la Figura 6.8.

Cada nivel de desvío requiere un cerrojo y un par de comparadores para examinar si, instrucciones adyacentes, comparten un destino y una fuente. La Figura 6.9 muestra la estructura de la ALU y su unidad de desvío, así como los valores que están en los registros de desvío para la secuencia de instrucciones de la Figura 6.7. Se necesitan dos buffers que contengan los resultados de la ALU que se van a almacenar en el registro destino en las dos siguientes etapas WB. Para operaciones de la ALU, el resultado se adelanta siempre que la instrucción que utiliza el resultado como fuente entra en su etapa EX. (La instrucción que calculó el valor que se va a adelantar, puede estar en sus etapas MEM o WB.) Los resultados de los buffers pueden ser entradas a uno de los puertos de la ALU, vía un par de multiplexores. El control de los multiplexores puede hacerse o por la unidad de control (que, entonces, debe rastrear los destinos y fuentes de todas las operaciones en curso) o localmente por la lógica asociada al desvío (en cuyo caso los buffers de desvío contendrán etiquetas que den los valores que estén destinados para los números de registro). En cualquiera de los eventos, la lógica debe examinar si alguna de las dos instrucciones anteriores escribió en un registro que sea entrada a la instrucción actual. Si es así, entonces el multiplexor selecciona el registro de resultado apropiado en lugar del bus. Debido a que la ALU opera en una sola etapa de la segmentación, no hay necesidad de ninguna detención por ninguna combinación de instrucciones de la ALU, una vez que se ha implementado el desvío.

Se crea un riesgo siempre que haya una dependencia entre instrucciones, y estén tan próximas que el solapamiento causado por la segmentación pueda cambiar el orden para acceder a un operando. Nuestros ejemplos de riesgos han sido con operandos de registro, pero también es posible crear, para un par de instrucciones, crear una dependencia escribiendo y leyendo la misma posición de memoria. Sin embargo, en la segmentación de DLX, las referencias a memoria se mantienen siempre en orden, evitando que surja este tipo de riesgos. Los fallos de la cache podrían desordenar las referencias a memoria si se permitiera que el procesador continuase trabajando con instrucciones posteriores, mientras estuviese accediendo a memoria una instrucción anterior que falló en la cache. En DLX detenemos la segmentación por completo, haciendo que la ejecución de la instrucción que causó el fallo se prolongue durante múltiples ciclos de reloj. En una sección avanzada de este capítulo, Sección 6.7, explicaremos máquinas que permiten que cargas y almacenamientos se ejecuten en un orden distinto del que tenían en el programa. Sin embargo, todos los riesgos por dependencia de datos explicados en esta sección, involucran registros de la CPU.

Se puede generalizar el adelantamiento para que incluya el paso de un resultado directamente a la unidad funcional que lo requiera: un resultado se adelanta desde la salida de una unidad a la entrada de otra, en lugar de permitir únicamente desde el resultado de una unidad a la entrada de la misma unidad. Tomar, por ejemplo, la siguiente secuencia:

```
ADD    R1,R2,R3
SW     25(R1),R1
```

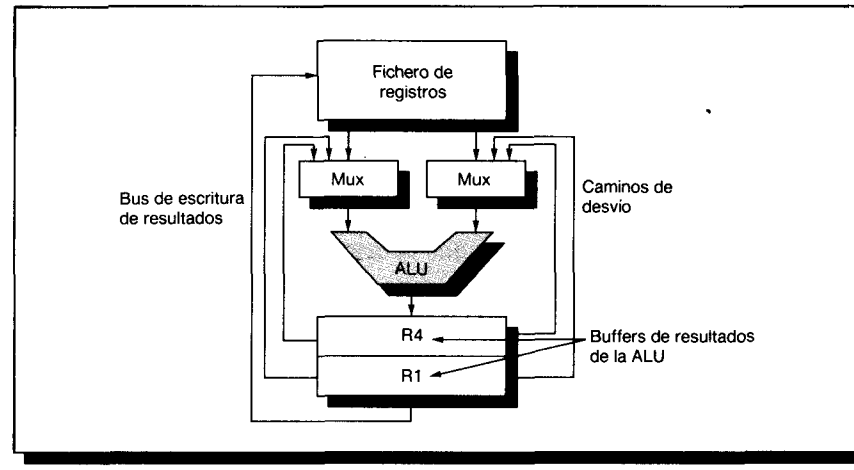


FIGURA 6.9 La ALU con su unidad de desvío. Los contenidos del buffer se muestran en el punto donde la instrucción AND de la secuencia de código de la Figura 6.8 está aproximadamente al comienzo de la etapa EX. La instrucción ADD que calculó R1 (en el segundo buffer) está en su etapa WB, y la entrada izquierda del multiplexor se inicializa para pasar el valor exactamente calculado de R1 (no el valor leído del fichero de registro) como primer operando de la instrucción AND. El resultado de la resta, R4, está en el primer buffer. Estos buffers corresponden a las variables `ALUoutput` y `ALUoutput1` de las Figuras 6.3 y 6.4.

Para prevenir una detención en esta secuencia, necesitaremos adelantar el valor de R1 desde la ALU a la ALU, para que se pueda utilizar en el cálculo de la dirección efectiva, y al MDR (registro de datos de memoria), para que se pueda almacenar sin ciclos de detención.

Los riesgos por dependencias de datos pueden clasificarse de tres formas distintas, dependiendo del orden de los accesos de lectura y escritura en las instrucciones. Por convenio, los riesgos se denominan por el orden en el programa que debe ser preservado por la segmentación. Considerar dos instrucciones *i* y *j*, presentándose *i* antes que *j*. Los posibles riesgos por dependencia de datos son:

- RAW (*lectura después de escritura-read after write*) —*j* trata de leer una fuente antes que la escriba *i*; así *j* toma incorrectamente el valor antiguo—. Este es el tipo más común de riesgos y es el que aparece en las Figuras 6.6 y 6.7.
- WAR (*escritura después de lectura-write after read*) —*j* intenta escribir un destino antes que sea leído por *i*; así *i* toma incorrectamente el nuevo valor—. Esto no puede ocurrir en nuestro ejemplo de segmentación porque todas las lecturas se hacen antes (en ID) y todas las escrituras después (en WB). Este riesgo se presenta cuando hay instrucciones que escriben anticipadamente los resultados en el curso de la instrucción, e instruciones que leen una fuente después que una instrucción posterior realice una escritura. Por ejemplo, autoincrementar el direccionamiento puede crear un riesgo WAR.
- WAW (*escritura después de escritura-write after write*) —*j* intenta escribir un operando antes de que sea escrito por *i*—. Las escrituras se están realizando en orden incorrecto, dejando en el destino el valor escrito por *i* en lugar del escrito por *j*. Este riesgo se presenta solamente en segmentaciones que escriben en más de una etapa (o permiten que proceda una instrucción aun cuando se encuentre detenida una instrucción anterior). La segmentación de DLX solamente escribe un registro en WB y evita esta clase de riesgos.

Observar que el caso RAR (*lectura después de lectura-read after read*) no es un riesgo.

No todos los riesgos por dependencias de datos se pueden manipular sin que tengan efecto en el rendimiento. Considerar la siguiente secuencia de instrucciones:

```
LW   R1,32(R6)
ADD  R4,R1,R7
SUB  R5,R1,R8
AND  R6,R1,R7
```

Este caso es diferente de la situación de las operaciones consecutivas de la ALU. La instrucción LW no tiene el dato hasta el final del ciclo MEM, mientras que la instrucción ADD necesita el dato al comienzo de ese ciclo de reloj. Por tanto, el riesgo de utilizar el resultado de una instrucción de carga no se puede eliminar completamente por hardware. **Podemos** adelantar el resultado a la ALU

LW R1,32(R6)	IF	ID	EX	**MEM**	WB
ADD R4,R1,R7		IF	ID	**EX**	MEM
SUB R5,R1,R8			IF	ID	**EX**
AND R6,R1,R7				IF	**ID**

FIGURA 6.10 Riesgos que se presentan cuando el resultado de una instrucción de carga lo utiliza la siguiente instrucción como operando fuente y se adelanta. El valor está disponible cuando lo devuelve de memoria al final del ciclo MEM de la instrucción de carga. Sin embargo, se necesita al comienzo de ese ciclo de reloj para ADD (la etapa EX de la suma). El valor de la carga se puede adelantar a la instrucción SUB y llegará a tiempo para esa instrucción (EX). AND puede sencillamente leer el valor durante ID, ya que lee los registros en la segunda mitad del ciclo y el valor se escribe en la primera mitad.

directamente desde el MDR, y para la instrucción SUB —que comienza dos ciclos de reloj después de la carga— el resultado llega a tiempo, como muestra la Figura 6.10. Sin embargo, para la instrucción ADD el resultado adelantado llega demasiado tarde —al final de un ciclo de reloj, aunque se necesita al comienzo.

La instrucción de carga tiene un retardo o latencia que no se puede eliminar sólo por adelantamiento; para hacer eso se requeriría que el tiempo de acceso al dato fuese cero. La solución más común a este problema es una adición de hardware denominada interbloqueo de la segmentación. En general, un *interbloqueo de la segmentación* detecta un riesgo y detiene la segmentación hasta que el riesgo desaparece. En este caso el interbloqueo detiene la segmentación comenzando a partir de la instrucción que quiere utilizar el dato hasta que lo produzca la instrucción correspondiente. Este ciclo de retardo, llamado *burbuja o detención del cauce* (*pipeline stall* or *bubble*), permite que el dato de carga llegue desde memoria; ahora puede ser adelantado por el hardware. El CPI para la instrucción detenida aumenta según la duración de la detención (un ciclo de reloj en este caso). La segmentación detenida se muestra en la Figura 6.11.

Cualquier instrucción	**IF**	**ID**	**EX**	**MEM**	**WB**				
LW R1,32(R6)		IF	ID	EX	**MEM**	WB			
ADD R4,R1,R7			IF	ID	*detención*	**EX**	MEM	WB	
SUB R5,R1,R8				IF	*detención*	**ID**	EX	MEM	WB
AND R6,R1,R7					*detención*	IF	ID	EX	MEM	WB

FIGURA 6.11 Efecto de la detención en la segmentación. Todas las instrucciones a partir de la instrucción que tiene la dependencia están retardadas. Con el retardo, el valor de la carga que está disponible al final de MEM ahora se puede adelantar al ciclo EX de la instrucción ADD. Debido a la detención, la instrucción SUB leerá ahora el valor de los registros durante su ciclo ID en lugar de tener que adelantarlo desde el MDR.

El proceso de permitir que una instrucción se desplace desde la etapa de decodificación de la instrucción (ID) a la de ejecución (EX) de este cauce, habitualmente, se denomina *emisión de la instrucción* (*instruction issue*); y una instrucción que haya realizado este paso se dice que ha sido *emitida* (*issued*). Para la segmentación de instrucciones sobre enteros de DLX, todos los riesgos de dependencias de datos se pueden comprobar durante la fase ID. Si existe un riesgo, la instrucción es detenida antes que sea emitida. Más tarde, en este capítulo, examinaremos situaciones donde la emisión de instrucciones es mucho más compleja. Detectar con antelación interbloqueos, reduce la complejidad del hardware porque el hardware nunca tiene que suspender ninguna instrucción que haya actualizado el estado de la máquina, a menos que la máquina completa esté detenida.

Ejemplo

Suponer que el 20 por 100 de las instrucciones son cargas, y que la mitad del tiempo la instrucción que sigue a una instrucción de carga depende del resultado de la carga. Si este riesgo crea un retardo de un solo ciclo, ¿cuántas veces es más rápida la máquina ideal segmentada (con un CPI de 1) que no retarda la segmentación, si se compara con una segmentación más realista? Ignorar cualquier otro tipo de detenciones.

Respuesta

La máquina ideal será más rápida según el cociente de los CPI. El CPI para una instrucción que sigue a una carga es 1,5, ya que se detiene la mitad del tiempo. Como las cargas son el 20 por 100 de la mezcla, el CPI efectivo es $(0{,}8 \cdot 1 + 0{,}2 \cdot 1{,}5) = 1{,}1$. Esto da una relación de rendimiento de $\frac{1{,}1}{1}$. Por consiguiente, la máquina ideal es un 10 por 100 más rápida.

Muchos tipos de detenciones son bastante frecuentes. El patrón normal de generación de código para una sentencia como A = B + C produce una detención para la carga del valor del segundo dato. La Figura 6.12 muestra que el almacenamiento no requiere otra detención, ya que el resultado de la suma se puede adelantar al MDR. Las máquinas, donde los operandos pueden provenir de memoria para operaciones aritméticas, necesitarán detener la segmentación en la mitad de la instrucción para esperar que se complete el acceso a memoria.

En lugar de permitir que se detenga la segmentación, el compilador puede intentar realizar una planificación que evite estas paradas, reorganizando la

`LW R1,B`	IF	ID	EX	MEM	WB				
`LW R2,C`		IF	ID	EX	MEM	WB			
`ADD R3,R1,R2`			IF	ID	*detención*	EX	MEM	WB	
`SW A,R3`				IF	*detención*	ID	EX	MEM	WB

FIGURA 6.12 Secuencia de código de DLX para A = B + C. La instrucción `ADD` debe ser detenida para permitir que se complete la carga de C. `SW` no necesita que se retarde más debido a que el adelantamiento hardware pasa directamente el resultado de la ALU al MDR para almacenarlo.

secuencia de código para eliminar el riesgo. Por ejemplo, el compilador trataría de evitar la generación de código con una instrucción de carga seguida por un uso inmediato del registro destino de la carga. Esta técnica, denominada *planificación de la segmentación* o *planificación de instrucciones*, se utilizó primero en los años sesenta, y llegó a ser un área de interés importante en los años ochenta, cuando estuvieron más extendidas las máquinas segmentadas.

Ejemplo

Generar código de DLX que evite detenciones del cauce para la siguiente secuencia:

```
a = b + c;
d = e - f;
```

Suponer que las cargas tienen una latencia de un ciclo de reloj.

Respuesta

Aquí está el código planificado:

```
LW   Rb,b
LW   Rc,c
LW   Re,e        ; intercambiada con la siguiente instrucción para evitar
ADD  Ra,Rb,Rc          detención
LW   Rf,f
SW   a,Ra        ; almacenamiento/carga intercambiado para evitar
SUB  Rd,Re,Rf          detención en SUB
SW   d,Rd
```

Ambos interbloqueos de la instrucción de carga (`LW Rc,c/ADD Ra,Rb,Rc` y `LW Rf,f/SUB Rd,Re,Rf`) han sido eliminados. Hay una dependencia entre la instrucción de la ALU y la de almacenamiento, pero la estructura de la segmentación permite que se adelante el resultado. Observar que el uso de registros diferentes para las sentencias primera y segunda fue crítico para que esta planificación fuese legal. En particular, si se cargase la variable `e` en los mismos registros que `b` o `c`, la planificación no sería legal. En general, la planificación de la segmentación puede incrementar el número de registros requeridos. En la Sección 6.8, veremos que este incremento puede ser sustancial para máquinas que pueden emitir múltiples instrucciones en un ciclo.

Esta técnica funciona tan bien que algunas máquinas responsabilizan al software de evitar este tipo de riesgos. Una carga que necesite que la instrucción siguiente no utilice su resultado se denomina *carga retardada* (*delayed load*). El espacio de la segmentación después de una carga se denomina con frecuencia *retardo de carga* (*load delay*) o *hueco de retardo* (*delay slot*). Cuando el compilador no puede planificar el interbloqueo, se puede insertar una instrucción de no operación. Esto no afecta al tiempo de ejecución, pero incrementa el espacio de código con respecto a una máquina con interbloqueo.

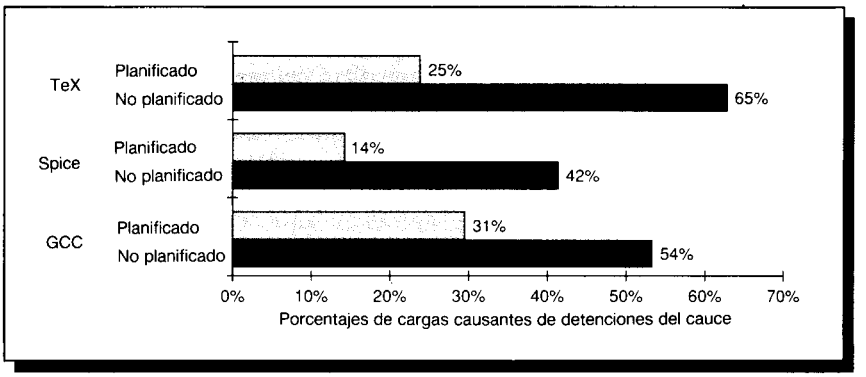


FIGURA 6.13 Porcentaje de las cargas que causan una detención con la segmentación de DLX. Las barras negras muestran la cantidad sin planificación por parte del compilador; las barras grises muestran el efecto de un buen, aunque sencillo, algoritmo de planificación. Estos datos muestran la eficacia de la planificación después de la optimización global (ver Capítulo 3, Sección 3.7). La optimización global, en realidad, hace la planificación relativamente más difícil porque hay menos candidatos disponibles para planificar en los huecos de retardo. Por ejemplo, en GCC y TeX, cuando se planifican los programas pero no se optimizan globalmente, el porcentaje de retardos de las cargas que causan una detención caen al 22 y 19 por 100, respectivamente.

Tanto si el hardware detecta este interbloqueo y detiene la segmentación como si no, el rendimiento mejorará si el compilador planifica instrucciones. Si se presenta una detención, el impacto sobre el rendimiento será el mismo, tanto si la máquina ejecuta un ciclo inactivo como si ejecuta una no operación. La Figura 6.13 muestra que la planificación puede eliminar la mayoría de estos retardos. En esta figura se ve que los retardos de carga en GCC son significativamente más difíciles de planificar que en Spice o TeX.

Implementación de la detección de riesgos por dependencias de datos en segmentaciones simples

La forma de cómo se implementen los interbloqueos de la segmentación depende mucho de la longitud y complejidad de la segmentación. Para una máquina compleja con instrucciones de larga ejecución e interdependencias multiciclo, puede ser necesaria una tabla central que refleje la disponibilidad de los operandos y de las escrituras pendientes (ver Sección 6.7). Para la segmentación de instrucciones sobre enteros de DLX, el único interbloqueo que necesitamos aplicar es una carga seguida de forma inmediata por su uso. Esto puede hacerse con un simple comparador que busque este patrón de fuente y destino de la carga. El hardware requerido para detectar y controlar los riesgos por dependencias de datos de la carga y adelantar el resultado de la carga es como sigue:

- Multiplexores adicionales en las entradas a la ALU (como se requería para el hardware de desvío para las instrucciones registro-registro).
- Caminos extra desde el MDR a las entradas de los multiplexores a la ALU.
- Un buffer para guardar los números del registro-destino de las dos instrucciones anteriores (igual que para el adelantamiento registro-registro).
- Cuatro comparadores para comparar los dos posibles campos del registro fuente con los campos destino de las instrucciones anteriores y buscar una coincidencia.

Los comparadores comprueban un interbloqueo de carga al comienzo del ciclo EX. Las cuatro posibilidades y las acciones requeridas se muestran en la Figura 6.14.

Para DLX, la detección de riesgos y el hardware de adelantamiento es razonablemente simple; veremos que las cosas son mucho más complicadas cuando las segmentaciones son muy profundas (Sección 6.6). Pero antes de que hagamos esto, veamos qué ocurre con los saltos de DLX.

Situación	Secuencia de código ejemplo	Acción

No dependencia	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		No posible riesgo porque no existe dependencia sobre R1 en las tres instrucciones inmediatamente siguientes

Dependencia que requiere detención	`LW R1,45(R2)`
	`ADD R5,R1,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		Los comparadores detectan el uso de R1 en ADD y detienen ADD (y SUB y OR) antes que ADD comience EX

Dependencia superada por adelantamiento	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R1,R7`
	`OR R9,R6,R7`
		Los comparadores detectan el uso de R1 en SUB y adelantan el resultado de la carga a la ALU en el instante en que SUB comienza EX.

Dependencia con accesos en orden	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R1,R7`
		No se requiere acción porque la lectura de R1 por OR se presenta en la segunda mitad de la fase ID, mientras que la escritura del dato cargado se presentó en la primera mitad. Ver Figura 6.8.

FIGURA 6.14 Situaciones que el hardware de detección de riesgos de la segmentación puede ver comparando los destinos y fuentes de instrucciones adyacentes. Esta tabla indica que la única comparación necesaria está entre los destinos y las fuentes de dos instrucciones que siguen a la instrucción que escribió el destino. En el caso de una detención, las dependencias del cauce se parecerán al tercer caso, una vez que continúa la ejecución.

Riesgos de control

Los *riesgos de control* pueden provocar mayor pérdida de rendimiento para la segmentación de DLX que los riesgos por dependencias de datos. Cuando se ejecuta un salto, puede o no cambiar el PC a algo diferente su valor actual más 4. (Recordar que si un salto cambia el PC a su dirección destino, el salto es efectivo; en caso contrario es no efectivo.) Si la instrucción *i* es un salto efectivo, entonces el PC, normalmente, no cambia hasta el final de MEM, después de que se complete el cálculo de la dirección y comparación, como muestra la Figura 6.4. Esto significa detención durante tres ciclos de reloj, al final de los cuales el nuevo PC es conocido y se puede buscar la instrucción adecuada. Este efecto se denomina un *riesgo de salto* o *de control*. La Figura 6.15 muestra una detención de tres ciclos para un riesgo de control.

El esquema de la Figura 6.15 no es posible porque no sabemos que la instrucción es un salto hasta después de la búsqueda de la siguiente instrucción. La Figura 6.16 soluciona esto repitiendo sencillamente la búsqueda una vez que se conoce el destino.

Instrucción de salto	IF	ID	EX	MEM	WB					
Instrucción $i + 1$		*detención*	*detención*	*detención*	IF	ID	EX	MEM	WB	
Instrucción $i + 2$			*detención*	*detención*	*detención*	IF	ID	EX	MEM	WB
Instrucción $i + 3$				*detención*	*detención*	*detención*	IF	ID	EX	MEM
Instrucción $i + 4$					*detención*	*detención*	*detención*	IF	ID	EX
Instrucción $i + 5$						*detención*	*detención*	*detención*	IF	ID
Instrucción $i + 6$							*detención*	*detención*	*detención*	IF

FIGURA 6.15 Detención ideal de DLX después de un riesgo de control. La instrucción etiquetada instrucción $i + k$ representa la instrucción *k*ésima ejecutada después del salto. Hay una dificultad, ya que la instrucción de salto no se decodifica hasta después que se ha buscado la instrucción $i + 1$. Esta figura muestra la dificultad conceptual, mientras que la Figura 6.16 lo que realmente ocurre.

Instrucción de salto	IF	ID	EX	MEM	WB					
Instrucción $i + 1$		IF	*detención*	*detención*	IF	ID	EX	MEM	WB	
Instrucción $i + 2$			*detención*	*detención*	*detención*	IF	ID	EX	MEM	WB
Instrucción $i + 3$				*detención*	*detención*	*detención*	IF	ID	EX	MEM
Instrucción $i + 4$					*detención*	*detención*	*detención*	IF	ID	EX
Instrucción $i + 5$						*detención*	*detención*	*detención*	IF	ID
Instrucción $i + 6$							*detención*	*detención*	*detención*	IF

FIGURA 6.16 Lo que realmente ocurre en la segmentación de DLX. Se busca la instrucción $i + 1$, aunque se ignora, y se vuelve a comenzar la búsqueda una vez que se conoce el destino del salto. Es probablemente obvio que si el salto no es efectivo, es redundante la segunda instrucción IF para la instrucción $i + 1$. Esto se tratará en breve.

Tres ciclos de reloj empleados en cada salto es una pérdida significativa. Con una frecuencia de saltos de un 30 por 100 y un CPI ideal de 1, la máquina con detenciones de saltos logra aproximadamente la mitad de la velocidad ideal de la segmentación. Por ello, reducir la penalización de los saltos llega a ser crítico. El número de ciclos de reloj de una detención de salto puede reducirse en dos pasos:

1. Averiguar si el salto es efectivo o no anteriormente en la segmentación.
2. Calcular anteriormente el PC efectivo (dirección destino del salto).

Para optimizar el comportamiento del salto deben realizarse los dos pasos anteriores —no ayuda conocer el destino del salto sin saber si la siguiente instrucción a ejecutar es el destino o la instrucción del PC + 4. Ambos pasos se deben realizar lo antes posible en la segmentación.

En DLX, los saltos (BEQZ y BNEZ) requieren examinar solamente la igualdad a cero. Por tanto, es posible completar esta decisión al final del ciclo ID, utilizando lógica especial dedicada a este test. Para aprovechar una decisión pronta sobre si el salto es efectivo, ambos PC (del salto efectivo y el del salto no efectivo) se deben calcular lo antes posible. El cálculo de la dirección del destino del salto requiere un sumador separado, que pueda sumar durante ID. Con el sumador separado y una decisión de salto tomada durante ID, solamente hay una detención de un ciclo de reloj en los saltos. La Figura 6.17 muestra la porción de salto de la tabla de ubicación de recursos revisada de la Figura 6.4.

En algunas máquinas, los riesgos de los saltos son aún más caros en ciclos de reloj que en nuestro ejemplo, ya que el tiempo para evaluar la condición de salto y calcular el destino puede ser aún mayor. Por ejemplo, una máquina con etapas separadas de decodificación y búsqueda de registros probablemente tendrá un *retardo de salto* —la duración del riesgo de control— que, como mínimo, es un ciclo de reloj mayor. El retardo de salto, a menos que sea tratado, puede ser una penalización del salto. Muchas VAX tienen retardos de salto de cuatro ciclos de reloj como mínimo; y máquinas segmentadas profundamente, con frecuencia, tienen penalizaciones de salto de seis a siete. En general, a mayor profundidad de la segmentación, peor penalización de salto en ciclos de reloj. Por supuesto, el efecto del rendimiento relativo de una mayor penalización de saltos depende del CPI global de la máquina. Una máquina con CPI alto puede soportar saltos más caros porque el porcentaje del rendimiento de la máquina que se perdería por los saltos sería menor.

Antes de hablar sobre métodos para reducir las penalizaciones de la segmentación que pueden surgir con los saltos, examinemos brevemente el comportamiento dinámico de los saltos.

Comportamiento de los saltos en los programas

Como los saltos pueden afectar enormemente al rendimiento del cauce, examinaremos su comportamiento para obtener algunas ideas sobre cómo se pueden reducir las penalizaciones de los saltos y bifurcaciones. Ya conocemos

Etapa del cauce	Instrucción de salto
IF	IR←Mem[PC]; PC←PC+4;
ID	A←Rs1; B←Rs2; **BTA←PC+((IR_{16})^{16}## $IR_{16..31}$)** **if (Rs1 *op* 0) PC←BTA**
EX	
MEM	
WB	

FIGURA 6.17 Estructura revisada de la segmentación (ver Fig. 6.4) mostrando el uso de un sumador separado para calcular la dirección destino del salto. Las operaciones que son nuevas o han cambiado están en negrita. Debido a que la suma de la dirección destino del salto (BTA) ocurre durante ID, ésta se llevará a cabo para todas las instrucciones; la condición de salto (Rs1 *op* 0) también se hará para todas las instrucciones. La última operación de ID es sustituir el PC. Se deberá saber que la instrucción es un salto antes de realizar este paso. Esto requiere la decodificación de la instrucción antes del final de ID, o hacer esta operación al comienzo de EX cuando se envíe el PC. Debido a que el salto se hace al final de ID, las etapas EX, MEM y WB no se usan para los saltos. Surge una complicación adicional para las bifurcaciones que tienen un desplazamiento mayor que los saltos. Esto se puede resolver utilizando un sumador adicional que sume el PC y los 26 bits inferiores del IR. Alternativamente se podría intentar un esquema más inteligente que haga una suma de 16 bits en la primera mitad del ciclo y determine si sumar 10 bits del IR en la segunda mitad del ciclo, decodificando antes los códigos de operación de bifurcación.

las frecuencias de los saltos para nuestros programas del Capítulo 4. La Figura 6.18 revisa la frecuencia global de las operaciones de flujo de control para tres de las máquinas y da el desglose entre saltos y bifurcaciones.

Todas las máquinas muestran una frecuencia de saltos condicionales del 11 al 17 por 100, mientras que la frecuencia de saltos incondicionales varía entre el 2 y el 8 por 100. Una pregunta obvia es, ¿cuántos saltos son efectivos? Conocer el desglose entre saltos efectivos y no efectivos es importante porque afectará a las estrategias para reducir penalizaciones de saltos. Para el VAX, Clark y Levy [1984] midieron qué saltos condicionales simples eran efectivos con una frecuencia de aproximadamente el 50 por 100. Otros saltos, que se presentan con menos frecuencia, tienen diferentes frecuencias. La mayoría de los saltos de test de bits no son efectivos, y los saltos de los bucles son efectivos con una probabilidad del 90 por 100.

Para DLX, medíamos el comportamiento de los saltos en el Capítulo 3 y los resumíamos en la Figura 3.22. Los datos mostraban que eran efectivos el 53 por 100 de los saltos condicionales. Finalmente, el 75 por 100 de los saltos ejecutados eran saltos hacia adelante. Con estos datos en mente, veamos formas de reducir las penalizaciones de los saltos.

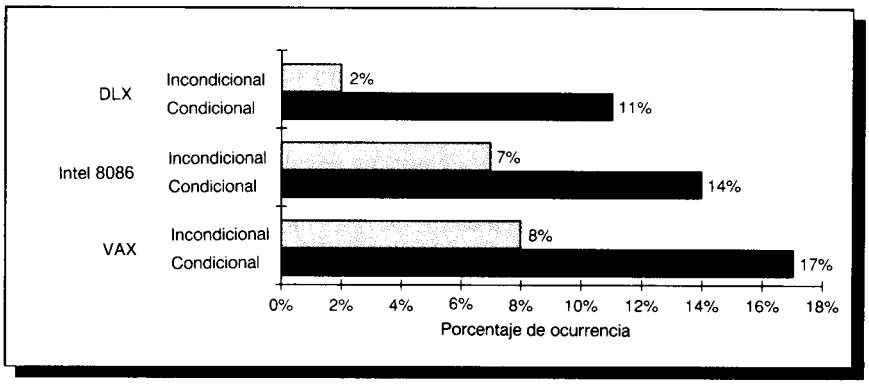


FIGURA 6.18 La frecuencia de instrucciones (saltos, bifurcaciones, llamadas y retornos) que puede cambiar el PC. Estos datos representan la media sobre los programas medidos en el Capítulo 4. Las instrucciones se dividen en dos clases: saltos, que son condicionales (incluyendo saltos de bucle), y los que son incondicionales (bifurcaciones, llamadas y retornos). Se omite el 360 porque no tiene separados los saltos incondicionales ordinarios de los condicionales. Emer y Clark [1984] informaron que el 38 por 100 de las instrucciones ejecutadas en sus medidas del VAX fueron instrucciones que podían cambiar el PC. Midieron que el 67 por 100 de esas instrucciones realmente provocan un salto en el flujo de control. Sus datos se tomaron de una carga de trabajo en tiempo compartido y reflejan muchos usos; sus medidas de frecuencias de saltos son mucho más altas que la de este diagrama.

Reducción de las penalizaciones de los saltos en la segmentación

Hay varios métodos para tratar con las detenciones de la segmentación debidas al retardo de los saltos, y en esta sección se explican cuatro sencillos esquemas en tiempo de compilación. En estos esquemas, las predicciones son estáticas —son fijas para cada salto durante la ejecución completa, y las predicciones son estimaciones en tiempo de compilación. En la Sección 6.7 se explican esquemas más ambiciosos que utilizan hardware para predecir dinámicamente los saltos.

El esquema más fácil es congelar la segmentación, reteniendo todas las instrucciones después del salto hasta que se conozca el destino del salto. Lo atractivo de esta solución radica, principalmente, en su simplicidad. Esta es la solución usada anteriormente en la segmentación de las Figuras 6.15 y 6.16.

Un esquema mejor y sólo ligeramente más complejo es predecir el salto como no efectivo, permitiendo, sencillamente, que el hardware continúe como si el salto no se ejecutase. Aquí hay que tener cuidado de no cambiar el estado de la máquina hasta que no se conozca definitivamente el resultado del salto. La dificultad que surge de esto —es decir, saber cuándo una instrucción puede cambiar el estado y cómo «deshacer» un cambio— puede hacernos reconsiderar la solución más simple de limpiar *(flushing)* la segmentación. En la segmentación de DLX, este esquema de predecir-no-efectivo *(predict-not-taken)*

se implementa continuando la búsqueda de las instrucciones, como si no ocurriese nada extraordinario. Sin embargo, si el salto es efectivo, necesitamos detener la segmentación y recomenzar la búsqueda. La Figura 6.19 muestra ambas situaciones.

Un esquema alternativo es predecir el salto como efectivo. Una vez que se decodifica el salto y se calcula la dirección destino, suponemos que el salto se va a realizar y comienza la búsqueda y ejecución en el destino. Como en la segmentación de DLX no se conoce la dirección del destino antes de que se conozca el resultado del salto, no hay ventaja en esta aproximación. Sin embargo, en algunas máquinas —especialmente las que tienen códigos de condición o condiciones de salto más potentes (y por consiguiente más lentas)— el destino del salto se conoce antes de que el resultado del salto, y este esquema tiene sentido.

Algunas máquinas han utilizado otra técnica denominada salto retardado, que se ha utilizado en muchas unidades de control microprogramadas. En un *salto retardado*, el ciclo de ejecución con un retardo de salto de longitud n es:

```
instrucción de salto
succesor secuencial1
succesor secuencial2
........
succesor secuencialn
destino de salto si efectivo
```

Instrucción de salto no efectivo	IF	**ID**	EX	MEM	WB				
Instrucción $i + 1$		**IF**	ID	EX	MEM	WB			
Instrucción $i + 2$			IF	ID	EX	MEM	WB		
Instrucción $i + 3$				IF	ID	EX	MEM	WB	
Instrucción $i + 4$					IF	ID	EX	MEM	WB

Instrucción de salto efectivo	IF	**ID**	EX	MEM	WB				
Instrucción $i + 1$		**IF**	**IF**	ID	EX	MEM	WB		
Instrucción $i + 2$			*detención*	IF	ID	EX	MEM	WB	
Instrucción $i + 3$				*detención*	IF	ID	EX	MEM	WB
Instrucción $i + 4$					*detención*	IF	ID	EX	MEM

FIGURA 6.19 El esquema de predecir-no-efectivo y la secuencia de la segmentación cuando no es efectivo el salto (en la parte superior) y cuando lo es (en la parte inferior). Cuando el salto no es efectivo, determinado durante ID, hemos buscado la siguiente instrucción y, simplemente, continuamos. Si el salto es efectivo durante ID, recomenzamos la búsqueda en el destino del salto. Esto hace que todas las instrucciones que siguen al salto se detengan un ciclo de reloj.

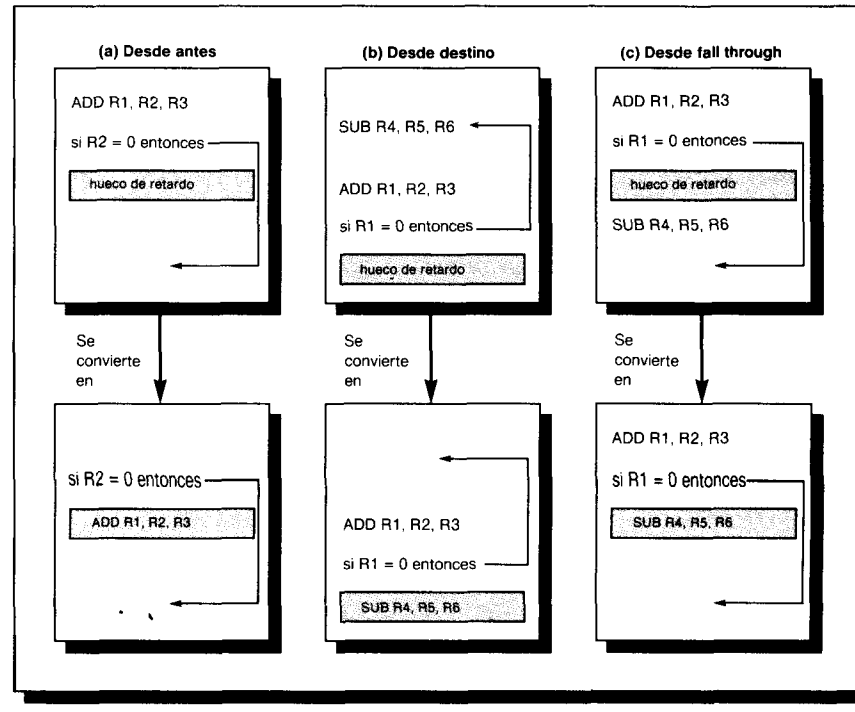


FIGURA 6.20 Planificación del hueco de retardo de los saltos. El cuadro superior de cada pareja muestra el código antes de la planificación y el cuadro inferior muestra el código planificado. En (a) el hueco de retardo se planifica con una instrucción independiente anterior al salto. Esta es la mejor elección. Las estrategias (b) y (c) se utilizan cuando no es posible (a). En las secuencias de código para (b) y (c), el uso de `R1` en la condición de salto impide que la instrucción `ADD` (cuyo destino es `R1`) sea transferida después del salto. En (b) el hueco de retardo de salto se planifica con el destino de salto; habitualmente, la instrucción del destino necesitará ser copiada porque puede ser alcanzada por otro camino. La estrategia (b) se prefiere cuando el salto se realiza con alta probabilidad, como, por ejemplo, un salto de bucle. Finalmente, el salto puede ser planificado con la instrucción siguiente, secuencialmente, como en (c). Para hacer esta optimización legal para (b) o (c), debe ser «OK» ejecutar la instrucción `SUB` cuando el salto vaya en la dirección no esperada. Por «OK» significamos que el trabajo se desperdicia, pero el programa todavía se ejecutará correctamente. Este es el caso, por ejemplo, si `R4` fuese un registro temporal no utilizado cuando el salto vaya en la dirección no esperada.

Los sucesores secuenciales están en *huecos de retardo del salto* (*branch-delay slots*). Como con los huecos de retardo de carga, la tarea del software es hacer las instrucciones sucesoras válidas y útiles. Se utiliza una serie de optimizaciones. La Figura 6.20 muestra las tres formas en las que se puede planificar el retardo de salto. La Figura 6.21 muestra las diferentes restricciones para cada uno de estos esquemas de planificación de saltos, así como las situaciones en las que se comportan mejor.

Estrategia de planificación	Requerimientos	¿Cuándo mejora el rendimiento?
(a) Desde antes del salto	Los saltos deben no depender de las instrucciones replanificadas	Siempre
(b) Desde el destino	Debe ser «OK» ejecutar las instrucciones replanificadas si no es efectivo el salto. Puede ser necesario duplicar instrucciones.	Cuando es efectivo el salto. Puede alargar el programa si las instrucciones están duplicadas
(c) Desde instrucciones siguientes	Debe ser «OK» ejecutar las instrucciones si es efectivo el salto	Cuando el salto no es efectivo

FIGURA 6.21 Esquemas de planificación de salto retardado y sus requerimientos. El origen de la instrucción que se planifica en el hueco de retardo determina la estrategia de planificación. El compilador debe hacer cumplir los requerimientos cuando busque instrucciones para planificar el retardo. Cuando los huecos no se puedan planificar, se rellenan con instrucciones de no operación. En la estrategia (b), si el destino del salto también es accesible desde otro punto del programa —como ocurriría si fuese la cabeza de un bucle— las instrucciones del destino deben ser copiadas y no transferidas.

Las limitaciones principales en la planificación de saltos retardados surgen de las restricciones sobre las instrucciones que se planifican en los huecos de retardo y de nuestra posibilidad para predecir en tiempo de compilación la probabilidad con que un salto va a ser efectivo. La Figura 6.22 muestra la efectividad de la planificación de saltos en DLX con un simple hueco de retardo de salto, utilizando un sencillo algoritmo de planificación de saltos. El algoritmo muestra que se llenan un poco más de la mitad de los huecos de retardo de salto, y la mayoría de los huecos llenos hacen un trabajo útil. Un promedio del 80 por 100, de los huecos de retardo llenos, contribuyen a la computación. Este número parece sorprendente, ya que los saltos solamente son efectivos aproximadamente el 53 por 100 de las veces. La frecuencia de éxitos es alta porque, aproximadamente, la mitad de los retardos de salto se rellenan con una instrucción anterior al salto (estrategia (a)), que es útil independientemente de que sea o no sea efectivo el salto.

Cuando el planificador de la Figura 6.22 no pueda utilizar la estrategia (a) —transferir una instrucción anterior al salto para rellenar el hueco de retardo del salto— solo utiliza la estrategia (b) —moviendo una instrucción del destino. (Por razones de simplicidad, el planificador no utiliza la estrategia (c).) En total, aproximadamente, la mitad de los huecos de retardo de salto son útiles dinámicamente, eliminando la mitad de las detenciones de saltos. Observando la Figura 6.22, vemos que la limitación principal es el número de huecos vacíos —los rellenos con instrucciones de no operación. Es improbable que la relación de huecos útiles a huecos llenos, aproximadamente el 80 por 100, pueda ser mejorada, ya que se requeriría una precisión mucho mejor en la predicción de saltos. En los ejercicios consideramos una extensión de la idea de salto retardado que trata de rellenar más huecos.

Hay un pequeño coste adicional de hardware para los saltos retardados. Debido al efecto retardado de los saltos, se necesitan múltiples PC (uno más que la duración del retardo) para restaurar correctamente el estado cuando se presente una interrupción. Considerar que la interrupción se presenta des-

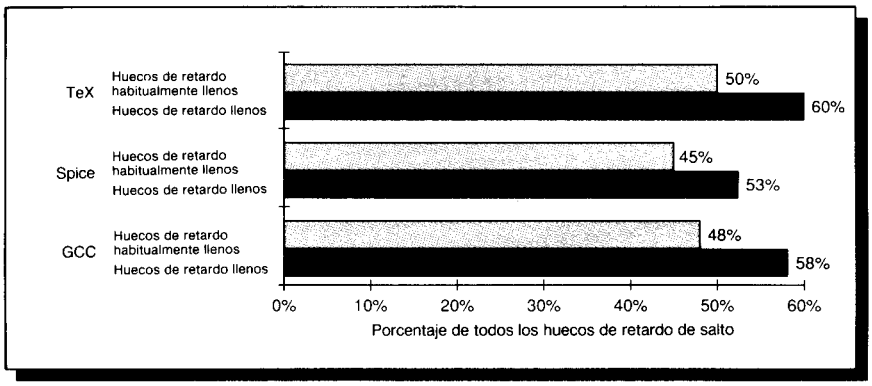


FIGURA 6.22 Frecuencia con que se rellena un único hueco de retardo de salto y frecuencia con que es útil la instrucción para la computación. La barra sólida muestra el porcentaje de los huecos de retardo de salto ocupados por alguna instrucción distinta de una no-operación. La diferencia entre el 100 por 100 y la columna oscura representa aquellos saltos que son seguidos por una no-operación. La barra sombreada muestra la frecuencia con que estas instrucciones hacen trabajo útil. La diferencia entre las barras sombreada y sólida es el porcentaje de instrucciones ejecutadas en un retardo de salto, pero que no contribuyen al cálculo. Estas instrucciones se presentan porque la optimización (b) es sólo útil cuando es efectivo el salto. Si se utilizase la optimización (c) también contribuiría a esta diferencia, ya que sólo es útil cuando no es efectivo el salto.

pués, que se completa una instrucción de salto efectivo, pero antes de que se completen todas las instrucciones de los huecos de retardo y del destino del salto. En este caso, los PC de los huecos de retardo y el PC del destino del salto se deben guardar, ya que no son secuenciales.

¿Cuál es el rendimiento efectivo de cada uno de estos esquemas? La aceleración efectiva de la segmentación con penalizaciones de salto es

$$\text{Aceleración de la segmentación} = \frac{\text{CPI Ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI Ideal} + \text{Ciclos de detención de la segmentación}}$$

Si suponemos que el CPI ideal es 1, entonces podemos simplificar esto:

$$\text{Aceleración de la segmentación} = \frac{\text{Profundidad de la segmentación}}{1 + \text{Ciclos de detención de la segmentación debidos a saltos}}$$

Ya que:

$$\text{Ciclos de detención de la segmentación debidos a saltos} =$$
$$= \text{Frecuencia de saltos} \cdot \text{Penalización de salto}$$

Esquema de planificación	Penalización de salto	CPI efectivo	Aceleración de la máquina segmentada respecto a la no segmentada	Aceleración respecto a la estrategia de detenerse en los saltos
Detención	3	1,42	3,5	1,0
Predicción «efectivo»	1	1,14	4,4	1,26
Predicción «no efectivo»	1	1,09	4,5	1,29
Salto retardado	0,5	1,07	4,6	1,31

FIGURA 6.23 Costes globales de una serie de esquemas de salto con la segmentación de DLX. Estos datos son para DLX utilizando la frecuencia medida de instrucciones de control del 14 por 100 y las medidas con que se rellena el hueco de retardo de la Figura 6.22. Además, sabemos que el 65 por 100 de las instrucciones de control realmente cambia el PC (saltos efectivos más cambios incondicionales). Se muestra el CPI resultante y la aceleración sobre una máquina no encauzada, que suponemos tendría una CPI de 5 sin ninguna penalización de saltos. La última columna de la tabla da la velocidad sobre un esquema que siempre se detiene en los saltos.

obtenemos:

$$\text{Aceleración de la segmentación} = \frac{\text{Profundidad de la segmentación}}{(1 + \text{Frecuencia de saltos} \cdot \text{Penalización de salto})}$$

Utilizando las medidas de DLX de esta sección, la Figura 6.23 muestra varias opciones hardware para tratar con los saltos, junto con sus rendimientos (suponiendo un CPI base de 1).

Recordar que los números de esta sección están **enormemente** afectados por la duración del retardo de salto y el CPI base. Un retardo de salto mayor provocará un incremento en la penalización y un porcentaje mayor de tiempo desperdiciado. Un retardo de solo un ciclo de reloj es pequeño —muchas máquinas tienen retardos mínimos de cinco o más ciclos—. Con un CPI bajo, el retardo debe mantenerse pequeño, mientras que un CPI base mayor reduciría la penalización relativa de los saltos.

Resumen: rendimiento de la segmentación para enteros

Cerramos esta sección sobre detección y eliminación de riesgos mostrando la distribución total de los ciclos inactivos de reloj para nuestros benchmarks cuando ejecutamos en el procesador entero de DLX software para planificación de la segmentación. La Figura 6.24 muestra la distribución de ciclos de reloj perdidos para retardos de carga y retardos de salto en nuestros tres pro-

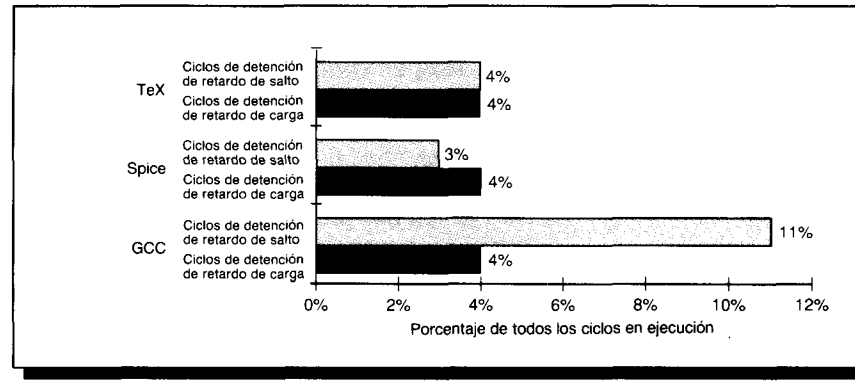


FIGURA 6.24 Porcentaje de los ciclos de reloj empleados en los retardos frente a ejecución de instrucciones. Esto supone un sistema perfecto de memoria; el recuento de ciclos de reloj y el recuento de instrucciones serían idénticos si no hubiese detenciones en el procesador de enteros segmentado. Este gráfico indica que del 7 al 15 por 100 de los ciclos de reloj son detenciones. El porcentaje restante del 85 al 93 por 100 son ciclos de reloj que emiten instrucciones. Los ciclos de reloj de Spice no incluyen detenciones en el procesador de FP, que se mostrará al final de la Sección 6.6. El planificador de la segmentación rellena retardos de cargas antes que retardos de saltos y esto afecta la distribución de los ciclos de retardo.

gramas, al combinar las medidas separadas mostradas en las Figuras 6.13 y 6.22.

Para los programas GCC y TeX, el CPI efectivo (ignorando cualquier detención, excepto las de los riesgos de la segmentación) en esta versión segmentada de DLX es 1.1. Comparar esto con el CPI para la versión cableada, no segmentada de DLX descrita en el Capítulo 5 (Sección 5.7), que es 5.8. Ignorando todas las demás fuentes de detenciones y suponiendo que las frecuencias de reloj son las mismas, la mejora del rendimiento de la segmentación es de 5.3 veces.

6.5 Qué hace difícil de implementar la segmentación

Ahora que comprendemos cómo detectar y resolver riesgos, podemos tratar algunas complicaciones que hemos evitado hasta ahora. En el Capítulo 5, veíamos que las interrupciones están entre los aspectos más difíciles de implementación de una máquina; la segmentación incrementa esa dificultad. En la segunda parte de esta sección, discutimos algunos de los retos planteados por diferentes repertorios de instrucciones.

Tratando con interrupciones

Las interrupciones son más difíciles de manejar en una máquina segmentada, porque el solapamiento de las instrucciones hace más difícil saber si una ins-

trucción puede cambiar, sin peligro, el estado de la máquina. En una máquina segmentada, una instrucción es ejecutada parte por parte y tarda varios ciclos de reloj en completarse. Incluso en el proceso de ejecución puede ser necesario actualizar el estado de la máquina. Entre tanto, una interrupción puede forzar la máquina a abortar la ejecución de la instrucción antes que se complete ésta.

Igual que en las implementaciones no segmentadas, las interrupciones más difíciles tienen dos propiedades: (1) se presentan durante las instrucciones, y (2) deben ser recomenzables. En nuestro procesador segmentado DLX, por ejemplo, un fallo de página de memoria virtual resultante de una búsqueda de datos no puede presentarse hasta el ciclo MEM de la instrucción. En el instante que se detecta el fallo, se estarán ejecutando otras instrucciones. Como un fallo de página debe ser recomenzable y requiere la intervención de otro proceso, tal como el sistema operativo, se debe detener las instrucciones en curso y guardar el estado para que la instrucción se pueda reiniciar en el estado correcto. Esto, habitualmente, se implementa guardando el PC de la instrucción para recomenzarla. Si la instrucción recomenzada no es un salto, entonces continuará la búsqueda de los sucesores secuenciales y comenzará su ejecución de forma normal. Si la instrucción reiniciada es un salto, entonces se evaluará la condición de salto y comenzará la búsqueda a partir del destino o a partir del camino secuencial. Cuando se presente una interrupción, se pueden realizar los siguientes pasos para guardar con seguridad el estado del procesador:

1. Forzar una instrucción de trap en el siguiente IF.
2. Hasta que el trap sea efectivo, eliminar todas las escrituras para la instrucción que causó el fallo y para las siguientes instrucciones. Esto previene cualquier cambio de estado para las instrucciones que no se hayan completado antes que sea tratada la interrupción.
3. Después que la rutina de tratamiento de interrupciones del sistema operativo reciba el control, se guarda inmediatamente el PC de la instrucción que causó el fallo. Este valor se utilizará a la vuelta de la interrupción.

Cuando se utilizan saltos retardados, no es posible restablecer el estado de la máquina con un solo PC, porque las instrucciones en curso pueden no estar relacionadas secuencialmente. En particular, cuando la instrucción que provoca la interrupción es un hueco de retardo de salto, y el salto fue efectivo, entonces las instrucciones para recomenzar son las del hueco más la instrucción del destino del salto. El salto ha completado la ejecución y no se reinicia. Las direcciones de las instrucciones del hueco del retardo del salto y del destino no son secuenciales. Por ello, será necesario guardar y restaurar un número de PC igual a la longitud del retardo de salto más uno. Esto se hace en el tercero de los pasos anteriores.

Una vez que ha sido tratada la interrupción, instrucciones especiales devuelven a la máquina al estado anterior a la interrupción recargando los PC y reiniciando el flujo de instrucciones (utilizando RFE en DLX). Si la segmentación se puede parar para que se completen las instrucciones anteriores a la del fallo y las posteriores se puedan reiniciar desde el principio, el procesador

segmentado dice que tiene *interrupciones precisas*. Idealmente, la instrucción del fallo no debería cambiar el estado, y, tratar correctamente algunas interrupciones, requiere que la instrucción del fallo no tenga efectos. Para otras interrupciones, como las excepciones de punto flotante, la instrucción del fallo en algunas máquinas escribe su resultado antes que pueda ser tratada la interrupción. En estos casos, el hardware debe estar preparado para recuperar los operandos fuente, aunque el operando destino sea idéntico a alguno de los operandos fuente.

Soportar interrupciones precisas es un requerimiento de muchos sistemas, mientras que, en otros, es valioso porque simplifica la interfaz del sistema operativo. Como mínimo, cualquier máquina con paginación bajo demanda o con manipuladores de traps aritméticos del IEEE deben hacer sus interrupciones precisas, bien por hardware o con algún soporte software.

Las interrupciones precisas son un reto debido a los mismos problemas que hacen difícil de recomenzar las instrucciones. Como vimos en el último capítulo, recomenzar es complicado por el hecho de que las instrucciones pueden cambiar el estado de la máquina antes de que se **garantice** que se completen (a veces se denominan instrucciones *comprometidas [committed]*). Como las instrucciones en curso pueden tener dependencias, no actualizar el estado de la máquina no es práctico si el procesador se va a mantener en pleno funcionamiento. Por tanto, cuando una máquina está más intensamente segmentada, llega a ser necesario poder dar marcha atrás en cualquier cambio de estado realizado antes que la instrucción estuviese comprometida (como se explicó en el Capítulo 5). Afortunadamente, DLX no tiene estas instrucciones, para la segmentación que estamos utilizando.

La Figura 6.25 muestra las etapas de la segmentación de DLX y las interrupciones «problema» que se pueden presentar en cada etapa. Como en la segmentación, hay múltiples instrucciones en ejecución, se pueden presentar múltiples interrupciones en el mismo ciclo de reloj. Por ejemplo, considerar la secuencia de instrucciones:

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Este par de instrucciones pueden provocar al mismo tiempo un fallo de página de datos y una interrupción aritmética, puesto que LW está en MEM mientras ADD está en EX. Este caso se puede manipular tratando sólo el fallo de página y recomenzando después la ejecución. La segunda interrupción reocurrirá (pero no la primera, si el software es correcto), y cuando pase esto se puede tratar independientemente.

En realidad, la situación no es así de sencilla. Las interrupciones se pueden presentar en cualquier instante; es decir, una instrucción puede provocar una interrupción antes de que la cause otra instrucción anterior. Considerar, nuevamente, la secuencia anterior de instrucciones LW; ADD. LW puede producir un fallo de página de datos, cuando la instrucción esté en MEM, y ADD puede producir un fallo de página de instrucciones, cuando la instrucción ADD esté en IF. El fallo de página de instrucciones, realmente, se presentará primero, ¡aun cuando esté provocado por una instrucción posterior! Esta situación se

Etapa de la segmentación	Problema de interrupción que se presenta
IF	Fallo de página en búsqueda de instrucción; acceso de memoria mal alineado; violación de la protección de memoria
ID	Código de operación ilegal o indefinido
EX	Interrupción aritmética
MEM	Fallo de página en búsqueda de datos; acceso de memoria mal alineado; violación de la protección de memoria
WB	Ninguno

FIGURA 6.25 Interrupciones del Capítulo 5 que provocan paradas y rearranques de la segmentación de DLX de una manera transparente. La etapa de la segmentación donde estas interrupciones se presentan se muestra también. Las interrupciones debidas a accesos a instrucciones o datos contabilizan seis de cada siete casos. Estas interrupciones y sus nombres correspondientes en otros procesadores están en las Figuras 5.9 y 5.11.

puede resolver de dos formas. Para explicarlas, denominemos a la instrucción en la posición LW «instrucción *i*» y a la instrucción en la posición ADD «instrucción $i + 1$».

El primer enfoque es completamente preciso y es el más sencillo de comprender para el usuario de la arquitectura. El hardware envía cada interrupción a un vector de status arrastrado con cada instrucción, a la vez que avanza en el procesador segmentado. Cuando una instrucción entra en WB (o está próxima a dejar MEM), se examina el vector de status de interrupciones. Si hay algunas interrupciones, son tratadas en el mismo orden que se produjeron —primero se trata la interrupción correspondiente a la instrucción más antigua—. Esto garantiza que todas las interrupciones se verán en la instrucción *i* antes que en la $i + 1$. Por supuesto, cualquier acción llevada a cabo por parte de la instrucción *i* puede ser inválida, pero como no se cambia de estado hasta WB, esto no es un problema en el procesador de DLX. No obstante, el control del procesador puede desear cualquier acción llevada a cabo por una instrucción *i* (y sus sucesores) tan pronto como sea reconocida la interrupción. Para segmentaciones que puedan actualizar el estado antes que WB, es necesaria esta inhabilitación.

El segundo enfoque consiste en tratar una interrupción tan pronto como aparezca. Esto se puede considerar un poco menos preciso porque las interrupciones se presentan en un orden distinto del que se podrían presentar si no existiese segmentación. La Figura 6.26 muestra dos interrupciones que se presentan en la segmentación de DLX. Como la interrupción de la instrucción $i + 1$ es tratada cuando aparece, el procesador se debe detener inmediatamente sin dejar completar ninguna de las instrucciones que tengan que cambiar el estado. Para la segmentación de DLX, esto será $i - 2$, $i - 1$, i, e $i + 1$, suponiendo que la interrupción sea reconocida al final de la etapa IF de la instrucción ADD. El procesador se reinicia entonces con la instrucción

Instrucción $i - 3$	IF	ID	EX	MEM	WB					
Instrucción $i - 2$		IF	ID	EX	MEM	*WB*				
Instrucción $i - 1$			IF	ID	EX	*MEM*	*WB*			
Instrucción i (LW)				IF	ID	*EX*	*MEM*	*WB*		
Instrucción $i + 1$ (ADD)					**IF**	*ID*	*EX*	*MEM*	*WB*	
Instrucción $i + 2$						*IF*	*ID*	*EX*	*MEM*	*WB*

Instrucción $i - 3$	IF	ID	EX	MEM	WB					
Instrucción $i - 2$		IF	ID	EX	MEM	WB				
Instrucción $i - 1$			IF	ID	EX	MEM	WB			
Instrucción i (LW)				IF	ID	EX	**MEM**	*WB*		
Instrucción $i + 1$ (ADD)					IF	ID	*EX*	*MEM*	*WB*	
Instrucción $i + 2$						IF	*ID*	*EX*	*MEM*	*WB*
Instrucción $i + 3$							*IF*	*ID*	*EX*	*MEM*
Instrucción $i + 4$								IF	ID	EX

FIGURA 6.26 Las acciones tomadas por las interrupciones que se presentan y se tratan en diferentes puntos de la segmentación. Esto muestra las instrucciones interrumpidas cuando se presenta un fallo de página de instrucción en la instrucción $i + 1$ (en el diagrama superior) y un fallo de página de dato en la instrucción i en el diagrama inferior. Las etapas en negrita son los ciclos durante los cuales se reconoce la interrupción. Las etapas en itálica son las instrucciones que no se completarán debido a la interrupción, y necesitarán recomenzarse. Debido a que el efecto más antiguo de la interrupción está en la etapa de la segmentación después que ésta ocurre, las instrucciones que están en la etapa WB cuando se presenta la interrupción se completarán, mientras que las que no hayan alcanzado la etapa WB serán detenidas y recomenzadas.

$i - 2$. Como la instrucción causante de la interrupción puede ser cualquiera de $i - 2$..., $i + 1$, el sistema operativo debe determinar la instrucción del fallo. Esto es fácil de resolver si se conoce el tipo de interrupción y su correspondiente etapa en la segmentación. Por ejemplo, solamente $i + 1$ (la instrucción ADD) produciría un fallo de página de instrucciones en este punto, y sólo $i - 2$ produciría un fallo de página de datos. Después de tratar el fallo para $i + 1$ y recomenzar en $i - 2$, el fallo de página de datos ocurrirá en la instrucción i, lo que hará que i, ..., $i + 3$ sean interrumpidas. Entonces puede ser tratado el fallo de página de datos.

Complicaciones del repertorio de instrucciones

Otra serie de dificultades surge de bits de estado singulares que pueden crear riesgos adicionales en la segmentación o pueden necesitar hardware extra para guardar y restaurar. Los códigos de condición son un buen ejemplo. Muchas máquinas modifican implícitamente los códigos de condición como parte de

la instrucción. A primera vista, esto parece una buena idea, ya que los códigos de condición separan la evaluación de la condición del salto real. Sin embargo, los códigos de condición modificados, implícitamente, pueden provocar dificultades para hacer los saltos más rápidos. Limitan la efectividad de la planificación de los saltos porque la mayoría de las operaciones modificarán el código de condición, haciendo difícil planificar instrucciones entre la inicialización del código de condición y el salto. Además, en máquinas con códigos de condición, el procesador debe decidir cuándo la condición de salto está fija. Esto involucra determinar cuándo se inicializó por última vez el código de condición antes del salto. En el VAX, la mayoría de las instrucciones modifican el código de condición, de forma que una implementación tendrá que detenerse si trata de determinar muy pronto la condición de salto. Alternativamente, la condición de salto la puede evaluar el propio salto al final de su ejecución, pero esto conduce a un gran retardo del salto. En los 360/370, muchas, pero no todas, las instrucciones modifican los códigos de condición. La Figura 6.27 muestra cómo difiere la situación en DLX, el VAX y el 360 para la siguiente secuencia de código C, suponiendo que `b` y `d` están inicialmente en los registros R2 y R3 (y no se deben destruir):

```
a = b + d;
if (b == 0) ...
```

Suponiendo que tuviéramos hardware de sobra, **todas** las instrucciones antes del salto en curso podrían ser examinadas para decidir cuándo se determina el salto. Por supuesto, las arquitecturas que explícitamente inicializan los códigos de condición evitan esta dificultad. Sin embargo, el control de la seg-

DLX	VAX	IBM 360
ADD R1,R2,R3	ADDL3 a,R2,R3	LR R1,R2
...	...	AR R1,R3
SW a,R1	CL R2,0	ST a,R1
...	BEQL etiqueta	...
BEQZ R2,etiqueta		LTR R2,R2
		BZ etiqueta

FIGURA 6.27 Secuencias de código para las dos instrucciones anteriores. Debido a que ADD calcula la suma de `b` y `d`, y la condición de salto depende sólo de `b`, en el VAX y el 360 se necesita una comparación explícita (en R2). En DLX, el salto depende sólo de R2 y puede estar arbitrariamente lejos de él. (Además SW podría situarse en el hueco de retardo de salto.) En el VAX todas las operaciones de la ALU y transferencias modifican los códigos de condición, de forma que una comparación debe estar justo antes del salto. En el 360, para este ejemplo, la instrucción de carga y examina registro (LTR) se utiliza para inicializar el código de condición. Sin embargo, la mayoría de las cargas en el 360 no inicializan los códigos de condición; por tanto, una carga (o un almacenamiento) podría situarse entre el LTR y el salto.

mentación debe todavía tomar nota de la última instrucción que inicializó el código de condición para saber cuándo se decide la condición de salto. En efecto, el código de condición se debe tratar como un operando que requiera detección de riesgos RAW para los saltos, de la misma manera que DLX hace con los registros.

Un área espinosa final de la segmentación encauzada es la de operaciones multiciclo. Imaginemos que queremos segmentar una secuencia de instrucciones VAX como ésta:

```
MOVL   R1,R2
ADDL3  42(R1),56(R1)+, @(R1)
SUBL2  R2,R3
MOVC3  @(R1)[R2],74(R2),R3
```

Estas instrucciones difieren radicalmente en el número de ciclos de reloj que necesitan: desde uno hasta varios centenares. También requieren diferente número de accesos de datos a memoria, desde cero a, posiblemente, cientos. Los riesgos por dependencias de datos son muy complejos y se presentan entre y en las instrucciones. La solución simple de hacer que todas las instrucciones se ejecuten durante el mismo número de ciclos de reloj es inaceptable, porque introduce un número enorme de riesgos y de condiciones de desvío, y hacen una segmentación inmensamente larga. Segmentar el VAX a nivel de instrucciones es difícil (como veremos en la Sección 6.9), pero los diseñadores del VAX 8800 encontraron una solución inteligente. Segmentaron la ejecución de las microinstrucciones; como éstas son sencillas (se parecen mucho a las de DLX), el control de la segmentación es mucho más fácil. Aunque no está claro que este enfoque pueda conseguir un CPI tan bajo como una segmentación a nivel de instrucciones para el VAX, es mucho más sencillo, y posiblemente en una menor duración del ciclo de reloj.

Las máquinas de carga/almacenamiento que tienen operaciones sencillas con similares cantidades de trabajo se segmentan más fácilmente. Si los arquitectos son conscientes de la relación entre el diseño del repertorio de instrucciones y segmentación, podrán diseñar arquitecturas para segmentaciones más eficientes. En la siguiente sección veremos cómo la segmentación de DLX trata instrucciones de larga ejecución.

6.6 Extensión de la segmentación de DLX para manipular operaciones multiciclo

Ahora queremos explorar cómo se puede ampliar la segmentación de DLX para manipular operaciones de punto flotante. Esta sección se concentra en el enfoque básicoy en alternativas de diseño, y se cierra con algunas medidas de rendimiento de una segmentación para punto flotante en DLX.

No es práctico exigir que todas las operaciones de punto flotante de DLX se completen en un ciclo de reloj, o incluso en dos. Hacer eso significaría aceptar un reloj lento o utilizar mucha lógica en las unidades de punto flo-

tante, o ambas cosas. En cambio, la segmentación de punto flotante (FP) permitirá una mayor latencia para las operaciones. Esto es más fácil de comprender si imaginamos que las instrucciones en punto flotante tienen la misma segmentación que las instrucciones enteras, con dos cambios importantes. Primero, el ciclo EX se puede repetir tantas veces como sea necesario para completar la operación; el número de repeticiones puede variar para diferentes operaciones. Segundo, puede haber múltiples unidades funcionales de punto flotante. Se presentará una detención si la instrucción que se va a tratar puede provocar o un riesgo estructural para la unidad funcional que utiliza o un riesgo por dependencias de datos.

En esta sección suponer que hay cuatro unidades funcionales separadas en nuestra implementación de DLX:

1. La unidad principal entera
2. Multiplicador FP y entero
3. Sumador FP
4. Divisor FP y entero

La unidad entera manipula todas las cargas y almacenamientos en ambos conjuntos de registros, todas las operaciones enteras (excepto la multiplicación y división), y los saltos. Por ahora supondremos también que las etapas de ejecución de las otras unidades funcionales no están segmentadas, para que ninguna otra instrucción que utilice la unidad funcional pueda aparecer hasta que la instrucción anterior deje EX. Además, si una instrucción no puede continuar a la etapa EX, se detendrá por completo la segmentación a partir de esa instrucción. La Figura 6.28 muestra la estructura de la segmentación resultante. En la sección siguiente trataremos con esquemas que permitan que progrese el procesador cuando haya más unidades funcionales o cuando las unidades funcionales estén segmentadas.

Como la etapa EX se puede repetir muchas veces —de 30 a 50 repeticiones para una división en punto flotante no sería excesivo— debemos encontrar una forma de detectar largas dependencias potenciales y resolver riesgos que duran decenas de ciclos de reloj, en lugar de solamente uno o dos. También hay que tratar el solapamiento entre instrucciones enteras y de punto flotante. Sin embargo, el solapamiento de instrucciones enteras y de FP no complica la detección de riesgos, excepto en las referencias a memoria de punto flotante y transferencias entre los conjuntos de registros. Esto es porque, excepto para estas referencias a memoria y transferencias, los registros enteros y FP son distintos, y todas las instrucciones enteras operan sobre registros enteros, mientras que las operaciones en punto flotante operan sólo sobre sus propios registros. Esta simplificación del control de la segmentación es la principal ventaja de tener ficheros de registros separados para datos enteros y de punto flotante.

Por ahora, supongamos que todas las operaciones en punto flotante emplean el mismo número de ciclos de reloj —por ejemplo, 20 en la etapa EX. ¿Qué tipo de circuitería de detección de riesgos se necesitará? Como todas las operaciones tardan la misma cantidad de tiempo, y las lecturas y escrituras de

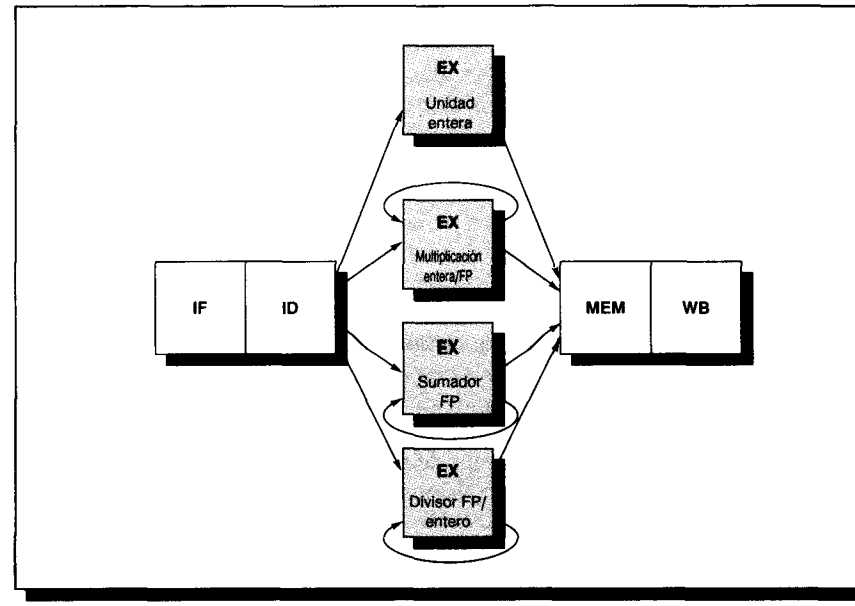


FIGURA 6.28 La segmentación de DLX con tres unidades funcionales de punto flotante no segmentadas. Debido a que se emite en cada ciclo de reloj sólo una instrucción, todas las instrucciones van a través de la segmentación estándar para operaciones enteras. Las operaciones en punto flotante, sencillamente, forman un bucle cuando alcanzan la etapa EX. Después que han terminado la etapa EX, proceden a MEM y WB para completar la ejecución.

los registros siempre ocurren en la misma etapa, solamente son posibles riesgos RAW. Los riesgos WAR o WAW no se pueden presentar. Entonces, todo lo que necesitamos conocer es el registro destino de cada unidad funcional activa. Cuando queramos emitir una nueva instrucción en punto flotante, realizaremos los siguientes pasos:

1. *Comprobar riesgos estructurales.* Esperar hasta que la unidad funcional requerida no esté ocupada.
2. *Comprobar riesgos por dependencias de datos RAW.* Esperar hasta que los registros fuente no estén en la lista de destinos para cualquiera de las etapas EX, en las unidades funcionales.
3. *Comprobar adelantamientos.* Examinar si el registro destino de una instrucción, en MEM o WB, es uno de los registros fuente de la instrucción de punto flotante; si es así, habilitar el multiplexor de entrada para utilizar ese resultado, en lugar del contenido del registro.

Surge una pequeña complicación de conflictos entre las instrucciones de carga en punto flotante y las operaciones en punto flotante cuando ambas alcanzan simultáneamente la etapa WB. Trataremos a continuación esta situación de una forma general.

La discusión anterior suponía que todos los tiempos de ejecución de las unidades funcionales FP eran iguales. Sin embargo, esto es imposible de mantener en la práctica: las sumas de punto flotante pueden realizarse, normalmente, en menos de 5 ciclos de reloj; las multiplicaciones en menos de 10, y las divisiones en 20 como mínimo. Lo que queremos es permitir que los tiempos de ejecución de las unidades funcionales sean diferentes, aunque todavía se solapen las ejecuciones. Esto no cambiaría la estructura básica de la segmentación de la Figura 6.28, aunque puede lograr que varíe el número de iteraciones en torno a los bucles. Sin embargo, el solapamiento en la ejecución de instrucciones cuyos tiempos de ejecución difieren, crea tres complicaciones: contención para accesos a registros al final de la segmentación, la posibilidad de riesgos WAR y WAW y mayor dificultad para proporcionar interrupciones precisas.

Ya hemos visto que las instrucciones de carga y las operaciones FP pueden competir en las escrituras al fichero de registros de punto flotante. Cuando las operaciones de punto flotante tengan distintos tiempos de ejecución, también pueden colisionar cuando traten de escribir resultados. Este problema se puede resolver estableciendo una prioridad estática para utilizar la etapa WB. Cuando múltiples instrucciones intenten entrar simultáneamente en la etapa MEM, todas, excepto la de mayor prioridad, son detenidas en su etapa EX. Una heurística sencilla, aunque a veces subóptima, es dar prioridad a la unidad con mayor latencia, ya que, probablemente, será la causante del cuello de botella. Aunque este esquema es razonablemente sencillo de implementar, este cambio de la segmentación de DLX es bastante significativo. En la segmentación de operaciones enteras, todos los riesgos se comprobaban antes de enviar la instrucción a la etapa EX. Con este esquema para determinar accesos al puerto de escritura de resultados, las instrucciones pueden detenerse después que se emitan.

El solapamiento de instrucciones con diferentes tiempos de ejecución puede introducir riesgos WAR y WAW en nuestro procesador DLX, porque el instante en el que las instrucciones escriben deja de ser fijo. Si todas las instrucciones todavía leen sus registros al mismo tiempo, no se introducirán riesgos WAR.

Los riesgos WAW se introducen porque las instrucciones pueden escribir sus resultados en un orden diferente en el que éstas aparecen. Por ejemplo, considerar la siguiente secuencia de código:

```
DIVF    F0,F2,F4
SUBF    F0,F8,F10
```

Se presenta un riesgo WAW entre las operaciones de división y resta. La resta se completará primero, escribiendo su resultado antes que la división escriba el suyo. ¡Observar que este riesgo solamente ocurre cuando se sobreescriba el resultado de la división, **sin** que lo utilice ninguna instrucción. Si hubiese una utilización de F0 entre `DIVF` y `SUBF`, la segmentación se detendría debido a la dependencia de los datos, y `SUBF` no se emitiría hasta que se completase `DIVF`. Podemos argüir que, para nuestra segmentación, los riesgos WAW sólo se presentan cuando se ejecuta una instrucción inoperante, pero debemos detectarla a pesar de todo y asegurarnos que el resultado de `SUBF` aparezca en F0 cuando

ambas instrucciones hayan finalizado. (Como veremos en la Sección 6.10, estas secuencias a veces se presentan en un código razonable.)

Hay dos formas posibles de tratar este riesgo WAW. El primer enfoque consiste en retardar la emisión de la instrucción de restar hasta que DIVF entre en MEM. El segundo enfoque consiste en eliminar el resultado de la división detectando el riesgo, e indicando a la unidad de dividir que no escriba su resultado. Por tanto, SUBF puede proseguir en seguida. Como este riesgo es raro, ambos esquemas funcionarán bien —se puede escoger el que sea más sencillo de implementar. Sin embargo, cuando una segmentación se vuelve más compleja, necesitaremos dedicar mayores recursos para determinar cuándo puede emitirse una instrucción.

Otro problema causado por las instrucciones de larga ejecución se puede ilustrar con una secuencia de código muy similar:

```
DIVF    F0,F2,F4
ADDF    F10,F10,F8
SUBF    F12,F12,F14
```

Esta secuencia de código parece sencilla; no hay dependencias. El problema con que nos enfrentamos surge porque una instrucción que aparezca antes, pueda completarse después de una instrucción que aparezca posteriormente. En este ejemplo, podemos esperar que ADDF y SUBF se completen **antes** que se complete DIVF. A esto se denomina *terminación fuera de orden* (*out-of-order completion*) y es común en los procesadores segmentados con operaciones de larga ejecución. Ya que la detección de riesgos impide que cualquier dependencia entre instrucciones sea violada ¿por qué es un problema la terminación fuera de orden? Suponer que SUBF provoca una interrupción aritmética de punto flotante cuando se haya completado ADDF, pero DIVF no. El resultado será una interrupción imprecisa, algo que estamos tratando de evitar. Da la impresión que esta situación puede tratarse parando la segmentación de punto flotante, como hacíamos para las instrucciones enteras. Pero la interrupción puede estar en una posición donde esto no sea posible. Por ejemplo, si DIVF provoca una interrupción aritmética de punto flotante una vez que se completa la suma, podríamos no tener una interrupción precisa a nivel hardware. En efecto, como ADDF destruye uno de sus operandos, no podríamos restaurar el estado en que anteriormente estaba DIVF, aun con la ayuda del software.

Este problema se crea porque las instrucciones se completan en un orden diferente del que aparecieron. Hay cuatro enfoques posibles para tratar la terminación fuera de orden. El primero es ignorar el problema y conformarse con interrupciones imprecisas. Este enfoque fue utilizado en los años sesenta y a comienzos de los setenta. Se utiliza todavía en algunos supercomputadores, donde no se permiten ciertas clases de interrupciones o son tratados por el hardware sin detener la segmentación. Pero en muchas máquinas construidas hoy día es difícil utilizar este enfoque, debido a características como memoria virtual y el estándar de punto flotante del IEEE, que, esencialmente, requieren interrupciones precisas, a través de una combinación de hardware y software.

Un segundo enfoque es poner en cola los resultados de una operación, hasta

que se completen todas las operaciones que comenzaron antes. Algunas máquinas utilizan realmente esta solución, pero llega a ser cara cuando la diferencia de los tiempos de ejecución entre las operaciones es grande, ya que el número de resultados puede hacer muy larga la cola. Además, los resultados de la cola deben ser desviados para continuar la ejecución de instrucciones mientras se espera la instrucción más larga. Esto requiere un gran número de comparadores y un multiplexor muy grande. Hay dos variaciones viables en este enfoque básico. El primero es un *fichero de historia*, utilizado en el CYBER 180/990. El fichero de historia mantiene los valores originales de los registros. Cuando se presente una interrupción y haya que volver al estado anterior antes que se complete alguna instrucción fuera de orden, el valor original del registro se puede restaurar a partir del fichero de historia. Una técnica similar se utiliza para el direccionamiento de autoincremento y autodecremento en máquinas como los VAX. Otro enfoque, el *fichero de futuro*, propuesto por J. Smith y Plezkun [1988], mantiene el valor más nuevo de un registro; cuando se hayan completado todas las instrucciones anteriores, el fichero principal de registros se actualiza a partir del fichero de futuro. En una interrupción, el fichero principal de registros tiene los valores precisos del estado interrumpido.

Una tercera técnica en uso es permitir que las interrupciones lleguen a ser algo imprecisas, pero manteniendo suficiente información para que las rutinas de manejo de traps puedan crear una secuencia precisa para la interrupción. Esto significa conocer las operaciones que estaban en el procesador y sus PC. Entonces, después de tratar un traps, el software termina cualquier instrucción que preceda a la última instrucción completada, y la secuencia puede recomenzar. Considerar la siguiente secuencia de código que plantea el peor caso:

Instrucción$_1$ —una instrucción de larga ejecución que, eventualmente, interrumpe la ejecución

Instrucción$_2$, ..., instrucción$_{n-1}$ —una serie de instrucciones que no se completan

Instrucción$_n$ —una instrucción que se termina

Dados los PC de todas las instrucciones en curso y el PC de retorno de interrupción, el software puede determinar el estado de la instrucción$_1$ y de la instrucción$_n$. Ya que la instrucción$_n$ se ha completado, queremos recomenzar la ejecución en la instrucción$_{n+1}$. Después de tratar la interrupción, el software debe simular la ejecución de: instrucción$_1$, ..., instrucción$_{n-1}$. Entonces, podemos volver de la interrupción y recomenzar la instrucción$_{n+1}$. La complejidad de ejecutar estas instrucciones adecuadamente es la dificultad principal de este esquema. Hay una simplificación importante: si instrucción$_2$, ..., instrucción$_n$ son instrucciones enteras, entonces sabemos que si se ha completado la instrucción$_n$, también se ha completado toda la secuencia: instrucción$_2$, ..., instrucción$_{n-1}$. Por tanto, sólo necesitan ser tratadas las operaciones de punto flotante. Para que este esquema sea tratable, se puede limitar el número de instrucciones de punto flotante cuya ejecución se pueda solapar. Por ejemplo, si sólo se solapan dos instrucciones, entonces sólo hay que completar por software la instrucción interrumpida. Esta restricción puede reducir el rendi-

miento potencial si las segmentación de FP es profunda o si hay un número significativo de unidades funcionales FP. Este enfoque se utiliza en la arquitectura SPARC para permitir el solapamiento de operaciones enteras y de punto flotante.

La técnica final es un esquema híbrido que permite que continúe la emisión de instrucciones sólo si es cierto que todas las instrucciones anteriormente emitidas terminarán sin provocar ninguna interrupción. Esto garantiza que cuando se presente una interrupción, no se complete ninguna instrucción posterior a la de la interrupción y que se puedan completar todas las instrucciones anteriores a la de la interrupción. Esto, a veces, significa detener la máquina para mantener interrupciones precisas. Para hacer que funcione este esquema, las unidades funcionales de punto flotante deben determinar al principio de la etapa si es posible que se provoque una interrupción al principio de la etapa EX (en los tres primeros ciclos de reloj de la segmentación del DLX), para impedir que se completen las siguientes instrucciones. Este esquema se utiliza en la arquitectura MIPS R2000/3000 y se explica en el Apéndice A, Sección A.7.

Rendimiento de una segmentación FP para DLX

Para examinar el rendimiento de la segmentación FP de DLX, necesitamos especificar la latencia y restricciones de emisión para las operaciones FP. Hemos escogido la estructura de la segmentación de la unidad FP MIPS R2010/

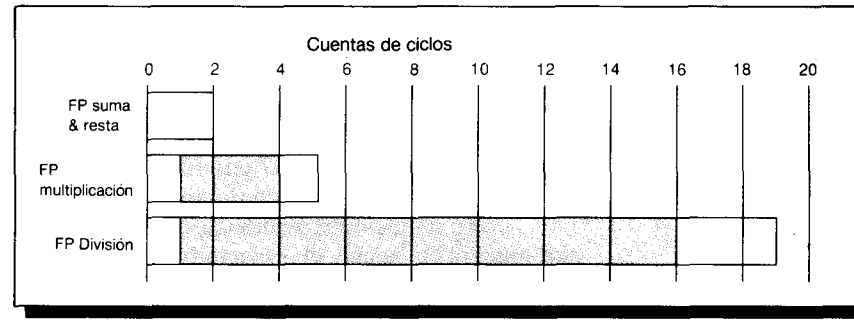


FIGURA 6.29 Recuento total de ciclos de reloj y solapamiento permisible entre operaciones de punto flotante en doble precisión en la unidad del FP de la MIPS R2010/3010. La longitud global de la barra muestra el número total de ciclos EX requeridos para completar la operación. Por ejemplo, después de cinco ciclos de reloj está disponible un resultado de multiplicación. Las regiones sombreadas son los intervalos durante los cuales pueden solaparse las operaciones FP. Como es común en muchas unidades FP, se comparte parte de la lógica FP —la lógica de redondeo, por ejemplo, se comparte con frecuencia—. Esto significa que las operaciones FP con diferentes tiempos de ejecución no se pueden solapar arbitrariamente. Observar también que la multiplicación y división no están segmentadas en esta unidad FP, de forma que sólo una multiplicación o división puede estar pendientes. La motivación para este diseño de segmentación se explica más adelante en el Apéndice A.

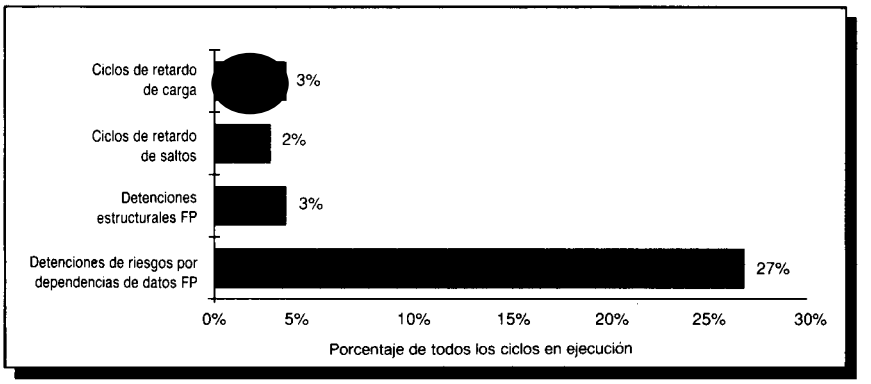


FIGURA 6.30 Porcentaje de los ciclos de reloj de Spice que son detenciones. Se asume de nuevo un sistema perfecto de memoria sin detenciones memoria-sistema. En total, el 35 por 100 de los ciclos de reloj de Spice son detenciones, y, sin ninguna detención, Spice correría, aproximadamente, un 50 por 100 más rápido. El porcentaje de detenciones difiere de la Figura 6.24 porque este recuento de ciclos incluye todas las detenciones de FP, mientras que el gráfico anterior incluye sólo las detenciones enteras.

3010. Aunque esta unidad tenga algunos riesgos estructurales, tiende a tener operaciones FP de baja latencia comparadas con la mayor parte de otras unidades FP. Las latencias y restricciones para las operaciones de punto flotante DP se explican en la Figura 6.29.

La Figura 6.30 da la descomposición de detenciones enteras y de punto flotante para Spice. Hay cuatro clases de detenciones: retardos de carga, retardos de salto, retardos estructurales de punto flotante y riesgos por dependencias de datos de punto flotante. El compilador trata de planificar los retardos de cargas y FP antes de planificar los retardos de salto. Es interesante saber que, aproximadamente, el 27 por 100 del tiempo de Spice se emplea en la espera de un resultado de punto flotante. Como los riesgos estructurales son pequeños, segmentaciones adicionales de la unidad de punto flotante no ganarían mucho. De hecho, el impacto puede ser fácilmente negativo si la latencia de la segmentación de punto flotante llegase a ser mayor.

6.7 Segmentación avanzada: planificación dinámica de la segmentación

Hasta ahora, hemos supuesto que nuestro procesador busca una instrucción y la emita, a menos que ya exista una dependencia de datos entre una instrucción en curso y la instrucción buscada. Si hay dependencia de datos, entonces detenemos la instrucción y cesa la búsqueda y emisión hasta que se elimine la dependencia. El software es responsable de la planificación de las instrucciones para minimizar estas detenciones. Este enfoque, que se denomina *planificación estática*, aunque se utilizó por primera vez en los años sesenta, ha lle-

gado a ser popular recientemente. Muchas de las primeras máquinas intensamente segmentados utilizaron la *planificación dinámica*, según la cual el hardware reorganiza la ejecución de la instrucción para reducir las detenciones.

La planificación dinámica ofrece una serie de ventajas: habilita el tratamiento de algunos casos cuando las dependencias son desconocidas en tiempo de compilación, y simplifica el compilador. También permite que un código compilado pensando en una determinada segmentación, se ejecute eficientemente en un procesador con diferente segmentación. Como veremos, estas ventajas suponen un aumento significativo de complejidad hardware. Las dos primeras partes de esta sección tratan de reducir el coste de las dependencias de datos, especialmente en máquinas segmentadas profundamente. En correspondencia con las técnicas dinámicas de hardware, para planificar dependencias relacionadas con los datos, están las técnicas dinámicas para tratar saltos. Estas técnicas son utilizadas para dos propósitos: predecir si un salto será efectivo, y determinar el destino de forma más rápida. La *predicción hardware de saltos*, el nombre para estas técnicas, es el tema de la tercera parte de esta sección avanzada.

Planificación dinámica en torno a los riesgos con marcador *(Scoreboard)*

La principal limitación de las técnicas de segmentación que hemos utilizado hasta ahora es que todas emiten en orden las instrucciones. Si en el procesador se detiene una instrucción, las posteriores no pueden proceder. Si hay múltiples unidades funcionales, éstas permanecerán inactivas. Así, si una instrucción *j* depende de una instrucción de larga ejecución *i*, actualmente en ejecución, entonces todas las instrucciones posteriores a *j* se deben detener hasta que se termine *i* y se pueda ejecutar *j*. Por ejemplo, considerar el código:

```
DIVF    F0,F2,F4
ADDF    F10,F0,F8
SUBF    F8,F8,F14
```

La instrucción SUBF no puede ejecutarse, ya que la dependencia de ADDF sobre DIVF hace que se detenga la segmentación; aunque SUBF no depende de ninguna cosa del procesador segmentado. Esto es una limitación de rendimiento que se puede eliminar al no ser necesario que las instrucciones se ejecuten en orden.

En la segmentación de DLX, los riesgos por dependencias de datos y los estructurales se comprobaban en ID: cuando una instrucción se podía ejecutar adecuadamente, se emitía desde ID. Para comenzar la ejecución de SUBF en el ejemplo anterior, debemos separar el proceso en dos partes: comprobar los riesgos estructurales y esperar la ausencia de riesgos por dependencia de datos. Se pueden comprobar todavía riesgos estructurales cuando se emita una instrucción; por ello, todavía se emiten en orden las instrucciones. Sin embargo, queremos que comience la ejecución de las instrucciones tan pronto como sus operandos estén disponibles. Por tanto, el procesador realizará la

ejecución fuera de orden, lo que, obviamente, implica *terminación fuera de orden*.

Al introducir la ejecución fuera de orden, hemos dividido esencialmente dos etapas de la segmentación de DLX en tres etapas. Las dos etapas de DLX eran:

1. ID —decodificar la instrucción, comprobar todos los riesgos y buscar operandos.
2. EX —ejecutar la instrucción.

En la segmentación de DLX, todas las instrucciones pasaban a través de una etapa de emisión en orden, y una instrucción detenida en ID provocaba una detención de todas las instrucciones que la seguían. Las tres etapas que se necesitan para permitir la ejecución fuera de orden son:

1. Emitir: decodificar instrucciones, comprobar riesgos estructurales

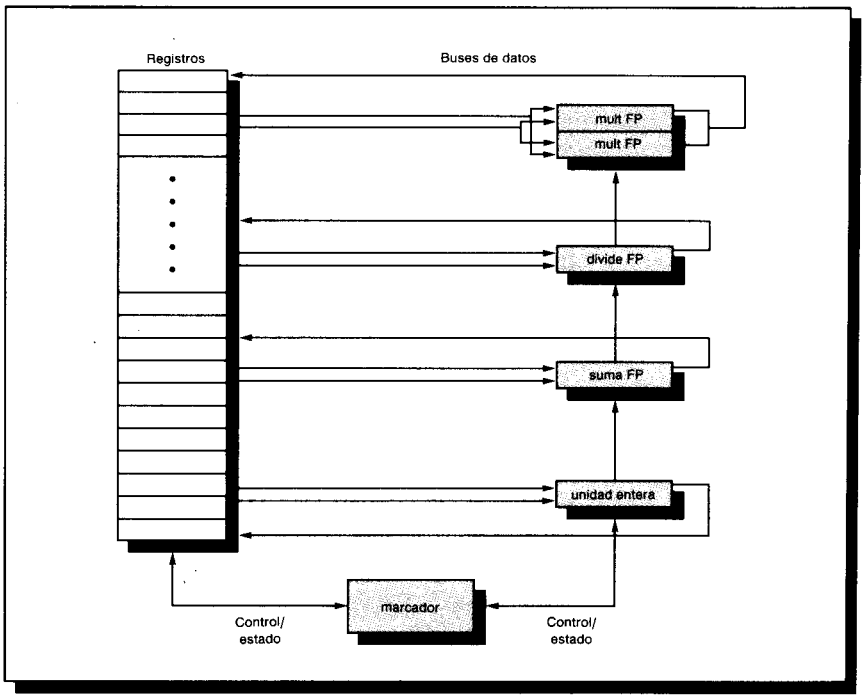


FIGURA 6.31 Estructura básica de una máquina DLX con un marcador. La función del marcador es controlar la ejecución de instrucciones (líneas de control verticales). Todos los datos fluyen sobre los buses (las líneas horizontales, llamadas troncales «trunks» en el CDC 6600) entre el fichero de registros y las unidades funcionales. Hay dos multiplicadores FP, un divisor FP, un sumador FP y una unidad entera. Un conjunto de buses (dos entradas y una salida) sirve un grupo de unidades funcionales. Los detalles del marcador se muestran en las Figuras 6.32-6.35.

2. Leer operandos: esperar hasta que no haya riesgos de datos, después leer operandos

3. Ejecutar

Estas tres etapas reemplazan las etapas ID y EX en la segmentación simple de DLX.

Aunque todas las instrucciones pasan a través de la etapa de emisión en orden (*issue in-order*), pueden ser detenidas o desviadas en la segunda etapa (lectura de operandos), y por ello entrar en la ejecución fuera de orden. El *marcaje* (*scoreboarding*) es una técnica que permite que las instrucciones se ejecuten fuera de orden cuando hay suficientes recursos y no dependencias de datos; este nombre procede del marcador del CDC 6600, que desarrolló esta capacidad.

Antes de que veamos cómo se puede utilizar el marcaje en la segmentación de DLX, es importante observar que los riesgos WAR, que no existen en las segmentaciones enteros o de punto flotante de DLX, pueden existir cuando las instrucciones se ejecutan fuera de orden. En el ejemplo anterior, el destino de SUBF es F8. Es posible ejecutar SUBF antes que ADDF, pero se obtendría un resultado incorrecto si ADDF no ha leído F8 antes que SUBF escriba su resultado. El riesgo, en este caso, se puede evitar con dos reglas: (1) leer registros sólo durante «Leer Operandos» y (2) poner en una cola tanto la operación ADDF **como** copias de sus operandos. Por supuesto, se deben detectar todavía los riesgos WAW, como ocurriría si el destino de SUBF fuese F10. Este riesgo WAW se puede eliminar deteniendo la emisión de la instrucción SUBF.

El objeto de un marcador es mantener una velocidad de ejecución de una instrucción por ciclo de reloj (cuando no hay riesgos estructurales) ejecutando cada instrucción lo antes posible. Por tanto, cuando se detiene la primera instrucción de la cola, se pueden emitir y ejecutar otras instrucciones si no dependen de ninguna instrucción activa o detenida. El marcador es el responsable de emitir y ejecutar las instrucciones, incluyendo todas las detecciones de riesgos. Para obtener beneficios de la ejecución fuera de orden se requiere que múltiples instrucciones estén simultáneamente en su etapa EX. Esto se puede conseguir con unidades funcionales múltiples o con unidades funcionales segmentadas. Como estas dos posibilidades —las unidades funcionales segmentadas y las múltiples— son esencialmente equivalentes para los propósitos de control de la segmentación, supondremos que la máquina tiene unidades funcionales múltiples.

El CDC 6600 tenía 16 unidades funcionales separadas, incluyendo 4 unidades de punto flotante, 5 para referencias a memoria y 7 para operaciones enteras. En DLX, los marcadores tienen sentido sólo en la unidad de punto flotante. Supongamos que hay dos multiplicadores, un sumador, una unidad de división y una única unidad entera para todas las referencias a memoria, saltos y operaciones enteras. Aunque este ejemplo es mucho más pequeño que el CDC 6600, es suficientemente potente para demostrar los principios. Como DLX y el CDC 6600 son de carga/almacenamiento, las técnicas son casi idénticas para las dos máquinas. La Figura 6.32 muestra cómo son las máquinas.

Cada instrucción va a través del marcador, donde se construye un cuadro de dependencias de datos; este paso corresponde a la emisión de una instruc-

ción y sustituye parte del paso ID en la segmentación de DLX. Este cuadro determina entonces cuándo la instrucción puede leer sus operandos y comenzar la ejecución. Si el marcador decide que la instrucción no se puede ejecutar inmediatamente, vigila cualquier cambio del hardware y decide cuándo puede ejecutar la instrucción. El marcador también controla cuándo una instrucción puede escribir su resultado en el registro destino. Así, todas las detecciones y resoluciones de riesgos están centralizadas en el marcador. Más tarde veremos la estructura del marcador (Fig. 6.32), pero primero necesitamos comprender los pasos de ejecución y distribución de la segmentación.

Cada instrucción pasa por cuatro pasos de ejecución. (Como nos estamos concentrando en las operaciones FP, no consideraremos un paso para acceso a memoria.) Examinemos primero informalmente los pasos y veamos después en detalle cómo mantiene el marcador la información necesaria que determina cuándo progresar de un paso al siguiente. Los cuatro pasos, que sustituyen los pasos ID, EX y WB del cauce estándar de DLX, son los siguientes:

1. Emisión *(issue)*. Si una unidad funcional está libre para la instrucción, y ninguna otra instrucción activa tiene el mismo registro destino, el marcador facilita la instrucción a la unidad funcional y actualiza su estructura interna de datos. Al asegurar que ninguna otra unidad funcional activa quiere escribir su resultado en el registro destino, se garantiza que no pueden estar presentes riesgos WAW. Si existe un riesgo estructural o WAW, entonces se detiene la emisión de la instrucción, y no se emiten más instrucciones hasta que desaparezcan estos riesgos. Este paso sustituye una parte del paso ID de la segmentación de DLX.

2. Lectura de operandos. El marcador vigila la disponibilidad de los operandos fuente. Un operando fuente está disponible si ninguna instrucción activa va a escribirlo, o si en el registro que contiene el operando está siendo escrito por unidad funcional activa en ese momento. Cuando los operandos fuente están disponibles, el marcador indica a la unidad funcional que proceda a leer los operandos de los registros y que comience la ejecución. El marcador resuelve dinámicamente, en este caso, los riesgos RAW y las instrucciones se pueden enviar a ejecución fuera de orden. Este paso, junto con el de Emisión, completa la función de la etapa ID en la segmentación simple de DLX.

3. Ejecución. La unidad funcional comienza la ejecución sobre los operandos. Cuando el resultado está listo, notifica al marcador que se ha completado la ejecución. Este paso sustituye al paso EX en la segmentación de DLX y emplea múltiples ciclos en la segmentación FP de DLX.

4. Escritura del resultado. Una vez que el marcador es consciente de que la unidad funcional ha completado la ejecución, el marcador comprueba los riesgos WAR. Un riesgo WAR existe si hay una secuencia de código como la de nuestro ejemplo anterior con ADDF y SUBF. En ese ejemplo teníamos el código

```
DIVF    F0,F2,F4
ADDF    F10,F0,F8
SUBF    F8,F8,F14
```

Estado de las instrucciones					
Instrucción		Emisión	Operandos lectura	Ejecución completa	Escritura resultado
LF	F6,34(R2)	√	√	√	√
LF	F2,45(R3)	√	√	√	
MULTF	F0,F2,F4	√			
SUBF	F8,F6,F2	√			
DIVF	F10,F0,F6	√			
ADDF	F6,F8,F2				

Estado de las unidades funcionales										
Nº UF	Nombre	Ocupada	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Entero	Sí	Carga	F2	R3					
2	Mult1	Sí	Mult	F0	F2	F4	1		No	Sí
3	Mult2	No								
4	Suma	Sí	Sub	F8	F6	F2		1	Sí	No
5	Divide	Sí	Div	F10	F0	F6	2		No	Sí

Estado de registros resultado									
	F0	F2	F4	F6	F8	F10	F12	...	F30
Nº UF	2	1			4	5			

FIGURA 6.32 Componentes del marcador. Cada instrucción que se ha emitido o que está pendiente de emitirse tiene una entrada en la tabla de estado de las instrucciones. Hay una entrada en la tabla de estado de unidades funcionales para cada unidad funcional. Una vez que se emite una instrucción, se almacenan los registros operando en la tabla de estado de las unidades funcionales. Finalmente, la tabla de registros resultados indica la unidad que producirá cada resultado pendiente; el número de entradas es igual al número de registros. El registro de estado de las instrucciones dice que (1) la primera `LF` ha completado y escrito su resultado, y (2) la segunda `LF` ha completado la ejecución pero todavía no ha escrito su resultado. `MULTF`, `SUBF` y `DIVF` han sido todas emitidas pero están detenidas, esperando a sus operandos. El estado de las unidades funcionales indica que la primera unidad de multiplicación está esperando a la unidad entera, la unidad de suma está esperando a la unidad entera y la unidad de división está esperando a la primera unidad de la multiplicación. La instrucción `ADDF` está detenida debido a un riesgo estructural; desaparecerá cuando se complete `SUBF`. Si no se utiliza ninguna entrada en alguna de estas tablas de marcadores, se deja en blanco. Por ejemplo, el campo Rk no se utiliza en una carga, y la unidad Mult2 está sin usar. Por consiguiente sus campos no tienen significado. También, una vez que un operando ha sido leído, los campos Rj y Rk son inicializados a No. Estos se dejan en blanco para minimizar la complejidad de las tablas.

ADDF tiene un operando fuente F8, que coincide con el registro destino de SUBF. Pero ADDF depende realmente de una instrucción anterior. A pesar de todo, el marcador detendrá SUBF hasta que ADDF lea sus operandos. En general, a una instrucción que se está completando no se le puede permitir que escriba sus resultados cuando

- hay una instrucción que no ha leído sus operandos,
- uno de los operandos está en el mismo registro que el resultado de la instrucción que se está completando, y
- el otro operando era el resultado de una instrucción anterior.

Cuando este riesgo WAR no existe, o cuando desaparece, el marcador indica a la unidad funcional que almacene su resultado en el registro destino. Este paso sustituye el paso WB de la segmentación de DLX.

Basado en su propia estructura de datos, el marcador controla el progreso de las instrucciones de un paso al siguiente comunicando con las unidades funcionales. Pero hay una pequeña complicación: sólo hay un número limitado de buses en el fichero de registros para leer los operandos fuente y escribir resultado. El marcador debe garantizar que el número de unidades funcionales permitidas para proceder en los pasos 2 y 4 no exceda del número de buses disponibles. No entraremos en más detalles sobre esto, salvo mencionar que CDC 6600 resolvió este problema agrupando las 16 unidades funcionales en cuatro grupos y proporcionando un conjunto de buses, denominados *troncos de datos* (*data trunks*), para cada grupo. Sólo una unidad de un grupo podía leer sus operandos o escribir su resultado durante un ciclo.

Ahora examinemos la estructura de datos detallada mantenida por un marcador de DLX con cinco unidades funcionales. La Figura 6.32 muestra la información de un marcador para una sencilla secuencia de instrucciones:

```
LF      F6,34(R2)
LF      F2,45(R3)
MULTF   F0,F2,F4
SUBF    F8,F6,F2
DIVF    F10,F0,F6
ADDF    F6,F8,F2
```

Hay tres partes en el marcador:

1. Estado de las instrucciones: indica en cuál de los cuatro pasos está la instrucción.
2. Estado de las unidades funcionales: indica el estado de la unidad funcional (FU). Hay nueve campos para cada unidad funcional:

   Ocupado: indica si la unidad está ocupada o no

   Op: operación a realizar en la unidad (por ejemplo, sumar o restar)

Fi: registro destino

Fj,Fk: números de los registros fuente

Qj,Qk: número de unidades que producen los registros fuentes Fj,Fk

Rj,Rk: señalizadores que indican cuándo Fj,Fk están listos

3. Estado de registro resultado: Indica qué unidad funcional escribirá en un registro, si una instrucción activa tiene el registro como destino.

Ahora, examinemos cómo la secuencia de código comenzada en la Figura 6.32 continúa la ejecución. Después de eso, podremos examinar con detalle las condiciones que utiliza el marcador para controlar la ejecución.

Ejemplo

Suponer las siguientes latencias del ciclo EX para las unidades funcionales de punto flotante: sumar, 2 ciclos de reloj; multiplicar, 10 ciclos de reloj; dividir, 40 ciclos de reloj. Utilizando el segmento de código de la Figura 6.32, y comenzando en el punto indicado por el estado de las instrucciones de la Figura 6.32, mostrar cómo serán las tablas de estado cuando `MULTF` y `DIVF` estén listas para ir al estado de escribir-resultado.

Respuesta

Hay riesgos de dependencia de datos RAW entre la segunda `LF` y `MULTF` y `SUBF`; entre `MULTF` y `DIVF`; entre `SUBF` y `ADDF`. Hay un riesgo de datos WAR entre `DIVF` y `ADDF`. Finalmente, hay un riesgo estructural en la unidad funcional de suma para `ADDF`. El aspecto de las tablas cuando `MULTF` y `DIVF` están listas para escribir el resultado se muestra en las Figuras 6.33 y 6.34, respectivamente.

Ahora, podemos ver con detalle cómo funciona el marcador, examinando lo que tiene que ocurrir para que el marcador permita que proceda cada instrucción. La Figura 6.35 muestra lo que el marcador comprueba para que avance cada instrucción y las acciones necesarias cuando la instrucción avanza.

Los costes y beneficios del marcador son una pregunta interesante. Los diseñadores del CDC 6600 midieron una mejora de rendimiento de 1,7 para programas FORTRAN y 2,5 para lenguaje ensamblador codificado a mano. Sin embargo, estas medidas se hicieron antes de que existiera la planificación software de la segmentación, memoria principal de semiconductores y caches (con tiempos de acceso a memoria más bajos). El marcador del CDC 6600 tenía aproximadamente tanta lógica como una de las unidades funcionales, lo que es sorprendentemente poco. El coste principal estuvo en el gran número de buses —aproximadamente cuatro veces los que necesitarían si la máquina ejecutase sólo instrucciones en orden (o si sólo iniciase una instrucción por ciclo de ejecución).

El marcador no trata algunas situaciones tan bien como podría. Por ejemplo, cuando una instrucción escribe su resultado, una instrucción dependiente en curso debe esperar para acceder al fichero de registros, porque todos los resultados se escriben a través del fichero de registros y no se adelantan nunca.

Estado de las instrucciones					
Instrucción		**Emisión**	**Operandos lectura**	**Ejecución completa**	**Escritura resultado**
LF	F6,34(R2)	√	√	√	√
LF	F2,45(R3)	√	√	√	√
MULTF	F0,F2,F4	√	√	√	
SUBF	F8,F6,F2	√	√	√	√
DIVF	F10,F0,F6	√			
ADDF	F6,F8,F2	√	√		

Estado de las unidades funcionales										
N° UF	**Nombre**	**Ocupada**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**
1	Entero	No								
2	Mult1	Sí	Mult	F0	F2	F4			No	No
3	Mult2	No								
4	Suma	Sí	Sum	F6	F8	F2			No	No
5	Divide	Sí	Div	F10	F0	F6	2		No	Sí

Estado de registros resultado									
	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	...	**F30**
N° UF	2			4		5			

FIGURA 6.33 Tablas de los marcadores antes que `MULTF` escriba el resultado. `DIVF` todavía no ha leído sus operandos, ya que tiene una dependencia sobre el resultado de la multiplicación. `ADDF` ha leído sus operandos y está en ejecución, aunque estuviese forzada a esperar hasta que `SUBF` terminase para obtener la unidad funcional. `ADDF` no puede proceder a escribir los resultados a causa del riesgo WAR en F6, que es utilizado por `DIVF`.

Esto incrementa la latencia y limita la posibilidad de múltiples instrucciones, esperando un resultado para comenzar su ejecución. Los riesgos WAW serán muy infrecuentes, así que las detenciones que provoquen, probablemente, no serán significativas en el CDC 6600. Sin embargo, en la sección siguiente veremos que la planificación dinámica ofrece la posibilidad de solapar la ejecución de múltiples iteraciones de un bucle. Para hacer esto, efectivamente, se requiere un esquema para tratar los riesgos WAW, cuya frecuencia, probablemente, incrementará cuando se solapen múltiples iteraciones.

Estado de las instrucciones					
Instrucción		Emisión	Operandos lectura	Ejecución completa	Escritura resultado
LF	F6,34(R2)	√	√	√	√
LF	F2,45(R3)	√	√	√	√
MULTF	F0,F2,F4	√	√	√	√
SUBF	F8,F6,F2	√	√	√	√
DIVF	F10,F0,F6	√	√	√	
ADDF	F6,F8,F2	√	√	√	√

Estado de las unidades funcionales										
Nº UF	Nombre	Ocupada	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Entero	No								
2	Mult1	No								
3	Mult2	No								
4	Suma	No								
5	Divide	Sí	Div	F10	F0	F6			No	No

Estado de registros de resultado									
	F0	F2	F4	F6	F8	F10	F12	...	F30
Nº UF						5			

FIGURA 6.34 Tablas de marcadores justo antes que `DIVF` escriba el resultado. `ADDF` pudo completarse tan pronto como `DIVF` pasó por la lectura de operandos y obtuvo una copia de F6. Sólo queda `DIVF` por terminar.

Otro enfoque de la planificación dinámica: el algoritmo de Tomasulo

Otro método de abordar los riesgos de ejecución paralela fue utilizado por la unidad de punto flotante del IBM 360/91. Este esquema se debe a R. Tomasulo. El IBM 360/91 fue terminado unos tres años más tarde que el CDC 6600, antes que aparecieran las caches en las máquinas comerciales. El objetivo de IBM era conseguir un alto rendimiento de punto flotante de un repertorio de instrucciones y de los compiladores diseñados para la familia completa de computadores 360, mejor que para aplicaciones intensivas de punto flotante. Recordar que la arquitectura 360 sólo tiene cuatro registros de punto flotante y doble precisión, lo cual limita la efectividad de la planificación del compilador; este hecho fue otra motivación para el enfoque de Tomasulo. Final-

mente, el IBM 360/91 tenía grandes retardos en los accesos a memoria y grandes retardos de punto flotante, y para superarlos se diseñó el algoritmo de Tomasulo. Al final de la sección, veremos que el algoritmo de Tomasulo también puede soportar la ejecución solapada de múltiples iteraciones de un bucle.

Explicaremos el algoritmo, que se centra en la unidad de punto flotante, en el contexto de una unidad segmentada de punto flotante para DLX. La diferencia principal entre DLX y el 360 es la presencia de instrucciones de registro-memoria en la última máquina. Aunque el algoritmo de Tomasulo utiliza una unidad funcional de carga, no son necesarios cambios significativos para añadir los modos de direccionamiento de registro-memoria; la principal adición es otro bus. El IBM 360/91 tenía unidades funcionales segmentadas, en lugar de unidades funcionales múltiples. La única diferencia entre éstas es que una unidad segmentada puede comenzar, como mucho, una operación por ciclo de reloj. Como realmente no hay diferencias fundamentales, describimos el algoritmo como si hubiese múltiples unidades funcionales. El IBM 360/91 podía acomodar tres operaciones para el sumador de punto flotante y dos para el multiplicador de punto flotante. Además, podrían estar pendientes hasta seis instrucciones de carga de punto flotante, o referencias a memoria, y hasta tres de almacenamiento en punto flotante. Los «buffers» de carga y de almacenamiento de datos se utilizan para esta función. Aunque no explicaremos las unidades de carga y almacenamiento, necesitamos incluir los buffers para los operandos.

El esquema de Tomasulo comparte muchas ideas con el marcador del CDC 6600; por ello suponemos que el lector lo ha comprendido a fondo. Sin embargo, hay dos diferencias significativas. Primero, la detección de riesgos y el control de la ejecución están distribuidos —*estaciones de reserva* en cada

Estado de la instrucción	Espera hasta	Acciones
Emisión	No ocupado (FU) y no resultado (D)	`Ocupado(FU)← sí; Resultado(D)←FU; Fm(FU)←op; Fi(FU)←D; Fj(FU)←S1; Fk(FU)←S2; Qj←Resultado (S1); Qk←Resultado(S2); Rj← no Qj; Rk← no Qk`
Lectura operandos	Rj y Rk	`Rj←No; Rk←No`
Ejecución completa	Finaliza unidad funcional	
Escribe resultado	$\forall f$((Fj(f)≠Fi(FU) o Rj(f)=No) & (Fk(f) ≠Fi(FU) o Rk(f)=No))	`∀f(if Qj (f)=FU then Rj(f)←Sí); ∀f(if Qk (f)=FU then Rj(f)←Sí); Result (Fi(FU))←Clear; Busy(FU)←No`

FIGURA 6.35 Comprobaciones requeridas y acciones en cada paso de la ejecución de la instrucción. FU significa unidad funcional utilizada por la instrucción, D es el registro destino, S1 y S2 son los registros fuente, y *op* es la operación que se va a realizar. Para acceder a la entrada del marcador denominada F_j para la unidad funcional FU utilizamos la notación F_j(FU). Resultado (D) es el valor del campo de resultado del registro para el registro D. El test en el caso de escritura de resultado impide la escritura cuando hay un riesgo WAR, que existe si otra instrucción tiene este destino de instrucción (Fi(FU)) como fuente (Fj(f) o Fk(f)), y si alguna otra instrucción ha escrito el registro (Rj = Sí o Rk = Sí).

unidad funcional controlan cuándo puede comenzar la ejecución de una instrucción en esa unidad. Esta función está centralizada en el marcador del CDC 6600. Segundo, los resultados se pasan directamente a las unidades funcionales en lugar de ir a través de los registros. El IBM 360/91 tenía un bus común de resultados (denominado *bus común de datos*, o CDB) que permite a todas las unidades esperar a que un operando se cargue simultáneamente. El CDC 6600 escribe los resultados en registros, donde las unidades funcionales que esperan pueden tener que competir por ellos. También el CDC 6600 tiene múltiples buses de terminación (dos en la unidad de punto flotante), aunque el IBM 360/91 tiene sólo uno.

La Figura 6.36 muestra la estructura básica de una unidad de punto flotante basada en Tomasulo para DLX; no se muestra ninguna de las tablas de control de ejecución. Las estaciones de reserva contienen las instrucciones que

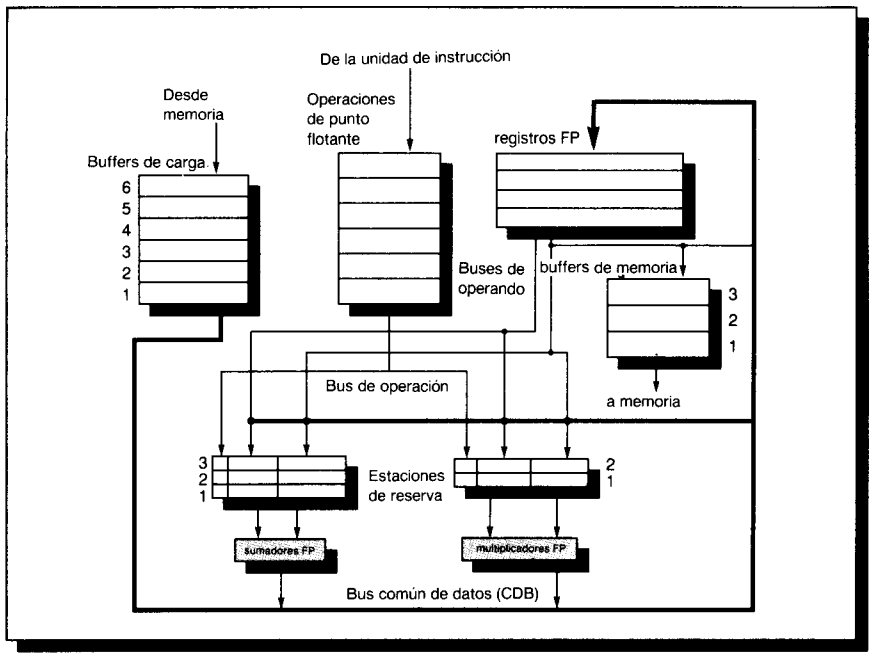


FIGURA 6.36 La estructura básica de una unidad FP para DLX utilizando el algoritmo de Tomasulo. Las operaciones de punto flotante cuando se emiten se envían de la unidad de instrucciones a una cola (llamada FLOS, o pila de operaciones de punto flotante, en el IBM 360/91). Las estaciones de reserva incluyen la operación y los operandos reales, así como la información utilizada para detectar y resolver riesgos. Hay buffers de carga para almacenar los resultados de las cargas pendientes y buffers de almacenamiento que contienen las direcciones de los almacenamientos pendientes esperando sus operandos. Todos los resultados o de las unidades FP o de la unidad de carga se ponen en el bus común de datos (CDB), que va al fichero de registros FP, así como a las estaciones de reserva y buffers de almacenamiento. Los sumadores FP implementan la suma y resta, mientras que los multiplicadores FP hacen la multiplicación y división.

han sido emitidas y están esperando su ejecución en una unidad funcional, así como la información necesaria para controlar la instrucción, una vez que haya comenzado la ejecución en la unidad. Los buffers de carga y de almacenamiento contienen datos que vienen y van a memoria. Los registros de punto flotante están conectados por un par de buses a las unidades funcionales y por un solo bus a los buffers de almacenamiento. Todos los resultados de las unidades funcionales y de memoria se envían al bus común de datos, que va a todos los sitios excepto al buffer de carga. Todos los buffers y estaciones de reserva tienen campos identificadores, empleados por el control de riesgos.

Antes de describir los detalles de las estaciones de reserva y el algoritmo, examinemos los pasos por los que pasa una instrucción —como hicimos para el marcador. Como los operandos se transmiten de forma diferente para un marcador, sólo hay tres pasos:

1. Emisión *(issue)*. Obtiene una instrucción de la cola de operaciones de punto flotante. Si la operación es de punto flotante, se emite si hay una estación de reserva vacía, y envía los operandos a la estación de reserva si están en registros. Si la operación es de carga o de almacenamiento, se puede emitir si hay un buffer disponible. Si no hay ninguna estación de reserva vacía o ningún buffer vacío, entonces se presenta un riesgo estructural y la instrucción se detiene hasta que queda libre una estación o buffer.

2. Ejecución. Si todavía no están disponibles los operandos, vigila el CDB a la espera del registro que se va a computar. Este paso comprueba los riesgos RAW. Cuando ambos operandos están disponibles, ejecuta la operación.

3. Escritura de resultado. Cuando el resultado está disponible, lo escribe en el CDB y desde allí en los registros y unidades funcionales que esperan este resultado.

Aunque estos pasos son fundamentalmente análogos a los del marcador, hay tres diferencias importantes. Primero, no hay comprobación de riesgos WAW y WAR —éstos se eliminan como consecuencia del algoritmo, como veremos dentro de poco. Segundo, el CDB se utiliza para difundir resultados en lugar de esperar que los datos estén en los registros. Tercero, las operaciones de carga y almacenamiento son tratadas como unidades funcionales básicas.

Las estructuras de datos utilizadas para detectar y eliminar riesgos están vinculadas con las estaciones de reserva, el fichero de registros y los buffers de carga y almacenamiento. Aunque se vincula a información diferente objetos diferentes, todos, excepto los buffers de carga, contienen un campo de etiqueta por entrada. El campo de etiqueta es una cantidad de cuatro bits que denota una de las cinco estaciones de reserva o uno de los seis buffers de carga. El campo de etiqueta se utiliza para describir la unidad funcional que producirá un resultado necesario como un operando fuente. Valores no usados, como cero, indican que el operando ya está disponible. Al describir la información, los nombres de los marcadores se utilizan siempre que no conduzcan a confusión. También se muestran los nombres utilizados por el IBM 360/91. Es importante recordar que las etiquetas del esquema de Tomasulo se refieren al

buffer o unidad que producirá un resultado; el número de registro se descarta cuando una instrucción se emite a una estación de reserva.

Cada estación de reserva tiene seis campos:

OP: la operación a realizar sobre los operandos fuente S1 y S2.

Qj,Qk: las estaciones de reserva que producirán el correspondiente operando fuente; un valor cero indica que el operando fuente está ya disponible en Vi o Vj, o es innecesario. El IBM 360/91 las denomina SINKunit y SOURCEunit (unidadSUMIDERO y unidadFUENTE).

Vj,Vk: el valor de los operandos fuente. Estos se denominan SINK y SOURCE en el IBM 360/91. Observar que sólo uno de los campos V o Q es válido para cada operando.

Ocupado: indica que están ocupadas esta estación de reserva y su unidad funcional acompañante.

El fichero de registros y cada buffer de almacenamiento tienen un campo, Qi.

Qi: el número de la unidad funcional que producirá un valor para que se almacene en este registro o en memoria. Si el valor de Qi es cero, ninguna instrucción actualmente activa está calculando un resultado destinado para este registro o buffer. Para un registro, esto significa que el valor viene dado por el contenido del registro.

Los buffers de carga y almacenamiento requieren un campo de «ocupado», que indique cuándo está disponible el buffer debido a que se haya completado una carga o almacenamiento allí asignadas. El buffer de almacenamiento también tiene un campo V, el valor que se va a almacenar.

Antes de que examinemos con detalle el algoritmo, veamos cuál es el aspecto del sistema de tablas para la siguiente secuencia de código:

```
1. LF     F6,34(R2)
2. LF     F2,45(R3)
3. MULTF  F0,F2,F4
4. SUBF   F8,F6,F2
5. DIVF   F10,F0,F6
6. ADDF   F6,F8,F2
```

Vimos anteriormente el aspecto del marcador cuando sólo la primera carga había escrito su resultado. La Figura 6.37 representa las estaciones de reserva, buffers de carga y almacenamiento, y las etiquetas de los registros. Los números añadidos a los nombres sum, mult, y load indican la etiqueta de la estación de reserva —Add1 (suma 1) es la etiqueta para el resultado de la primera unidad de suma—. Además, hemos incluido una tabla central denominada «Estado de las instrucciones». Esta tabla sólo se incluye para ayudar al lector a comprender el algoritmo; realmente **no** es una parte del hardware. En cambio, el estado de cada operación que se ha emitido se mantiene en una estación de reserva.

Estado de las instrucciones			
Instrucción	**Emisión**	**Ejecución**	**Escritura resultado**
`LF F6,34(R2)`	√	√	√
`LF F2,45(R3)`	√	√	
`MULTF F0,F2,F4`	√		
`SUBF F8,F6,F2`	√		
`DIVF F10,F0,F6`	√		
`ADDF F6,F8,F2`	√		

Estaciones de reserva						
Nombre	**Ocupada**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	Sí	SUB	(Load1)			Load2
Add2	Sí	ADD			Add1	Load2
Add3	No					
Mult1	Sí	MULT		(F4)	Load2	
Mult2	Sí	DIV		(Load1)	Mult1	

Estado de los registros									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Mult1	Load2		Add2	Add1	Mult2			
Ocupado	Sí	Sí	No	Sí	Sí	Sí	No	...	No

FIGURA 6.37 Estaciones de reserva y etiquetas de registros. Todas las instrucciones se han emitido, pero sólo se ha completado la primera instrucción de carga y se ha escrito su resultado en el CDB. La tabla de estado de las instrucciones, realmente, no está presente, pero la información equivalente está distribuida por todo el hardware. La notación (X), donde X es o un número de registro o una unidad funcional, indica que este campo contiene el resultado de la unidad funcional X o el contenido del registro X en el instante de la emisión. Las demás instrucciones están todas en estaciones de reserva, o como en el caso de la instrucción 2, completando una referencia de memoria. No se muestran los buffers de carga y almacenamiento. El buffer de carga 2 es el único buffer de carga ocupado y está siendo utilizado por la mitad de la instrucción 2 de la secuencia —cargar desde la dirección de memoria R3 + 45. No hay almacenamientos, por ello no se muestra el buffer de almacenamiento. Recordar que en cualquier instante un operando se especifica o por el campo Q o por el campo V.

Hay dos diferencias importantes con respecto a los marcadores, que son observables en estas tablas. Primero, el valor de un operando se almacena en la estación de reserva en uno de los campos V tan pronto como está disponible; no se lee del fichero de registros una vez que se ha emitido la instrucción. Segundo, se ha emitido la instrucción ADDF. Esta quedaba bloqueada con la técnica del marcador por un riesgo estructural. Las grandes ventajas del esquema de Tomasulo son: (1) la distribución de lógica de detección de riesgos, y (2) la eliminación de detenciones para riesgos WAW y WAR. La primera

ventaja surge de las estaciones de reserva distribuidas y del uso del CDB. Si muchas instrucciones están esperando un solo resultado, y cada instrucción ya tiene su otro operando, entonces las instrucciones se pueden liberar simultáneamente cuando el operando es difundido a través del CDB. En el marcador, todas las instrucciones que esperan deben leer sus resultados de los registros cuando estén disponibles los buses de los registros.

Los riesgos WAW y WAR se eliminan renombrando los registros que utilizan las estaciones de reserva. Por ejemplo, en nuestra secuencia de código de la Figura 6.37, hemos distribuido DIVF y ADDF, aun cuando haya un riesgo WAR involucrado a F6. El riesgo se elimina de una de dos formas. Si se ha completado la instrucción que proporciona el valor para DIVF, entonces Vk almacenará el resultado permitiendo que DIVF se ejecute independientemente de ADDF (éste es el caso mostrado). Por otro lado, si no se ha completado LF, entonces Qk señalaría a Load1 (Carga1) y la instrucción DIVF sería independiente de ADDF. Por tanto, en cualquier caso, ADDF puede emitirse y comenzar a ejecutarse. Otros usos del resultado de MULTF apuntarían a la estación de reserva, permitiendo que ADDF se complete y almacene su valor en los registros sin afectar a DIVF. En breve veremos un ejemplo de eliminación de un riesgo WAW. Pero examinemos primero cómo continúa la ejecución nuestro ejemplo anterior.

Ejemplo Suponer las mismas latencias para las unidades funcionales de punto flotante que dimos para la Figura 6.34: suma, 2 ciclos de reloj; multiplicación, 10 ciclos de reloj; división, 40 ciclos de reloj. Con el mismo segmento de código, mostrar cómo son las tablas de estado cuando MULTF esté preparada para escribir el resultado.

Respuesta El resultado se muestra en las tres tablas de la Figura 6.38. De forma distinta al ejemplo con el marcador, ADDF se ha completado, ya que los operandos de DIVF se copian eliminando así el riesgo WAR.

La Figura 6.39 muestra los pasos por los pasa cada instrucción. Las cargas y almacenamientos sólo son ligeramente especiales. Una carga se puede ejecutar tan pronto como esté disponible. Cuando se complete la ejecución, y el CDB esté disponible, una carga pone su resultado en el CDB como cualquier unidad funcional. Los almacenamientos reciben sus valores del CDB o del fichero de registros y se ejecutan autónomamente; cuando se han ejecutado desactivan el campo ocupado para indicar disponibilidad, igual que un buffer de carga o una estación de reserva.

Para comprender la potencia total de la eliminación de riesgos WAW y WAR mediante el renombramiento dinámico de los registros, debemos examinar un bucle. Considerar la siguiente secuencia para multiplicar los elementos de un vector por un escalar en F2:

```
Loop:  LD     F0,0(R1)
       MULTD  F4,F0,F2
       SD     0(R1),F4
       SUB    R1,R1, #8
       BNEZ   R1,Loop  ; salta sí R1≠0
```

Estado de las instrucciones			
Instrucción	**Emisión**	**Ejecución**	**Escritura resultado**
`LF F6,34(R2)`	√	√	√
`LF F2,45(R3)`	√	√	√
`MULTF F0,F2,F4`	√	√	
`SUBF F8,F6,F2`	√	√	√
`DIVF F10,F0,F6`	√		
`ADDF F6,F8,F2`	√	√	√

Estaciones de reserva						
Nombre	**Ocupada**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	No					
Add2	No					
Add3	No					
Mult1	Sí	MULT	(Load2)	(F4)		
Mult2	Sí	DIV		(Load1)	Mult1	

Estado de los registros									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	...	**F30**
Qi	Mult1					Mult2			
Ocupado	Sí	No	No	No	No	Sí	No	...	No

FIGURA 6.38 Multiplicación y división son las únicas instrucciones no terminadas. Esto es diferente del caso del marcador, debido a que la eliminación de los riesgos WAR ha permitido a `ADDF` que termine justo después de que `SUBF`, de la que dependía.

Con una estrategia de salto efectivo, la utilización de estaciones de reserva permitirá que múltiples ejecuciones de este bucle estén en curso a la vez. Esta ventaja se obtiene sin desenrollar el bucle —en efecto, el hardware desenrolla dinámicamente el bucle. En la arquitectura del 360, la presencia de sólo 4 registros FP limitaría severamente el uso del desenrollamiento. (Veremos brevemente que, cuando desenrollamos un bucle y lo planificamos para evitar interbloqueos, se requieren muchos más registros.) El algoritmo de Tomasulo soporta la ejecución solapada de múltiples copias del mismo bucle con sólo un número pequeño de registros utilizados por el programa.

Supongamos que todas las instrucciones de dos iteraciones sucesivas del bucle se han emitido, pero no se ha completado ninguna de las operaciones, entre las que se encuentran las cargas/almacenamientos de punto flotante. Las estaciones de reserva, tablas de estado de los registros y buffers de almacenamiento y carga, en este punto se muestran en la Figura 6.40. (La operación entera de la ALU se ignora, y se supone que el salto se predijo como efectivo.)

Una vez que el sistema alcanza este estado, dos copias del bucle deberían ser sostenidas con un CPI próximo a uno, siempre que las multiplicaciones puedan completarse en cuatro ciclos de reloj. En la Sección 6.8 veremos cómo las técnicas de compiladores pueden conseguir un resultado similar.

Un elemento adicional, que es crítico para hacer que funcione el algoritmo de Tomasulo, se muestra en este ejemplo. La instrucción de carga de la segunda iteración del bucle podría completarse fácilmente antes que el alma-

Estado de la instrucción	Espera hasta	Acción

Emisión	Estación o buffer vacío	`if (Registro[S1].Qi ≠0)`
		`  {RS[r].Qj←Registro[S1].Qi}`
		`else {RS[r].Vj←S1; RS[r].Qj←0};`
		`if (Registro[S2].Qi≠0)`
		`{RS[r].Qk←Registro[S2].Qi};`
		`else {RS[r].Vk←S2; RS[r].Qk←0}`
		`RS[r].Ocupado←sí;`
		`Registro[D].Qi=r;`

Ejecución	(RS[r].Qj=0) y (RS[r].Qk=0)	Ninguno —operandos están en Vj y Vk

Escritura resultado	Ejecución completada en *r* y CDB disponible	`∀x(if Registro[x].Qi=r)  {Fx←resultado;`
		`  Registro[x].Qi←0});`
		`∀x(if RS[x].Qj=r) {RS[x].Vj←resultado`
		`  RS[x].Qj←0});`
		`∀x(if RS[x].Qk=r)  {RS[x].Vk←resultado;`
		`  RS[x].Qk←0});`
		`∀x(if (almacenamiento[x].Qi=r)`
		`  {almacenamiento[x].V←resultado;`
		`  almacenamiento[x].Qi←0});`
		`RS[r].ocupado←No`

FIGURA 6.39 Pasos del algoritmo y qué se requiere en cada paso. Para la emisión de la instrucción, D es el destino, S1 y S2 son las fuentes, y *r* es la estación de reserva o buffer que tiene asignado D. RS es la estructura de datos de la estación de reserva. El valor devuelto por una estación de reserva o por la unidad de carga se denomina «resultado». Registro es la estructura de datos registro, mientras que Almacenamiento es la estructura de datos del buffer de almacenamiento. Cuando se emite una instrucción, el registro destino tiene su campo Qi inicializado con el número del buffer o estación de reserva a la que se emitió la instrucción. Si los operandos están disponibles en los registros, se almacenan en los campos V. En cualquier otro caso, los campos Q se inizalizan para indicar la estación de reserva que producirá los valores necesarios como operandos fuente. La instrucción espera en la estación de reserva hasta que sus operandos están disponibles, lo que viene indicado por el valor cero en los campos Q. Los campos Q se ponen a cero o cuando se emite esta instrucción o cuando una instrucción de la cual depende esta instrucción se completa y hace su post-escritura (write back). Cuando una instrucción ha terminado la ejecución y el CDB está disponible, puede hacer su postescritura. Todos los buffers, registros y estaciones de reserva cuyo valor de Qj o Qk es el mismo que la estación de reserva que acaba de finalizar, actualizan sus valores a partir del CDB y marcan los campos Q para indicar que los valores ya se han recibido. Por tanto, el CDB puede difundir su resultado a muchos destinos en un solo ciclo de reloj, y si las instrucciones de espera tienen sus operandos, pueden todos comenzar la ejecución en el siguiente ciclo de reloj.

Estado de las instrucciones					
Instrucción		**De la iteración**	**Emisión**	**Ejecución**	**Escritura resultado**
LD	F0,0(R1)	1	√	√	
MULTD	F4,F0,F2	1	√		
SD	0,(R1),F4	1	√		
LD	F0,0(R1)	2	√	√	
MULTD	F4,F0,F2	2	√		
SD	0(R1),F4	2	√		

Estaciones de reserva						
Nombre	**Ocupada**	**Fm**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	No					
Add2	No					
Add3	No					
Mult1	Sí	MULT		(F2)	Load1	
Mult2	Sí	MULT		(F2)	Load2	

Estado de los registros									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Load2		Mult2						
Ocupado	sí	no	sí	no	no	no			

Buffers de almacenamiento			
Campo	**Store 1**	**Store 2**	**Store 3**
Qi	Mult1	Mult2	
Ocupado	Sí	Sí	No
Dirección	(R1)	(R1)–8	

Buffers de carga			
Campo	**Load 1**	**Load 2**	**Load 3**
Dirección	(R1)	(R1)–8	
Ocupado	Sí	Sí	No

FIGURA 6.40 Dos iteraciones activas del bucle con ninguna instrucción todavía completada. Los buffers de carga y almacenamiento se incluyen, con las direcciones que se van a cargar y almacenar. Las cargas están en el buffer de carga; las entradas en las estaciones de reserva del multiplicador indican que los operandos fuente son las cargas pendientes. Los buffers de almacenamiento indican que van a almacenar el resultado de la multiplicación.

cenamiento de la primera iteración, aunque el orden secuencial normal sea diferente. La carga y almacenamiento pueden realizarse de forma segura en un orden diferente, siempre que la carga y almacenamiento accedan a direcciones diferentes. Esto se comprueba examinando las direcciones del buffer de almacenamiento siempre que se emita una carga. Si la dirección de la carga

coincide con la dirección del buffer de almacenamiento, debemos parar y esperar hasta que el buffer de almacenamiento obtenga un valor; entonces podemos accederlo u obtener el valor de memoria.

Este esquema puede dar un rendimiento muy alto, siempre que el coste de los saltos pueda mantenerse pequeño —éste es un problema que veremos más tarde en esta sección. También hay limitaciones impuestas por la complejidad del esquema de Tomasulo, que requiere gran cantidad de hardware. En particular, hay muchas memorias asociativas que deben correr a alta velocidad, así como lógica de control compleja. Finalmente, la ganancia de rendimiento está limitada por el bus de terminación (CDB). Aunque se pueden añadir CDB adicionales, cada CDB debe interaccionar con todo el hardware del procesador, incluyendo las estaciones de reserva. En particular, el hardware asociativo de emparejamiento de etiquetas necesitará duplicarse en todas las estaciones para cada CDB.

Aunque el esquema de Tomasulo puede ser atractivo si el diseñador está forzado a segmentar una arquitectura para la que sea difícil planificar el código o tenga escasez de registros, los autores piensan que las ventajas del enfoque de Tomasulo están limitadas para arquitecturas que se puedan segmentar eficientemente y planificar estáticamente con software. Sin embargo, cuando el número de puertas disponibles crece y se alcanzan los límites de la planificación software, podemos comprender que se emplee la planificación dinámica. Una posible dirección es una organización híbrida que utilice planificación dinámica para cargas y almacenamientos, aunque con planificación estática de operaciones registro-registro.

Reducción de las penalizaciones de saltos con predicción dinámica hardware

La sección anterior describe técnicas para resolver los riesgos por dependencias de datos. Si no se tratan los riesgos de control, la Ley de Amdahl predice, que limitarán el rendimiento de la ejecución segmentada. Antes, examinamos esquemas hardware sencillos para tratar los saltos (suponiéndolos efectivos o no efectivos), y enfoques orientados al software (saltos retardados). Esta sección se centra en utilizar el hardware para predecir dinámicamente el resultado de un salto —la predicción cambiará si los saltos cambian su comportamiento mientras se está ejecutando el programa.

El esquema dinámico de predicción de saltos más simple es un *buffer de predicción de saltos.* Un buffer de predicción de saltos es una pequeña memoria indexada por la parte menos significativa de la dirección de la instrucción de salto. La memoria contiene un bit que indica si recientemente el salto fue efectivo o no. Este es el tipo más sencillo de buffer; no tiene etiquetas y es útil sólo para reducir el retardo de salto cuando es mayor que el tiempo para calcular los posibles PC del destino. De hecho, no sabemos si la predicción es correcta —puede haber sido puesto allí por otro salto que tenga iguales los bits menos significativos de la dirección—. Pero esto no importa. Se supone que es correcta, y comienza la búsqueda en la dirección predicha. Si la predicción de salto resulta ser errónea, se invierte el bit de predicción.

Este esquema de predicción, de un solo bit, tiene un problema de rendi-

miento: si un salto es efectivo casi siempre, cuando no lo sea, lo predeciremos incorrectamente dos veces, en lugar de una. Considerar un salto en un bucle cuyo comportamiento es: es efectivo secuencialmente nueve veces, y no es efectivo una vez. Si la siguiente vez se predice como no efectivo, la predicción será errónea. Entonces, la precisión de la predicción será sólo del 80 por 100, aun en saltos el 90 por 100 efectivos. Para remediar esto, se utilizan, con frecuencia, esquemas de predicción de dos bits. En un esquema de dos bits, una predicción debe errar dos veces antes de que se cambie. La Figura 6.41 muestra la máquina de estados finitos para el esquema de predicción de dos bits.

El buffer de predicción de saltos se puede implementar como una pequeña cache especial accedida por la dirección de la instrucción durante la etapa IF de la segmentación, o como un par de bits vinculados a cada bloque de la cache de instrucciones y buscados con la instrucción (ver Sección 8.3 del Capítulo 8). Si la instrucción se predice como un salto y si el salto se predice como efectivo, la búsqueda del destino comienza tan pronto como se conozca el PC. En cualquier otro caso, continúa la búsqueda y la ejecución secuencial. Si la predicción resulta ser errónea, se cambian los bits de predicción, como muestra la Figura 6.41. Aunque este esquema es útil para muchas segmentaciones, la segmentación de DLX averigua al mismo tiempo si el salto es efectivo y el destino del salto. Por tanto, este esquema no ayuda a la sencilla segmentación de DLX; algo más tarde exploraremos un esquema que pueda funcionar para DLX. Primero, veamos cómo funciona un buffer de predicción con una segmentación más larga.

La precisión de un esquema de predicción de dos bits está afectada por el número de veces que la predicción es correcta para cada salto, y por el número de veces que la entrada en el buffer de predicción coincide con el salto

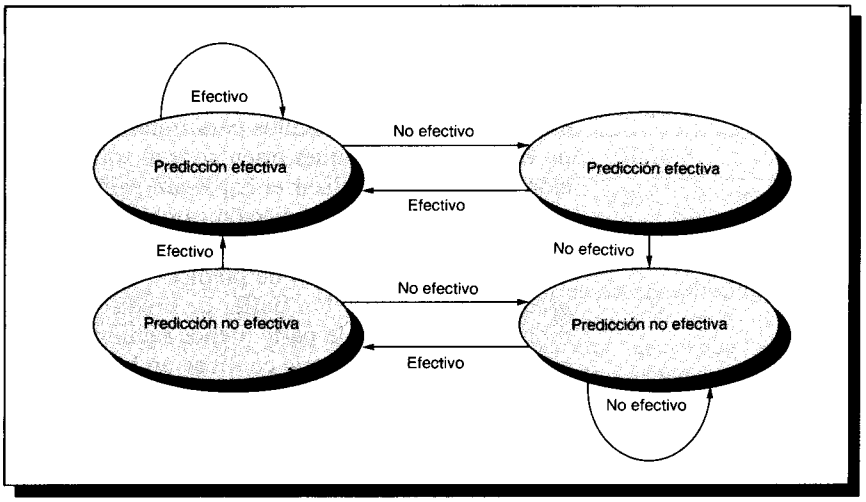


FIGURA 6.41 Esquema de predicción con dos bits de estado. Utilizando dos bits en lugar de uno, un salto que la mayoría de las veces sea efectivo o no efectivo —como ocurre con muchos saltos— estará mal predicho sólo una vez. Se utilizan dos bits para codificar los cuatro estados del sistema.

que se está ejecutando. Cuando la entrada no coincide, el bit de predicción se utiliza de todas formas porque no hay mejor información disponible. Aun cuando la entrada fuese para otro salto, la suposición podría haber tenido éxito. En efecto, aproximadamente, hay un 50 por 100 de probabilidad de que la predicción sea correcta, aunque la predicción sea para otro salto. Estudios sobre esquemas de predicción de saltos han encontrado que la predicción de dos bits tiene una precisión de, aproximadamente, el 90 por 100, cuando la entrada del buffer es la entrada del salto. Un buffer de entre 500 y 1 000 entradas tiene una frecuencia de aciertos del 90 por 100. La precisión de la predicción global está dada por

$$\text{Precisión} = (\%\ \text{correctamente predicho} \cdot \%\ \text{predicción es para esta instruccion}) +$$

$$+ (\%\ \text{conjetura afortunada}) \cdot (1 - \%\ \text{predicción es para esta instrucción})$$

$$\text{Precisión} = (90\ \% \cdot 90\ \%) + (50\ \% \cdot 10\ \%) = 86\ \%$$

Este número es mayor que el de nuestra frecuencia de aciertos para saltos retardados y sería útil en una segmentación con un retardo de salto mayor. Ahora, veamos un esquema de predicción dinámica que es utilizable para DLX y veremos cómo se compara con nuestro esquema de retardo de salto.

Para reducir la penalización de saltos en DLX, necesitamos conocer qué dirección buscar al final de IF. Esto significa que debemos conocer si la instrucción todavía no decodificada es un salto y, si lo es, cuál será el siguiente PC. Si la instrucción es un salto y sabemos cuál es el siguiente PC, podemos tener una penalización de salto cero. Una cache de predicción de saltos que almacena la dirección predicha de la siguiente instrucción después del salto se denomina un *buffer de destinos de saltos* (*branch-target buffer*). Como estamos prediciendo la dirección de la siguiente instrucción y la enviaremos **antes** de decodificar la instrucción, **debemos** saber si la instrucción buscada se predice como un salto efectivo. También queremos saber si la dirección del buffer de destinos es para una predicción de salto efectivo o no efectivo, para que podemos reducir el tiempo para determinar un salto mal predicho. La Figura 6.42 muestra el aspecto el buffer de destinos de saltos. Si el PC de la instrucción buscada coincide con un PC en el buffer, entonces el correspondiente PC predicho se utiliza como PC siguiente. En el Capítulo 8 explicaremos las caches con mucho más detalle; veremos que el hardware para este buffer de destinos de saltos es similar al hardware para una cache.

Si en el buffer de destinos de saltos se encuentra una entrada que coincide, la búsqueda comienza inmediatamente en el PC predicho. Observar que (de forma distinta a un buffer de predicción de saltos), la entrada debe ser para esta instrucción, porque el PC predicho se enviará fuera incluso antes que conozca si esta instrucción es un salto. Si no comprobamos si la entrada coincide con este PC, entonces el PC erróneo podría ser utilizado por instrucciones que no sean saltos, dando como resultado una máquina más lenta. La Figura 6.43 muestra los pasos seguidos cuando se utiliza un buffer de destinos de saltos y cuándo se presentan estos pasos en la segmentación. A partir de esto podemos ver que no habrá retardo de salto si en el buffer se encuentra una entrada de predicción de salto y es correcta. En cualquier otro caso, habrá

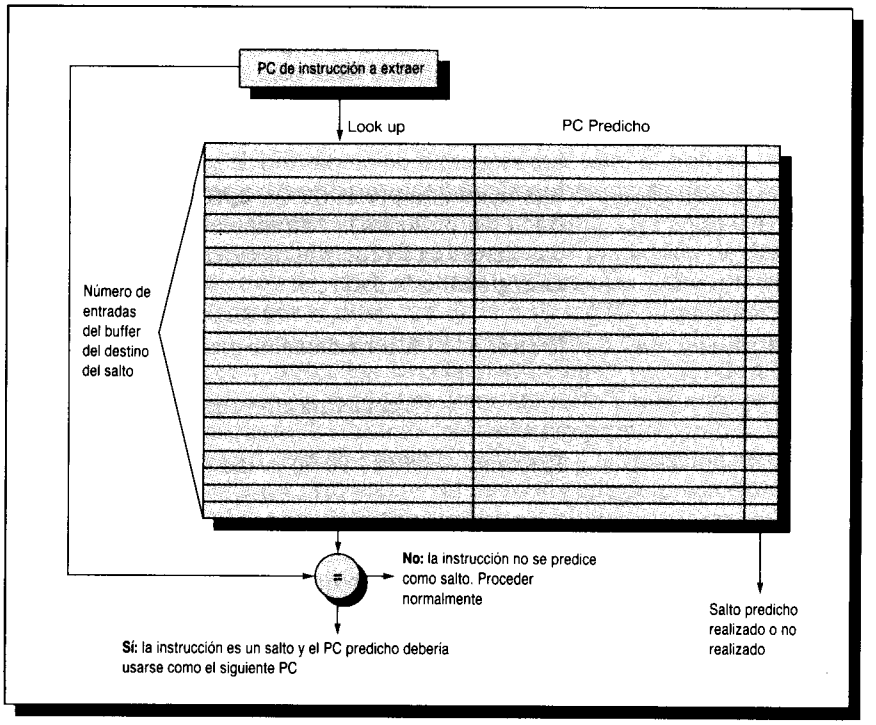


FIGURA 6.42 Un buffer de destinos de saltos. El PC de la instrucción que se está buscando se compara con un conjunto de direcciones de instrucciones almacenadas en la primera columna; éstas representan direcciones de saltos conocidos. Si el PC coincide con una de estas entradas, entonces la instrucción que se está buscando es un salto. Si es un salto, entonces el segundo campo, PC predicho, contiene la predicción para el siguiente PC después del salto. Las búsquedas comienzan inmediatamente en esa dirección. El tercer campo, exactamente, informa si el salto se predijo como efectivo o no efectivo y ayuda a mantener pequeña la penalización en caso de mala predicción.

una penalización de, al menos, un ciclo de reloj. En la práctica, habrá una penalización de dos ciclos de reloj, porque se debe actualizar el buffer de destinos de saltos. Podemos suponer que la instrucción que sigue a un salto o la del destinos de salto no es un salto, y hacer la actualización durante ese tiempo de instrucción. Sin embargo, esto complica el control. Por ello, tomaremos una penalización de dos ciclos de reloj cuando el salto no se prediga correctamente.

Para evaluar cómo funciona un buffer de destinos de saltos, primero debemos determinar qué penalizaciones hay en todos los casos posibles. La Figura 6.44 contiene esta información.

Utilizando las mismas probabilidades que para un buffer de predicción-de-salto —90 por 100 de probabilidad de encontrar la entrada y 90 por 100 de probabilidad de predicción correcta— y el porcentaje de efectivos/no efec-

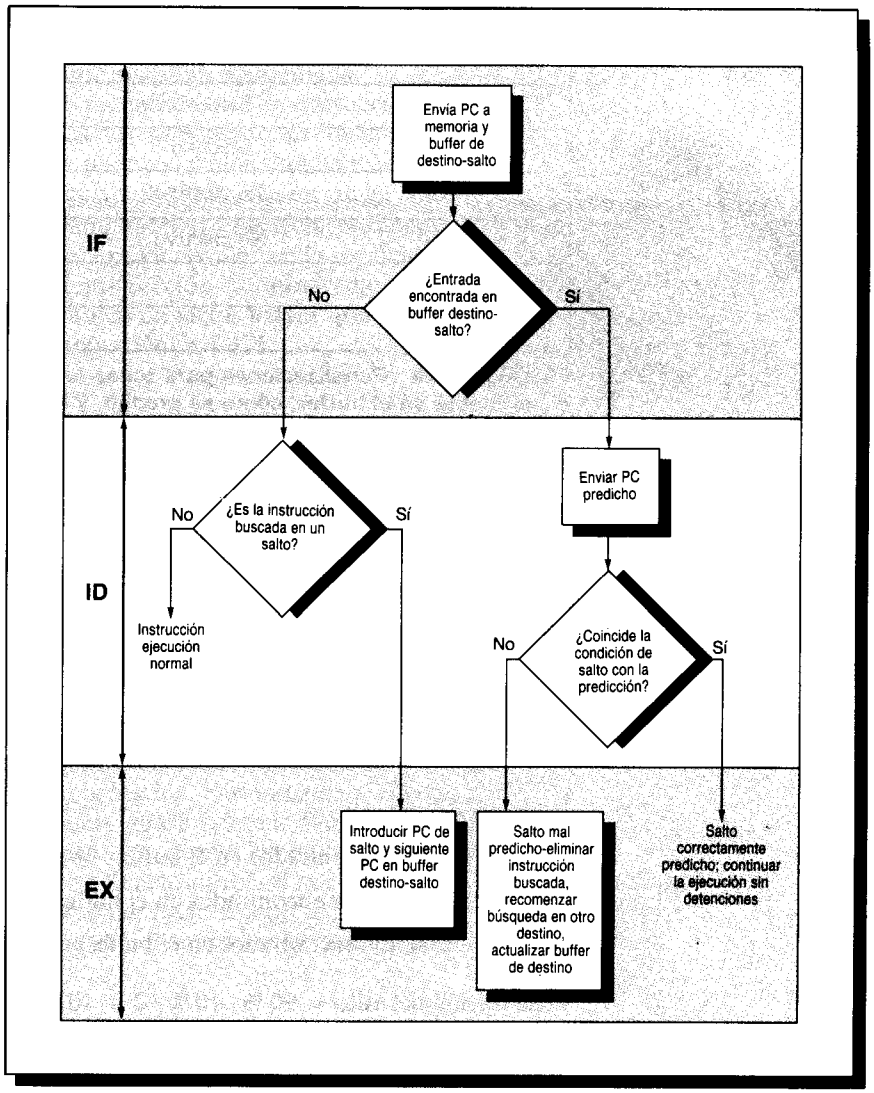


FIGURA 6.43 Los pasos involucrados en la manipulación de una instrucción con un buffer destinos de saltos. Si el PC de una instrucción se encuentra en el buffer, entonces la instrucción debe ser un salto, y la búsqueda comienza inmediatamente desde el PC predicho en ID. Si no se encuentra la entrada y posteriormente resulta ser un salto, se incluye en el buffer junto al destino, que es conocido al final de ID. Si la instrucción es un salto, se encuentra y se predice correctamente, entonces la ejecución procede sin retardos. Si la predicción es incorrecta, sufrimos un retardo de un ciclo de reloj buscando la instrucción errónea y recomenzando la búsqueda un ciclo de reloj más tarde. Si el salto no se encuentra en el buffer y la instrucción resulta ser un salto, también procederemos como si la instrucción fuese un salto y podemos convertir esto en una estrategia de asumir-no-efectivo; la penalización diferirá dependiendo de que el salto sea realmente efectivo o no.

Instrucción en bufer	Predicción	Salto real	Ciclos de penalización
Sí	Efectivo	Efectivo	0
Sí	Efectivo	No efectivo	2
Sí	No efectivo	No efectivo	0
Sí	No efectivo	Efectivo	2
No		Efectivo	2
No		No efectivo	1

FIGURA 6.44 Penalizaciones para todas las posibles combinaciones de si el salto está en el buffer, cómo se predijo, y lo que ocurre realmente. No hay penalización de salto si todo se predice correctamente y el salto se encuentra en el buffer de destinos. Si el salto no se predice correctamente, la penalización es de un ciclo de reloj para actualizar el buffer con la información correcta (durante la cual no se puede buscar ninguna instrucción) y un ciclo de reloj, si se necesita, para recomenzar la búsqueda de la siguiente instrucción correcta para el salto. Si el salto no se encuentra y no es efectivo, la penalización es sólo un ciclo de reloj porque el procesador supone que no es efectivo cuando no está enterado que la instrucción es un salto. Otros fallos de coincidencia cuestan dos ciclos de reloj, ya que debemos recomenzar la búsqueda y actualizar el buffer.

tivos tomado anteriormente en este capítulo, podemos determinar la penalización total del salto:

Penalización del salto =

= % saltos encontrados en el buffer · % predicciones incorrectas · 2 +

+ (1 − % saltos encontrados en el buffer) · % saltos realizados · 2 +

(1 − % saltos encontrados en el buffer) · % saltos no realizados · 1

Penalización del salto = 90 % · 10 % · 2 + 10 % · 60 % · 2 + 10 % · 40 % · 1

Penalización del salto = 0,34 ciclos de reloj

Esto es comparable con la penalización para saltos retardados que es, aproximadamente, 0,5 ciclos de reloj por salto. Recordar, sin embargo, que la mejora para predicción dinámica de saltos crece al aumentar el retardo de los saltos.

Los esquemas de predicción de saltos están limitados por la precisión de la predicción y por la penalización de las predicciones erróneas. Es improbable que podamos mejorar el éxito de predicción de saltos más del 80 o 90 por 100. En cambio, podemos tratar de reducir la penalización para predicciones erróneas. Esto se hace buscando la dirección predicha y la no predicha. Esto requiere que el sistema de memoria con dos puertos tenga una cache intercalada. Aunque esto añade coste al sistema, puede ser la única forma de reducir las penalizaciones de los saltos por debajo de un cierto punto.

Hemos visto diversos esquemas estáticos basados en software y esquemas dinámicos basados en hardware para tratar de aumentar el rendimiento de nuestra máquina segmentada. La segmentación intenta explotar el paralelismo potencial entre las instrucciones secuenciales. El caso ideal sería que todas las instrucciones fuesen independientes, y nuestro procesador segmentado DLX podría explotar simultáneamente el paralelismo entre las cinco instrucciones simultáneamente en curso. Ambas técnicas de planificación estática de la última sección y las técnicas dinámicas de esta sección se centran en mantener la productividad de la segmentación en una instrucción por ciclo. En la siguiente sección examinaremos técnicas que intenten explotar el solapamiento en un factor mayor que 5, al cual estamos restringidos con la sencilla segmentación de DLX.

6.8 Segmentación avanzada: aprovechando más el paralelismo de nivel de instrucción

Para mejorar más el rendimiento deberíamos decrementar el CPI a menos de la unidad. Pero el CPI no puede reducirse por debajo de uno si sólo emitimos una instrucción cada ciclo de reloj. El objetivo de las técnicas explicadas en esta sección es permitir emitir múltiples instrucciones en un ciclo de reloj.

Como sabemos de secciones anteriores, para mantener el procesador a pleno rendimiento, se debe explotar el paralelismo entre las instrucciones buscando secuencias de instrucciones no relacionadas que se puedan solapar en la segmentación. Dos instrucciones relacionadas deben estar separadas por una distancia igual a la latencia de la segmentación de la primera de las instrucciones. En esta sección supondremos las latencias de la Figura 6.45. Los saltos todavía tienen un retardo de un ciclo de reloj. Suponemos que las unidades funcionales están completamente segmentadas o replicadas, y que se puede emitir una operación cada ciclo de reloj.

Instrucción que produce resultado	Instrucción destino	Latencia en ciclos
op FP ALU	Otra op FP ALU	3
op FP ALU	almacenamiento doble	2
carga doble	op FP ALU	1
carga doble	almacenamiento doble	0

FIGURA 6.45 Latencias de operaciones utilizadas en esta sección. La primera columna muestra el tipo de instrucción que origina la latencia. La segunda columna es el tipo de instrucción que hace efectiva esa latencia. La última columna es la separación en ciclos de reloj para evitar una detención. Estos números son similares a las latencias medias que vimos en una unidad FP, como la que describimos para DLX en la Figura 6.29.

Cuando tratemos de ejecutar más instrucciones en cada ciclo de reloj y tratemos de solapar más instrucciones, necesitaremos encontrar y explotar más el paralelismo a nivel de instrucción. Entonces, antes de examinar organizaciones de la segmentación que requieran más paralelismo entre instrucciones, examinemos una técnica sencilla de compilación que ayudará a crear paralelismo adicional.

Incrementar el paralelismo a nivel de instrucción con desenrollamiento de bucles

Para comparar los enfoques explicados en esta sección, utilizaremos un bucle que suma un valor escalar a un vector en memoria. El código de DLX, no teniendo en cuenta la segmentación, sería:

```
Loop: LD    F0,0(R1)   ; carga el elemento del vector
      ADDD  F4,F0,F2   ; suma el escalar de F2
      SD    0(R1),F4   ; almacena el elemento del vector
      SUB   R1,R1,#8   ; decrementa el puntero en
                         8 bytes (por DW)
      BNEZ  R1,LOOP    ; salta cuando no es cero
```

Por simplicidad, suponemos que el array comienza en la posición 0. Si estuviese cargado en cualquier otro lugar, el bucle requeriría una instrucción adicional entera.

Comencemos viendo cómo ejecutará el bucle cuando se planifica en una segmentación sencilla para DLX con las latencias antes explicadas.

Ejemplo

Mostrar cómo sería en DLX el bucle de suma del vector, planificado y no planificado, incluyendo todas las detenciones o ciclos de reloj inactivos.

Respuesta

Sin planificación el bucle se ejecutaría como sigue:

```
                                  Emitida en ciclo
                                    de reloj
Loop: LD    F0,0(R1)                  1
      detención                       2
      ADDD  F4,F0,F2                  3
      detención                       4
      detención                       5
      SD    0(R1),F4                  6
      SUB   R1,R1,#8                  7
      BNEZ  R1,LOOP                   8
      detención                       9
```

Esto requiere 9 ciclos de reloj por iteración. Podemos planificar el bucle para obtener

```
Loop: LD    F0,0(R1)
      detención
      ADDD  F4,F0,F2
      SUB   R1,R1,#8
      BNEZ  R1,LOOP     ; salto retardado
      SD    8(R1),F4    ; cambiado a causa del intercambio con SUB
```

El tiempo de ejecución se ha reducido de 9 a 6 ciclos de reloj.

Observar que para crear esta planificación, el compilador había de determinar que podía intercambiar SUB y SD cambiando la dirección de SD: la dirección era 0(R1) y ahora es 8(R1). Esto no es trivial, ya que la mayoría de los compiladores verían que la instrucción SD depende de SUB y rechazarían el intercambiarlas. Un compilador más inteligente calcularía la relación y realizaría el intercambio. La dependencia entre LD, ADDD y SD determina el número de ciclos de reloj para este bucle.

En el ejemplo anterior, completamos una iteración del bucle y terminamos un elemento del vector cada 6 ciclos de reloj, pero el trabajo real de operación sobre el elemento del vector necesita exactamente 3 de los 6 ciclos de reloj. Los 3 ciclos de reloj restantes se emplean en gastos del bucle —SUB y BNEZ— y en una detención. Para eliminar estos 3 ciclos de reloj necesitamos obtener más operaciones en el bucle. Un sencillo esquema para incrementar el número de instrucciones entre las ejecuciones de los saltos del bucle *es desenrollar el bucle* (*loop unrolling*). Esto se hace replicando múltiples veces el cuerpo del bucle, ajustando su código de terminación y planificando entonces el bucle desenrollado. Para lograr una planificación efectiva, utilizaremos diferentes registros para cada iteración, incrementando así el número de registros.

Ejemplo

Mostrar el bucle anterior desenrollado tres veces (manteniendo cuatro copias del cuerpo del bucle), suponiendo que R1, inicialmente, es un múltiplo de 4. Eliminar cualquier cálculo obviamente redundante, y no reusar ningún registro.

Respuesta

Aquí está el resultado de eliminar las operaciones innecesarias SUB y BNEZ duplicadas durante el desenrollamiento.

```
Loop:  LD     F0,0(R1)
       ADDD   F4,F0,F2
       SD     0(R1),F4 se eliminan SUB & BNEZ
       LD     F6,-8(R1)
       ADDD   F8,F6,F2
       SD     -8(R1),F8 se eliminan SUB & BNEZ
```

```
LD      F10,-16(R1)
ADDD    F12,F10,F2
SD     -16(R1),F12 se eliminan SUB & BNEZ
LD      F14,-24(R1)
ADDD    F16,F14,F2
SD      -24(R1),F16
SUB     R1,R1,#32
BNEZ    R1,LOOP
```

Hemos eliminado tres saltos y tres decrementos de R1. Se han ajustado las direcciones de las cargas y almacenamientos. Sin planificación, cada operación va seguida por una operación dependiente, y, por tanto, provocará una detención. Este bucle se ejecutará en 27 ciclos de reloj —cada LD necesita 2 ciclos de reloj, cada ADDD 3, el salto 2, y las demás instrucciones 1—, o 6,8 ciclos de reloj para cada uno de los cuatro elementos.

Aunque esta versión desenrollada sea más lenta que la versión planificada del bucle original, esto cambiará cuando planifiquemos el bucle desenrollado. El desenrollamiento del bucle, normalmente, se hace al principio del proceso de compilación, para que los cálculos redundantes puedan ser expuestos y eliminados por el optimizador.

En programas reales, normalmente, no conocemos el límite superior del bucle. Supongamos que es n, y que nos gustaría desenrollar el bucle k veces. En lugar de un solo bucle desenrollado, generamos un par de bucles. El primero se ejecuta (n mod k) veces y tiene un cuerpo que es el del bucle original. La versión desenrollada del bucle está circundada por un bucle externo que se itera (n div k) veces. En el ejemplo anterior, el desenrollamiento mejora el rendimiento de este bucle eliminando instrucciones que producen un gasto extra, aunque incrementa sustancialmente el tamaño de código. ¿Qué ocurrirá con el aumento de rendimiento cuando el bucle se planifique en DLX?

Ejemplo

Mostrar el bucle desenrollado del ejemplo anterior una vez que haya sido planificado en DLX.

Respuesta

```
Loop:  LD      F0,0(R1)
       LD      F6,-8(R1)
       LD      F10,-16(R1)
       LD      F14,-24(R1)
       ADDD    F4,F0,F2
       ADDD    F8,F6,F2
       ADDD    F12,F10,F2
       ADDD    F16,F14,F2
       SD      0(R1),F4
       SD      -8(R1),F8
```

```
SD      -16(R1),F12
SUB     R1,R1,#32   ; dependencia de salto
BNEZ    R1,LOOP
SD      8(R1),F16 ; 8-32 = -24
```

El tiempo de ejecución del bucle desenrollado ha caído a un total de 14 ciclos de reloj, o 3,5 ciclos de reloj por elemento, comparado con 6,8 por elemento antes de la planificación.

La ganancia de planificación sobre el bucle desenrollado es aún mayor que en el original. Esto es porque, al desenrollar el bucle, se descubre más cálculo que el que se puede planificar. Planificar el bucle de esta forma exige darse cuenta que las instrucciones de carga y almacenamiento son independientes y se pueden intercambiar.

Desenrollar el bucle es un método sencillo pero útil para incrementar el tamaño de los fragmentos de código lineal que pueden ser planificados efectivamente. Esta transformación en tiempo de compilación es similar a la que hace el algoritmo de Tomasulo con el renombramiento de registros y la ejecución fuera de orden. Como veremos, esto es muy importante al intentar disminuir el CPI emitiendo instrucciones a elevada frecuencia.

Una versión superescalar de DLX

Un método para disminuir el CPI de DLX es emitir más de una instrucción por ciclo de reloj. Esto permitirá que la frecuencia de ejecución de instrucciones exceda a la frecuencia de reloj. Las máquinas que emiten múltiples instrucciones independientes por ciclo de reloj, cuando están planificadas adecuadamente por el compilador, se denominan *máquinas superescalares*. En una máquina superescalar, el hardware puede emitir un pequeño número (por ejemplo, de 2 a 4) de instrucciones independientes en un solo ciclo. Sin embargo, si las instrucciones del flujo de instrucciones son dependientes o no cumplen ciertos criterios, sólo se emitirá la primera instrucción de la secuencia. Una máquina donde el compilador tenga completa responsabilidad para crear un paquete de instrucciones que se puedan emitir simultáneamente, y el hardware no tome dinámicamente decisiones sobre múltiples emisiones, probablemente debería considerarse del tipo VLIW (palabra de instrucción muy larga —*very long instruction word*—), que explicamos en la siguiente sección.

¿Qué hacer para que la máquina DLX parezca un superescalar? Supongamos que se emiten dos instrucciones por ciclo de reloj. Una de las instrucciones podría ser de carga, almacenamiento, salto u operación entera de la ALU, y la otra podría ser cualquier operación de punto flotante. Como veremos, emitir una operación entera en paralelo con una operación en punto flotante es mucho más simple y menos exigente que emitir dos instrucciones cualesquiera.

Emitir dos instrucciones por ciclo requerirá buscar y decodificar 64 bits de instrucciones. Para lograr una decodificación sencilla, necesitaríamos que las instrucciones estuviesen emparejadas y alineadas sobre un límite de 64 bits,

Tipo de instrucción	Cauce	Etapas						
Instrucción entera	IF	ID	EX	MEM	WB			
Instrucción FP	IF	ID	EX	MEM	WB			
Instrucción entera		IF	ID	EX	MEM	WB		
Instrucción FP		IF	ID	EX	MEM	WB		
Instrucción entera			IF	ID	EX	MEM	WB	
Instrucción FP			IF	ID	EX	MEM	WB	
Instrucción entera				IF	ID	EX	MEM	WB
Instrucción FP				IF	ID	EX	MEM	WB

FIGURA 6.46 Segmentación superescalar en operación. Las instrucciones enteras y de punto flotante se emiten a la vez, y cada una se ejecuta a su propio ritmo a través de la segmentación. Este esquema sólo mejorará el rendimiento de programas con una cantidad respetable de punto flotante.

apareciendo primero la parte entera. La Figura 6.46 muestra cómo avanzan el par de instrucciones en la segmentación. Esta tabla no trata cómo las operaciones de punto flotante extienden el ciclo EX, pero el caso superescalar no es diferente del caso de la segmentación ordinaria de DLX; los conceptos de la Sección 6.6 se aplican directamente. Con segmentación, hemos aumentado sustancialmente la frecuencia a la cual se pueden emitir las instrucciones de punto flotante. Sin embargo, para hacer esto útil necesitamos unidades de punto flotante segmentadas o múltiples unidades independientes. En cualquier otro caso, las instrucciones de punto flotante pueden sólo ser buscadas, y no emitidas, ya que todas las unidades de punto flotante estarán ocupadas.

Al emitir en paralelo una operación en punto flotante y otra entera, se minimiza la necesidad de hardware adicional —las operaciones enteras y de punto flotante utilizan diferentes conjuntos de registros y diferentes unidades funcionales. El único conflicto surge cuando la instrucción entera es una carga, almacenamiento o transferencia de punto flotante. Esto crea contención para los puertos de los registros de punto flotante, y también puede crear un riesgo si la operación de punto flotante utiliza el resultado de una carga de punto flotante emitida al mismo tiempo. Ambos problemas pueden resolverse detectando esta contención como un riesgo estructural y retrasando la emisión de la instrucción de punto flotante. La contención también se puede eliminar suministrando dos puertos adicionales, uno de lectura y otro de escritura, en el fichero de registros de punto flotante. También sería necesario añadir algunos caminos de desvío adicionales para evitar pérdida de rendimiento.

Hay otra dificultad que puede limitar la efectividad de una segmentación superescalar. En nuestra segmentación básica de DLX, las cargas tienen una latencia de un ciclo de reloj; esto impedía que una instrucción utilizase el resultado sin detención. En la segmentación superescalar, el resultado de una instrucción de carga no se puede utilizar en el mismo ciclo de reloj ni en el siguiente. Esto significa que las tres instrucciones siguientes no pueden utilizar

el resultado de la carga sin detención; sin puertos extra, las transferencias entre los conjuntos de registros están afectadas de la misma manera. El retardo de los saltos también llega a ser tres instrucciones. Para explotar efectivamente el paralelismo disponible en una máquina superescalar, necesitaremos implementar técnicas más ambiciosas de planificación del compilador, así como una decodificación más compleja de instrucciones. Desenrollar el bucle ayuda a generar mayores fragmentos lineales para su planificación; casi al final de esta sección se explican técnicas de compilación más potentes.

Veamos cómo funciona la planificación y el desenrollamiento de bucles en una versión superescalar de DLX con los mismos retardos en ciclos de reloj.

Ejemplo

¿Cómo se planificaría el bucle desenrollado de la página 340 en una segmentación superescalar para DLX? Para planificarlo sin retardos, necesitaremos desenrollarlo para hacer cinco copias del cuerpo.

Respuesta

El código resultante se muestra en la Figura 6.47.

	Instrucción entera		Instrucción FP	Ciclo de reloj
Loop:	LD	F0,0(R1)		1
	LD	F6,-8(R1)		2
	LD	F10,-16(R1)	ADDD F4,F0,F2	3
	LD	F14,-24(R1)	ADDD F8,F6,F2	4
	LD	F18,-32(R1)	ADDD F12,F10,F2	5
	SD	0(R1),F4	ADDD F16,F14,F2	6
	SD	-8(R1),F8	ADDD F20,F18,F2	7
	SD	-16(R1),F12		8
	SD	-24(R1),F16		9
	SUB	R1,R1,#40		10
	BNEZ	R1,LOOP		11
	SD	8(R1),F20		12

FIGURA 6.47 Código planificado y desenrollado como aparecería en un DLX superescalar.

Este bucle superescalar desenrollado se ejecuta ahora en 12 ciclos de reloj por iteración, o 2,4 ciclos de reloj por elemento, frente a 3,5 para el bucle planificado y desenrollado de la segmentación ordinaria de DLX. En este ejemplo, el rendimiento del DLX superescalar está limitado por el equilibrio entre los cálculos enteros y de punto flotante. Cada instrucción de punto flotante se emite junto con una instrucción entera, pero no hay suficientes instrucciones de punto flotante para mantener a pleno rendimiento el procesador de punto flotante. Cuando se planificó, el bucle original se ejecutaba a 6 ciclos de reloj por iteración. Hemos mejorado esto en un factor de 2,5, del que más de la mitad procede de desenrollar el bucle, que nos llevaba de 6 a 3,5, proviniendo el resto de emitir más de una instrucción por ciclo de reloj.

Idealmente, nuestra máquina superescalar tomará dos instrucciones y las emitirá si la primera es entera y la segunda de punto flotante. Si no cumplen este patrón, que puede detectarse rápidamente, entonces se distribuyen secuencialmente. Esto apunta una de las principales ventajas de una máquina superescalar general: hay poco impacto en la densidad de código y se pueden ejecutar programas incluso sin planificar. El número de emisiones y clases de instrucciones que se pueden emitir juntas son los factores más importantes que diferencian a los procesadores superescalares.

Emisión de múltiples instrucciones con planificación dinámica

El emitir múltiples instrucciones también se puede aplicar a máquinas planificadas dinámicamente. Comenzaremos con el esquema del marcador o con el algoritmo de Tomasulo. Supongamos que queremos extender el algoritmo de Tomasulo para que soporte la emisión de dos instrucciones por ciclo de reloj, una entera y otra en punto flotante. No queremos emitir instrucciones de la cola fuera de orden, ya que esto hace imposible la contabilidad del fichero de registros. Mejor dicho, empleando estructuras de datos para los registros entero y de punto flotante, ambos tipos de instrucciones pueden ser emitidas a sus respectivas estaciones de reserva, siempre que las dos instrucciones de la cabeza de la cola de instrucciones no accedan al mismo conjunto de registros. Desgraciadamente, este enfoque impide emitir dos instrucciones con una dependencia en el mismo ciclo de reloj. Esto es, por supuesto, cierto en el caso superescalar, donde claramente, es el problema del compilador. Hay tres enfoques que se pueden utilizar para lograr una emisión doble. Primero, podemos utilizar planificación software para asegurar que las instrucciones dependientes no aparezcan juntas. Sin embargo, esto requeriría software de planificación de la segmentación, contrarrestando de ese modo una de las ventajas de las segmentaciones planificadas dinámicamente.

Un segundo enfoque es segmentar la etapa de emisión de instrucciones para que corra dos veces más rápida que la frecuencia básica del reloj. Esto permite actualizar las tablas antes de tratar la siguiente instrucción; entonces la dos instrucciones pueden comenzar la ejecución a la vez.

El tercer enfoque está basado en la observación de que, si no se emiten múltiples instrucciones a la misma unidad funcional, entonces sólo serán las instrucciones de carga y almacenamiento las que creen dependencias entre las instrucciones que deseemos emitir juntas. La necesidad de tablas de reserva para cargas y almacenamientos se puede eliminar utilizando colas para el resultado de una carga y para el operando fuente de un almacenamiento. Como la planificación dinámica es más efectiva para cargas y almacenamientos, mientras que la planificación estática es altamente efectiva en secuencias de código de registro-registro, podremos utilizar planificación estática para eliminar completamente las estaciones de reserva y contar con las colas para cargas y almacenamientos. Este estilo de organización de la máquina se ha denominado *arquitectura desacoplada (decompled architecture)*.

Por simplicidad, supongamos que hemos segmentado la lógica de emisión

de instrucciones para que podamos emitir dos operaciones que son dependientes pero que utilizan unidades funcionales diferentes. Veamos cómo funcionará esto con nuestro ejemplo.

Ejemplo

Considerar la ejecución de nuestro bucle en un procesador segmentado DLX extendido con el algoritmo de Tomasulo y con emisiones múltiples. Suponer que las operaciones enteras y en punto flotante se pueden emitir en cada ciclo de reloj, aunque estén relacionadas. El número de ciclos de latencia por instrucción es el mismo. Suponer que la emisión y la escritura de resultados necesita un ciclo de reloj cada una, y que hay hardware de predicción dinámica de saltos. Crear una tabla mostrando cuándo se emite cada instrucción, comienza la ejecución, y escribe su resultado, para las dos primeras iteraciones del bucle. Aquí está el bucle original:

```
Loop:  LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    0(R1),F4
       SUB   R1,R1,#8
       BNEZ  R1,LOOP
```

Respuesta

El bucle se desenrollará dinámicamente y, siempre que sea posible, las instrucciones se distribuirán en pares. El resultado se muestra en la Figura 6.48.

El bucle corre en $4 + \frac{7}{n}$ ciclos de reloj por resultado para n iteraciones. Para un n grande esto tiende a 4 ciclos de reloj por resultado.

Número de iteración	Instrucciones	Se distribuye en el ciclo de reloj número	Se ejecuta en el ciclo de reloj número	Escribe el resultado en el ciclo de reloj número
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUB R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUB R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

FIGURA 6.48 El instante de emitir, ejecutar y escribir resultado para una versión de doble emisión de nuestra segmentación de Tomasulo. La etapa de escribir resultado no se aplica a los almacenamientos ni saltos, ya que no escriben ningún registro.

El número de emisiones dobles es pequeño porque solamente hay una operación en punto flotante por iteración. El número relativo de emisiones dobles puede mejorarse si el compilador desenrolla el bucle para reducir el número de instrucciones eliminando gastos del bucle. Con esa transformación, el bucle se ejecutará tan rápido como en una máquina superescalar. Volveremos a esta transformación en los Ejercicios 6.16 y 6.17.

El enfoque VLIW

Nuestra máquina superescalar DLX puede emitir dos instrucciones por ciclo de reloj. Eso quizá podría extenderse a tres o, como máximo, cuatro, pero llega a ser difícil determinar si se pueden emitir tres o cuatro instrucciones, simultáneamente, sin saber en qué orden estarán las instrucciones cuando se busquen de memoria y qué dependencias pueden existir entre ellas. Una alternativa es una arquitectura LIW (*Palabra de Instrucción Larga —Long Instruction Word—*) o VLIW (*Palabra de Instrucción Muy Larga —Very Long Instruction Word—*). Las VLIW utilizan múltiples unidades funcionales independientes. En lugar de intentar emitir múltiples instrucciones independientes a las unidades, una VLIW empaqueta múltiples operaciones en una instrucción muy larga, de ahí el nombre. Una instrucción VLIW podría incluir dos operaciones enteras, dos operaciones en punto flotante, dos referencias a memoria, y un salto. Una instrucción tendrá un conjunto de campos para cada unidad funcional —quizá de 16 a 24 bits por unidad, dando una longitud de instrucción entre 112 y 168 bits—. Para mantener ocupadas las unidades funcionales debe haber suficiente trabajo en una secuencia de código lineal para mantener planificadas las instrucciones. Esto se realiza desenrollando bucles y planificando código a través de bloques básicos, utilizando una técnica llamada *planificación de trazas (trace scheduling)*. Además de eliminar saltos desenrollando bucles, la planificación de rastros proporciona un método para mover instrucciones a través de puntos de salto. Explicaremos esto con más detalle en la sección siguiente. Por ahora, supongamos que tenemos una técnica para generar largas secuencias de código lineal para construir instrucciones VLIW.

Ejemplo

Suponer que tenemos una VLIW que pueda emitir dos referencias a memoria, dos operaciones FP y una operación entera o salto en cada ciclo de reloj. Mostrar una versión desenrollada del bucle de suma de vectores para esa máquina. Desenrollar las veces que sea necesario para eliminar cualquier detención. Ignorar los huecos de retardo de salto.

Respuesta

El código se muestra en la Figura 6.49. El bucle se ha desenrollado seis veces, lo que elimina detenciones, y se ejecuta en 9 ciclos. Esto proporciona una frecuencia de ejecución de 7 resultados en 9 ciclos, o 1,28 ciclos por resultado.

Referencia a memoria 1	Referencia a memoria 2	Operación FP 1	Operación FP2	Operación entera salto
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2	ADDD F24,F22,F2	
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2		
SD -16(R1),F12	SD -24(R1),F16			
SD -32(R1),F20	SD -40(R1),F24			SUB R1,R1,#48
SD -0(R1),F28				BNEZ R1,LOOP

FIGURA 6.49 Instrucciones VLIW que ocupan el bucle interno y sustituyen la secuencia desenrollada. Este código emplea nueve ciclos suponiendo que no hay retardos debido a saltos; normalmente el salto se debería planificar también. La frecuencia de emisión es de 23 operaciones en 9 ciclos de reloj, o 2,5 operaciones por ciclo. La eficiencia, el porcentaje de los huecos disponibles que contiene una operación, es aproximadamente el 60 por 100. Para lograr esta velocidad de emisión se requiere un número mucho mayor de registros del que DLX utilizaría normalmente en este bucle.

¿Cuáles son las limitaciones y costes de un enfoque VLIW? Si podemos emitir 5 operaciones por ciclo de reloj, ¿por qué no 50? Se han encontrado tres limitaciones diferentes: paralelismo limitado, recursos hardware limitados y explosión del tamaño del código. La primera es la más simple: hay una cantidad limitada de paralelismo disponible en las secuencias de instrucciones. A menos que los bucles se desenrollen gran número de veces, puede no haber suficientes operaciones para rellenar las instrucciones. En un primer vistazo, puede parecer que 7 instrucciones que pudieran ejecutarse en paralelo serían suficientes para mantener completamente ocupada nuestra VLIW. Sin embargo, éste no es el caso. Algunas de estas unidades funcionales —las unidades de memoria, de salto y de punto flotante— serán segmentadas, necesitando un número mucho mayor de operaciones que se puedan ejecutar en paralelo. Por ejemplo, si la segmentación de punto flotante tiene 8 pasos, las 2 operaciones que se emiten en un ciclo de reloj no pueden depender de ninguna de las 14 operaciones en curso en el procesador de punto flotante. Por ello, necesitamos encontrar un número de operaciones independientes aproximadamente igual a la profundidad media de la segmentación por el número de unidades funcionales. Esto significa que se necesitarán de unas 20 a 30 operaciones para mantener una VLIW con 7 unidades funcionales ocupadas.

El segundo coste, los recursos hardware para una VLIW, parece bastante claro; duplicar las unidades funcionales enteras de punto flotante es fácil y el coste aumenta linealmente. Sin embargo, hay un gran incremento en el ancho de banda del fichero de registros y de la memoria. Aun con un fichero de registros dividido en parte entera y de punto flotante, nuestra VLIW necesitará 7 puertos de lectura, 3 de escritura en el fichero de registros enteros y 5 puertos de lectura y 3 de escritura en el fichero de registros de punto flotante. Este ancho de banda no puede soportarse sin un coste sustancial en el tamaño del fichero de registros ni posible degradación de velocidad del reloj. Nuestra

VLIW de 7 unidades también tiene dos puertos de memoria de datos. Además, si queremos expandirla, necesitaremos continuar añadiendo puertos de memoria. Añadir solamente unidades aritméticas no ayudaría, ya que la máquina estaría limitada por el ancho de banda de memoria. Cuando crece el número de puertos de la memoria de datos, lo hace la complejidad del sistema de memoria. Para permitir múltiples accesos a memoria en paralelo, ésta se debe descomponer en bancos que contengan diferentes direcciones con la esperanza que las operaciones de una misma instrucción no tengan accesos conflictivos. Un conflicto puede hacer que se detenga la máquina entera, ya que todas las unidades funcionales deben mantenerse sincronizadas. Este mismo factor hace extremadamente difícil utilizar caches de datos en VLIW.

Finalmente, está el problema del tamaño de código. Hay dos elementos diferentes que se combinan para aumentar sustancialmente el tamaño del código. Primero, generar suficientes operaciones en un fragmento de código lineal requiere bucles ambiciosamente desenrollados, lo que incrementa el tamaño del código. Segundo, siempre que las instrucciones no sean completas, las unidades funcionales no utilizadas implican bits desaprovechados en la codificación de las instrucciones. En la Figura 6.49, vemos que se utilizaron aproximadamente el 60 por 100 de las unidades funcionales; casi la mitad de cada instrucción estaba vacía. Para combatir este problema algunas veces se utilizan codificaciones inteligentes. Por ejemplo, puede haber sólo un gran campo de inmediatos que lo utilice alguna unidad funcional. Otra técnica es comprimir las instrucciones en memoria principal y expandirlas cuando sean leídas en la cache o sean decodificadas.

El reto más importante para estas máquinas es tratar de explotar grandes cantidades de paralelismo a nivel de instrucción. Cuando el paralelismo proviene de desenrollar bucles, el bucle original, probablemente, se habría ejecutado eficientemente en una máquina vectorial (ver el siguiente capítulo). No está claro que sea preferible una VLIW sobre una máquina vectorial para estas aplicaciones; los costes son similares, y la máquina vectorial tiene normalmente la misma o mayor velocidad. La cuestión abierta en 1990 es si hay grandes clases de aplicaciones que no son aconsejables para máquinas vectoriales, pero ofrecen todavía suficiente paralelismo para justificar el enfoque VLIW en lugar de otra más simple, tal como una máquina superescalar.

Aumento del paralelismo a nivel de instrucción con segmentación software y planificación de trazas

Ya hemos visto que se utiliza una técnica de compilación, desenrollar bucles, para ayudar a explotar el paralelismo entre las instrucciones. Desenrollar bucles crea secuencias más largas de código, que se pueden utilizar para explotar más el paralelismo a nivel de instrucción. Para este propósito se han desarrollado otras dos técnicas más generales: la segmentación software y la planificación de trazas.

La segmentación software es una técnica para reorganizar bucles, de tal forma que, cada iteración en el código segmentado por software se haga a partir de secuencias de instrucciones escogidas en diferentes iteraciones del seg-

mento de código original. Esto se comprende más fácilmente examinando el código planificado para la versión superescalar de DLX. El planificador, esencialmente, intercala instrucciones de diferentes iteraciones de bucles, juntando todas las cargas, después todas las sumas, después todos los almacenamientos. Un bucle, segmentado por software, intercala instrucciones de diferentes iteraciones sin desenrollar el bucle. Esta técnica es la contrapartida software a lo que el algoritmo de Tomasulo hace en hardware. El bucle segmentado por software contendrá una carga, una suma y un almacenamiento, cada uno de una iteración diferente. También hay código de arranque que se necesita antes que comience el bucle, así como código para concluir una vez que se complete el bucle. Ignoramos éstos en esta discusión.

Ejemplo

Mostrar una versión segmentada por software de este bucle:

```
Loop:  LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    0(R1),F4
       SUB   R1,R1,#8
       BNEZ  R1,LOOP
```

Se pueden omitir los códigos de arranque y finalización

Respuesta

Dado el vector M de memoria e ignorando los códigos de arranque y terminación, tenemos:

```
Loop:  SD   0(R1),F4     ; almacena en M[i]
       ADDD F4,F0,F2     ; suma a M[i-1]
       LD   F0,-16(R1)   ; carga M[i-2]
       BNEZ R1,LOOP
       SUB  R1,R1,#8      ; resta en hueco de retardo
```

Este bucle se puede ejecutar a una velocidad de 5 ciclos por resultado, ignorando las partes de arranque y terminación. Como la carga busca dos elementos del array anteriores a la cuenta de elementos, el bucle deberá correr durante dos iteraciones menos. Esto se logrará decrementando R1 en 16 antes del bucle.

La segmentación software puede considerarse un desenrollamiento simbólico de bucles. En efecto, algunos de los algoritmos para segmentación software utilizan desenrollamiento de bucles para determinar cómo segmentar el bucle. La principal ventaja de la segmentación software sobre el desenrollamiento directo de bucles es que aquél consume menos espacio de código. La segmentación software y el desenrollamiento de bucles, además de lograr un bucle interno mejor planificado, cada uno reduce un tipo diferente de gasto. El desenrollamiento de bucles reduce gastos del bucle —código de saltos y de actualización de contadores. La segmentación software reduce la porción de tiempo en la que el bucle no se ejecuta a la velocidad máxima a una única vez

al comienzo y al final del bucle. Si desenrollamos un bucle que haga 100 iteraciones un número constante de veces, por ejemplo 4, incurrimos en gastos extras de 100/4 = 25 veces —cada vez que se reinicia el bucle interno desenrollado. La Figura 6.50 muestra gráficamente este comportamiento. Como estas técnicas acometen dos tipos de gasto, el mejor rendimiento se logra utilizando ambas.

La otra técnica utilizada para generar paralelismo adicional es la *planificación de trazas*. Esta es particularmente útil para arquitecturas VLIW, para las que se desarrolló originalmente la técnica. La planificación de trazas es una combinación de dos procesos separados. El primer proceso, llamado *selección de trazas* trata de encontrar la secuencia más probable de operaciones para juntarlas en un pequeño número de instrucciones; esta secuencia se denomina *traza*. El desenrollamiento de bucles se utiliza para generar trazas largas, ya que los saltos del bucle son efectivos con una probabilidad alta. Una vez que se selecciona una traza, el segundo proceso, denominado *compactación de trazas* trata de compactar la traza en un pequeño número de instrucciones anchas. La compactación de trazas intenta mover operaciones tan pronto como pueda en una secuencia (traza), empaquetando las operaciones en el mínimo número posible de instrucciones anchas.

Hay dos consideraciones diferentes al compactar una traza: las dependencias de los datos, que fuerzan un orden parcial sobre las operaciones, y los puntos de salto, que crean lugares a través de los cuales no se puede mover el

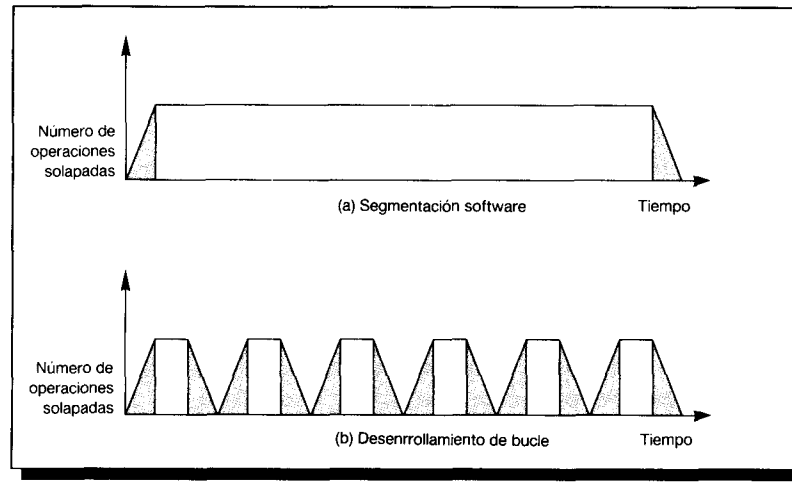


FIGURA 6.50 Patrón de ejecución para (a) un bucle segmentado por software y (b) un bucle desenrollado. Las áreas sombreadas son las veces que el bucle no se ejecuta con solapamiento o paralelismo máximo entre instrucciones. Esto ocurre una vez al comienzo del bucle y otra al final para el bucle segmentado por software. Para el bucle desenrollado ocurre m/n veces si el bucle tiene un total de m ejecuciones y se desenrolla n veces. Cada bloque representa un desenrollamiento de n iteraciones. Incrementando el número de desenrollamientos se reducirán los gastos de arranque y terminación.

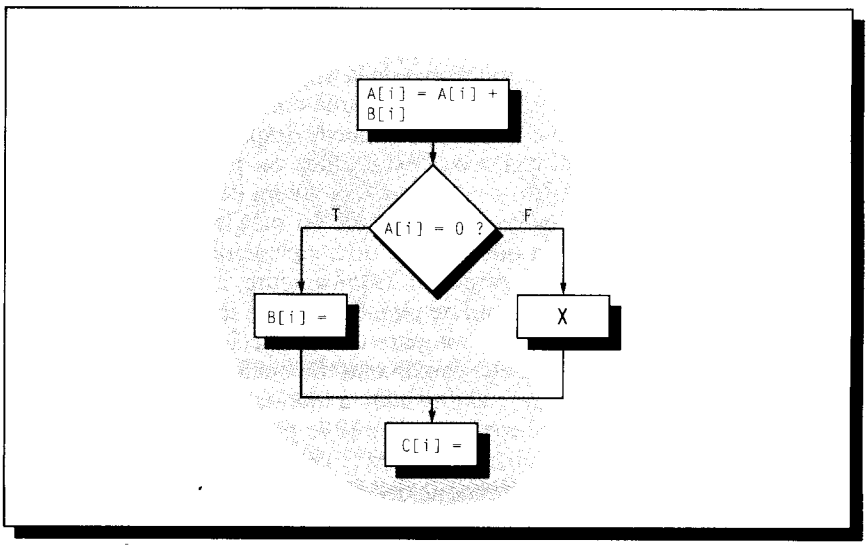


FIGURA 6.51 Un fragmento de código y la traza seleccionada sombreada en gris. Esta traza se seleccionaría primero si la probabilidad de que el salto sea efectivo es mucho mayor que la probabilidad de que no sea efectivo. La rama de la decisión (A[i] = 0) a X es una rama hacia afuera de la traza, y la rama desde X a la asignación de C es una rama hacia adentro de la traza. Estas ramas son las que hacen difícil la compactación de la traza.

código fácilmente. En esencia, se quiere compactar el código en la secuencia más corta posible que preserve las dependencias de los datos; los saltos son el principal impedimento para este proceso. La ventaja principal de la planificación de trazas, sobre las técnicas más sencillas de planificación de procesadores segmentados, es que incluye un método para mover código a través de los saltos. La Figura 6.51 muestra un fragmento de código que se puede considerar como una iteración de un bucle desenrollado y la traza seleccionada.

Una vez que se selecciona la traza, como muestra la Figura 6.51, se debe compactar para rellenar la palabra de instrucción ancha. Compactar la traza involucra desplazar las asignaciones a las variables B y C al bloque antes de la decisión de salto. Consideremos primero el problema de desplazar la asignación a B. Si la asignación a B se desplaza antes del salto (y por tanto fuera de la traza), el código de X quedará afectado si utiliza B, ya que desplazar la asignación cambiaría el valor de B. Por ello, para desplazar la asignación a B, B no debe ser leído en X. Uno puede imaginar esquemas más inteligentes si B se leyese en X —por ejemplo, haciendo una copia de B y actualizar más tarde B. Estos esquemas no se utilizan generalmente porque son complejos de implementar y porque ralentizan el programa si la traza seleccionada no es óptima y las operaciones acaban requiriendo instrucciones adicionales. También, ya que la asignación a B es desplazada antes que el test de la sentencia «if», para que esta planificación sea válida o X también asigna a B, o B no se lee después de la sentencia if.

Desplazar la asignación a C hasta antes del primer salto requiere desplazarla primero a la rama desde X hacia adentro de la traza. Para hacer esto, se hace una copia de la asignación a C en la rama hacia adentro de la traza. Todavía debe hacerse una comprobación, como se hizo para B, para asegurarse que la asignación puede desplazarse a la rama hacia afuera de la traza. Si C se desplaza con éxito antes del primer salto y se toma la dirección «falso» del salto —la rama hacia afuera de la traza—, la asignación a C se habrá hecho dos veces. Esto puede ser más lento que el código original, dependiendo de si esta operación u otras operaciones desplazadas crean trabajo adicional en la traza principal. Irónicamente, cuanto mayor es la cantidad de código que el algoritmo de planificación de trazas mueve a través de los saltos, mayor es la penalización para una predicción errónea.

Desenrollamiento de bucles, planificación de trazas, y segmentación software, todos ayudan a intentar incrementar la cantidad de paralelismo local de las instrucciones que se puede explotar por una máquina emitiendo más de una instrucción cada ciclo de reloj. La efectividad de cada una de estas técnicas y su idoneidad para diversos enfoques arquitectónicos están entre las áreas más significativas de investigación abiertas en el diseño de procesadores segmentados.

6.9 Juntando todo: un VAX segmentado

En esta sección examinaremos la segmentación del VAX 8600, un VAX macrosegmentado. Esta máquina se describe con detalle en DeRosa y cols. [1985] y Troiani y cols. [1985]. La segmentación del 8600 es una estructura más dinámica que la segmentación de instrucciones enteras de DLX. Esto es porque los pasos del procesamiento pueden ocupar múltiples ciclos de una etapa de la segmentación. Adicionalmente, la detección de riesgos es más complicada debido a la posibilidad de que las etapas progresen independientemente y debido a que las instrucciones pueden modificar los registros antes de que se completen. Técnicas similares a las utilizadas en la segmentación FP de DLX para manipular instrucciones de duración variable se utilizan en la segmentación del 8600.

El 8600 está macrosegmentado —la segmentación comprende la estructura de las instrucciones VAX y solapa su ejecución, comprobando los riesgos en los operandos de la instrucción. Por comparación, el VAX 8800 está microsegmentado —las microinstrucciones se solapan y la detección de riesgos se realiza en la unidad del microprograma. Un ejemplar de Digital Technical Journal [Digital, 1987] describe esta máquina, y Clark [1987] describe la segmentación y su rendimiento. Los diseños son interesantes de comparar.

La Figura 6.52 muestra el 8600 dividido en cuatro componentes estructurales principales. MBox es responsable de la traducción de direcciones y de los accesos a memoria (ver Cap. 8). IBox es el núcleo de la segmentación del 8600; es responsable de la búsqueda y decodificación de instrucciones, cálculo de direcciones de operandos y búsqueda de operandos. EBox y FBox son responsables de la ejecución de las operaciones enteras y de punto flotante, y su

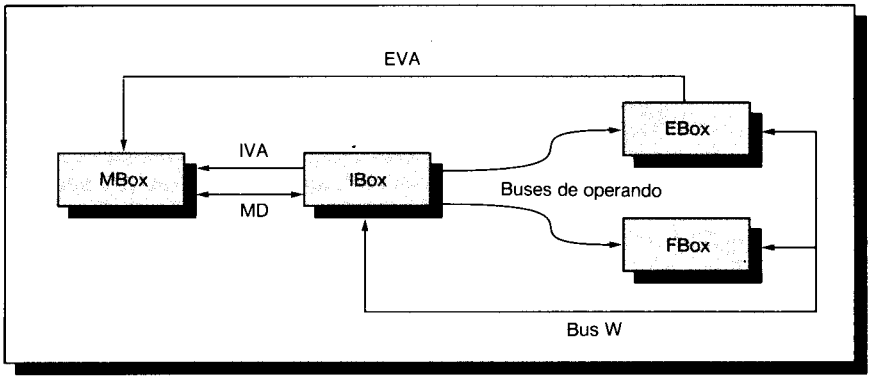


FIGURA 6.52 La estructura básica del 8600 consta de MBox (responsable de los accesos a memoria), IBox (manipula las instrucciones y tratamiento de operandos), EBox (todas las interpretaciones de códigos de operación excepto el punto flotante), y FBox (realiza las operaciones de punto flotante). Estas cuatro unidades están conectadas por seis buses importantes. IVA y EVA llevan la dirección de un acceso a memoria a MBox desde IBox y EBox. El bus MD lleva los datos de memoria a o desde MBox; todos los datos fluyen a través de IBox. EBox inicia directamente los accesos a memoria con MBox sólo bajo condiciones inusuales (por ejemplo, referencias mal alineadas). Los buses de operandos llevan operandos desde IBox (donde son buscados en memoria o en los registros) a EBox y FBox. Finalmente, el Bus W lleva los resultados que se van a escribir desde EBox y FBox a los GPR y a memoria, vía IBox.

función principal es implementar la parte de código de operación de una instrucción. (Debido a que FBox es opcional, EBox también contiene microcódigo para realizar el punto flotante, aunque con un rendimiento mucho menor. La presencia opcional de FBox complica además el procesamiento de operandos en EBox.) Como EBox y FBox no están segmentados, centraremos nuestra atención principalmente en IBox. Al explicar la función de IBox nos referiremos ocasionalmente a EBox; habitualmente, los mismos comentarios pueden aplicarse a FBox.

La Figura 6.53 divide la ejecución de una instrucción VAX en cuatro pa-

Paso	Función	Localizado en
1. Ifetch	Prebusca bytes de la instrucción y los decodifica	IBox
2. Opfetch	Cálculo de las direcciones y búsqueda de los operandos	IBox
4. Ejecución	Ejecuta código de op. y escribe resultado	EBox, FBox
4. Almacena resultado	Escribe resultado en memoria o registros	EBox, IBox

FIGURA 6.53 La estructura básica de la segmentación del 8600 tiene cuatro etapas, cada una de las cuales emplea desde 1 a un gran número de ciclos de reloj. Hasta cuatro instrucciones VAX se están procesando a la vez.

sos solapados. El número de ciclos de reloj por paso puede variar ampliamente, aunque cada paso de la segmentación emplee como mínimo un ciclo de reloj.

Una instrucción VAX puede necesitar muchos ciclos de reloj en un paso dado. Por ejemplo, con múltiples operandos de memoria, la instrucción necesitará múltiples ciclos de reloj en el paso de Búsqueda de Operando *(Opfetch)*. Debido a esto, una instrucción que necesite muchos ciclos de reloj en una etapa puede provocar la detención de etapas anteriores; esto puede alcanzar eventualmente el paso de prebúsqueda de instrucciones *(Ifetch)*, donde ocasionará que se detenga la búsqueda de instrucciones. Adicionalmente, varios recursos (por ejemplo, el W Bus y los puertos GPR) son utilizados por múltiples etapas de la segmentación. En general, estos problemas se resuelven utilizando un esquema de prioridad fija.

Búsqueda de decodificación de operandos

La mayoría del trabajo de interpretar una instrucción VAX está en el especificador de operandos y en el proceso de decodificación, y éste es el núcleo de IBox. Se dedica un esfuerzo sustancial para decodificar y buscar los operandos con tanta rapidez como sea posible para mantener a las instrucciones fluyendo a través de la segmentación. La Figura 6.54 muestra el número de ciclos empleados en la búsqueda de operandos bajo condiciones ideales (no hay

Especificador	Ciclos
Literal o inmediato	1
Registro	1
Diferido	1
Desplazamiento	1
Relativo PC y absoluto	1
Autodecremento	1
Autoincremento	2
Autoincremento diferido	5
Desplazamiento diferido	4
Relativo PC diferido	4

FIGURA 6.54 Número mínimo de ciclos empleados en Opfetch por cada especificador de operandos. Esto muestra los datos para un operando del tipo byte, palabra, o palabra larga que se lee. Los operandos modificados y escritos emplean un ciclo adicional, excepto para el modo registro e inmediato o literal, donde no se permiten escrituras. Los operandos de cuatro y ocho palabras pueden emplear mucho más tiempo. Si se producen algunas detenciones, se incrementará el número de ciclos.

fallos de cache u otras detenciones de la jerarquía de memoria) para cada especificador de operando. Si el resultado es un registro, EBox almacena el resultado. Si el resultado es un operando de memoria, Opfetch calcula la dirección y espera que EBox esté lista; entonces IBox almacena el resultado durante el paso de Almacenar Resultado. Si un resultado de instrucción se va a almacenar en memoria, EBox indica a IBox cuándo empieza el último ciclo de ejecución para la instrucción. Esto permite a Opfetch solapar el primer ciclo de una escritura en memoria de dos ciclos con el último ciclo de ejecución (aun cuando la operación sólo emplee un ciclo).

Para maximizar el rendimiento de la máquina, hay tres copias de los GPR —en IBox, EBox y FBox. Una escritura se difunde desde FBox, EBox o IBox (en el caso de direccionamiento de autoincremento o autodecremento) a las otras dos unidades, para que se puedan actualizar su copias de los registros.

Manipulación de dependencias de datos

Los riesgos de los registros son detectados en Opfetch manteniendo una pequeña tabla con los registros pendientes de escribir. Siempre que una instrucción pase a través de Opfetch, su registro de resultado se marca como ocupado. Si una instrucción que utiliza ese registro llega en Opfetch y ve el señalizador de ocupado a 1, se detiene hasta que el señalizador se ponga a 0. Esto evita los riesgos RAW. El señalizador de ocupado se borra cuando se escribe el registro. Como sólo hay dos etapas después de Opfetch (ejecutar y escribir resultado en memoria), el señalizador de ocupado se puede implementar como una memoria asociativa de dos entradas. Las escrituras se mantienen en orden y siempre al final de la segmentación, y todas las lecturas se hacen en Opfetch. Esto elimina todos los riesgos explícitos WAW y WAR. Los únicos riesgos posibles son los que puedan presentarse sobre operandos implícitos, como los registros escritos por una instrucción `MOVC3`. Los riesgos sobre operandos implícitos se evitan mediante el control explícito en el microcódigo.

Opfetch optimiza el caso de que el especificador del último operando sea un registro, procesando el especificador de operando registro al mismo tiempo que el penúltimo especificador. Además, cuando el registro de resultado de una instrucción es el operando fuente de la instrucción siguiente, en lugar de detener la instrucción dependiente, Opfetch simplemente, señala esta relación a EBox, permitiendo que proceda la ejecución sin ninguna detención. Esto es como el desvío en la segmentación de DLX.

Los riesgos de memoria entre lecturas y escrituras se resuelven fácilmente porque hay un único puerto de memoria, e IBox decodifica todas las direcciones de los operandos.

Manipulación de las dependencias de control

Hay dos aspectos para manipular saltos en un VAX: sincronizar el código de condición y tratar los riesgos de saltos. La mayor parte del procesamiento de

saltos lo realiza IBox. Se utiliza una estrategia de predicción efectiva; los siguientes pasos son necesarios cuando IBox ve un salto:

1. Calcula la dirección destino del salto, la envía a MBox e inicia una búsqueda de la dirección del destino. Espera a que EBox facilite CCSYNC, que indica que los códigos de condición están disponibles en el siguiente ciclo de reloj.
2. Evalúa los códigos de condición de EBox para comprobar la predicción. Si la predicción fue incorrecta, se aborta el acceso iniciado en MBox. El PC actual señala a la siguiente instrucción o su primer especificador de operando.
3. Suponiendo que sea efectivo, el salto IBox limpia las etapas de búsqueda y decodificación y comienza cargando el registro de instrucción y procesando el nuevo flujo del destino. Si no es efectivo el salto, el acceso al destino potencial ya ha sido eliminado y el procesador puede continuar utilizando lo que está en las etapas de búsqueda y decodificación.

Los saltos condicionales simples (BEQL, BNEQ), los saltos incondicionales (BRB, BRW) y los saltos calculados (por ejemplo, AOBLEQ) son tratados por IBox. EBox maneja saltos más complejos y también las instrucciones utilizadas para llamadas y retornos.

Un ejemplo

Para comprender realmente cómo funciona esta segmentación, examinemos cómo se ejecuta una secuencia de código. Este ejemplo está algo simplificado, pero es suficiente para demostrar las interacciones principales de la segmentación. La secuencia de código que consideraremos es como sigue (recordar que por consistencia se escribe primero el operando resultado de ADDL3):

```
        ADDL3    56(R3),R1,R2
        CMPL     45(R1),@54(R2)
        BEQL     target
        MOVL     ...
target: SUBL3    ...
```

La Figura 6.55 muestra un diagrama de cómo progresarían estas instrucciones a través de la segmentación de la 8600.

Tratamiento de interrupciones

El 8600 mantiene tres contadores de programa para que la interrupción y rearranque de instrucciones sean posibles. Estos contadores de programa y lo que contienen son:

- Contador del Programa Actual: señala el siguiente byte que va a ser procesado y consumido en Opfetch.

Instr.	Ciclo de Reloj 1	2	3	4	5	6	7	8	9
ADDL3	**IF:** Busca ADDL	**IF:** Continúa prebúsqueda si espacio y MBox disponibles	**IF:** Decodifica R1.	**IF:** Decodifica R2. **OP:** Busca R1	**IF:** Decodifica 56(R3). **OP:** Busca R2	**OP:** Computa 56+(R3) **EX:** obtiene primer operando	**OP:** Comienza escritura **EX:** Add	**WR:** Almacena	
CMPL						**IF:** Decodifica 45(R1)		**IF:** Decodifica @54(R2) **OP:** Busca 45(R1)	**OP:** Busca 54(R2)
BEQL									**IF:** Decodifica BEQL desplaza
SUBL									

Instr.	Ciclo de Reloj 10	11	12	13	14	15	16	17	18
ADDL3									
CMPL	**OP:** detención. **EX:** obtiene primer operando	**OP:** obtiene dirección indirecta	**OP:** Busca @54(R2)		**EX:** compara e inicializa CC				
BEQL				**OP:** Carga VA	**OP:** Busca destino del salto	**OP:** Busca destino +4 carga VIBA; limpia IBuffer			
SUBL							**IF:** Decodifica SUBL3	**OP:** Busca primer operando	**OP:** Busca segundo operando

- Dirección de Comienzo de IBox: apunta a la instrucción actual en Opfetch.
- Dirección de Comienzo de EBox: apunta a la instrucción ejecutándose en EBox o FBox.

Además, la unidad de prebúsqueda mantiene una dirección para prebuscar (VIBA, Dirección del Buffer de Instrucciones Virtuales —Virtual Instruction Buffer Address—), pero esto no afecta al manejo de interrupciones. Cuando una operación de prebúsqueda provoca una excepción se marca el byte del buffer de instrucción. Cuando Opfetch, eventualmente, pregunta por el byte, verá la excepción, y en el contador de programa actual estará la dirección del byte que provocó la excepción.

Estos PC se actualizan cuando una instrucción entra en la etapa correspondiente de la segmentación. Por consiguiente, si se presenta una interrupción en una etapa dada, el PC se puede volver a inicializar al comienzo de esa instrucción. Se necesitan estos PC porque la longitud de las instrucciones VAX es variable y sólo se pueden determinar calculando el byte de código de operación.

Además, para restaurar la dirección de comienzo de la instrucción que provocó la interrupción, debemos deshacer cualquier actualización de registros hecha por los modos de direccionamiento procesados en Opfetch para instrucciones que están después de la instrucción que interrumpe al procesador. IBox mantiene una anotación de las actualizaciones del fichero de registros realizadas por múltiples instrucciones, como hicimos en la Sección 5.6. Se deshacen los efectos de cualquier cambio y se restaura el PC. Esto permite que el sistema operativo tenga un estado de la máquina limpio para empezar a trabajar.

Comentarios finales

El 8600 utiliza una segmentación de cuatro pasos. El rendimiento teórico máximo con el reloj de 80-ns es 12,5 millones de instrucciones VAX por segundo. Algunas secuencias simples de instrucciones realmente pueden obtener este rendimiento máximo con un CPI igual a 1. Normalmente, el

FIGURA 6.55 (ver pág. adjunta). El VAX 8600 ejecutando una secuencia de código. La parte superior muestra los eventos en los instantes de reloj 1-9, mientras la parte inferior muestra los eventos en los instantes 10-18. Las etapas de la segmentación se abrevian por IF (Búsqueda de Instrucción), OP (Búsqueda de Operandos), EX (Ejecución) y WR (Escribir resultado) y se muestran en negrita. Cada instrucción pasa por la segmentación del 8600 tan pronto como la etapa de la segmentación está vacía y el dato requerido está disponible. Observar que una instrucción puede estar en ambas etapas IF y OP al mismo tiempo. Esta figura supone que al comienzo del ciclo 1, está vacío el buffer de prebúsqueda. La prebúsqueda en la etapa IF continúa buscando instrucciones mientras haya sitio en el buffer de prebúsqueda y un ciclo MBox disponible. Se omite del diagrama por simplicidad. La acción «detención» indica una detención para un operando de memoria durante una búsqueda de operando. En total, las tres instrucciones VAX ejecutadas emplean 15 ciclos, suponiendo no detenciones del sistema de memoria. Esta secuencia fue escogida para demostrar el funcionamiento de la segmentación —no es necesariamente típica.

rendimiento sobre código entero es aproximadamente 1,75 millones de instrucciones VAX por segundo para un CPI de aproximadamente 7. Esto da aproximadamente 3,5 veces el rendimiento de un VAX-11/780.

6.10 Falacias y pifias

Falacia: El diseño del repertorio de instrucciones tiene poco impacto sobre la segmentación.

Esto es quizá la concepción errónea más destacada sobre la segmentación y una de las más extendidas hasta hace poco. La mayoría de las dificultades de la segmentación surgen debido a complicaciones en el repertorio de instrucciones. Aquí hay algunos ejemplos, muchos de los cuales se mencionan en el capítulo:

- Tiempos de ejecución y longitudes de instrucción variables pueden conducir a desequilibrar las etapas de la segmentación, haciendo que otras etapas retrocedan. También complican severamente la detección de riesgos y el mantenimiento de interrupciones precisas. Por supuesto, hay excepciones a cada regla. Por ejemplo, los fallos de caches son causantes de tiempos de ejecución variables; sin embargo, las ventajas sobre el rendimiento de las caches hacen aceptable la complejidad añadida. Para minimizar la complejidad, muchas máquinas congelan la segmentación en un fallo de cache. Otras máquinas intentan continuar la ejecución de partes de la segmentación; aunque esto es muy complejo, puede remediar algunas pérdidas de rendimiento de los fallos de la cache.
- Modos de direccionamiento sofisticados pueden conducir a diferentes clases de problemas. Los modos de direccionamiento que actualizan registros, como postautoincremento, complican la detección de riesgos. También aumentan ligeramente la complejidad de recomenzar las instrucciones. Otros modos de direccionamiento que requieren múltiples accesos a memoria complican sustancialmente el control de la segmentación y hacen difícil de mantener un flujo constante de instrucciones.
- Las arquitecturas que permiten escritura en el espacio de instrucciones (código automodificable) pueden provocar problemas en la segmentación (así como en los diseños de la cache). Por ejemplo, si una instrucción puede modificar otra instrucción, debemos comprobar constantemente si la dirección que está escribiendo una instrucción corresponde a la dirección de una instrucción posterior actualmente en ejecución. Si es así, el procesador debe ser limpiado o la instrucción actualizada de alguna forma.
- Inicializar implícitamente los códigos de condición incrementa la dificultad para determinar cuándo se ha decidido un salto y para planificar los retardos de los saltos. El primer problema se presenta cuando la inicialización del código de condición no es uniforme, haciendo difícil decidir qué instrucción inicializa el último código de condición. El último problema se presenta cuando la inicialización del código de condición no está bajo con-

trol del programa. Esto hace difícil encontrar instrucciones que se puedan planificar entre la evaluación de la condición y el salto. Muchas arquitecturas más modernas evitan los códigos de condición o los inicializan explícitamente bajo control del programa para eliminar las dificultades de la segmentación.

A título de ejemplo, supongamos que el formato de instrucción de DLX fuese más complejo, de forma que se necesitase una etapa separada de decodificación en la segmentación antes de buscar los registros. Esto aumentaría el retardo de los saltos en dos ciclos de reloj. En el mejor de los casos, el segundo hueco de retardo de salto se desperdiciaría, por lo menos, con tanta frecuencia como el primero. Gross [1983] encontró que un segundo hueco de retardo se utilizaba con la mitad de la frecuencia que el primero. Esto conduce a una penalización del rendimiento para el segundo retardo de más de 0,1 ciclos de reloj por instrucción.

Pifia: Secuencias de ejecución inesperadas pueden provocar riesgos inesperados.

En un primer vistazo, los riesgos WAW parece que nunca ocurrirán porque el compilador no generará dos escrituras en el mismo registro sin una lectura intermedia. Pero se pueden presentar cuando no se espere la secuencia. Por ejemplo, la primera escritura podía estar en el hueco de retardo de un salto efectivo cuando el planificador consideró que el salto no iba a ser efectivo. Aquí está la secuencia de código que puede producir esto:

```
      BNEZ   R1,foo
      DIVD   F0,F2,F4  ; transferido a hueco de retardo
                       ; desde la rama secuencial del salto
      .....
      .....
foo:  LD     F0,qrs
```

Si el salto se realiza, entonces, antes que pueda completarse DIVD, LD alcanzará WB, provocando un riesgo WAW. El hardware debe detectar esto y puede detener la distribución de LD. Otra forma en que puede ocurrir esto es si la segunda escritura está en una rutina de trap. Esto se presenta cuando una instrucción que genera un trap y está escribiendo resultados continúa y se completa después de una instrucción que escribe el mismo registro en el manipulador de traps. El hardware también debe detectar e impedir esto.

Falacia: Incrementar la profundidad de la segmentación siempre incrementa el rendimiento.

Se combinan dos factores para limitar la mejora de rendimiento obtenida por la segmentación. Las dependencias de los datos en el código significan que, incrementar la profundidad de la segmentación, incrementará el CPI, ya que un porcentaje mayor de los ciclos serán detenciones. Segundo, el sesgo del re-

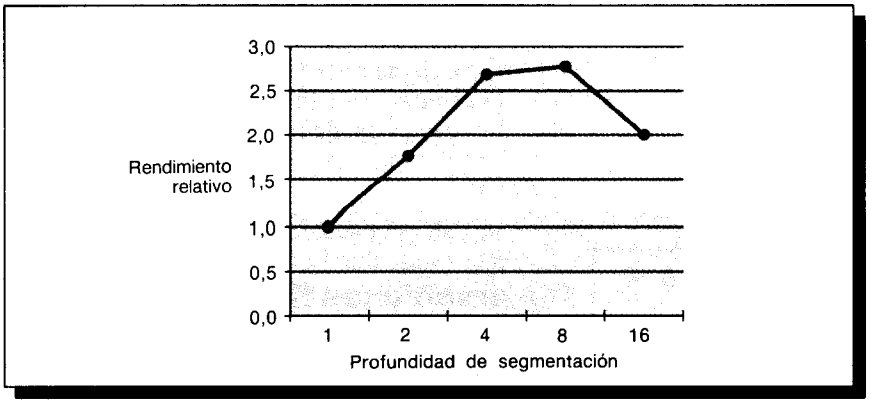


FIGURA 6.56 Profundidad de la segmentación frente a la aceleración obtenida. Este dato está basado en la Tabla 2 en Kunkel y Smith [1986]. El eje x muestra el número de etapas en la porción EX de la segmentación de punto flotante. Una segmentación de una sola etapa corresponde a 32 niveles de lógica, que podrían ser apropiados para una sola operación FP.

loj y los gastos de cerrojos se combinan para limitar la disminución del período de reloj obtenida mediante segmentación adicional. La Figura 6.56 muestra el compromiso entre profundidad de segmentación y rendimiento para los 14 primeros Livermore Loops —Bucles de Livermore— (ver Cap. 2). El rendimiento se hace plano cuando la profundidad de la segmentación alcanza 4 y cae realmente cuando la parte de ejecución está segmentada con profundidad 16.

Pifia: Evaluar un planificador en base a código no optimizado.

El código no optimizado —que contiene cargas, almacenamientos y otras operaciones redundantes que pueden ser eliminadas por un optimizador— es mucho más fácil planificar que el código altamente optimizado. Durante la ejecución de GCC en una DECstation 3100, la frecuencia de ciclos de reloj inactivos incrementa el 18 por 100 al comparar código no optimizado y planificado con el código optimizado y planificado. TeX muestra un incremento del 20 por 100 para la misma medida. Para evaluar con imparcialidad un planificador, se debe utilizar código optimizado, ya que en el sistema real se obtendrá un buen rendimiento de otras optimizaciones además de la planificación.

Pifia: La segmentación extensiva puede impactar otros aspectos de un diseño, llevando a una disminución global de coste/rendimiento.

El mejor ejemplo de este fenómeno proviene de dos implementaciones del VAX, el 8600 y el 8700. Explicamos la segmentación de las instrucciones del 8600 en la Sección 6.9. Recién aparecido el 8600, tenía un ciclo de 80 ns. Posteriormente, se introdujo una versión rediseñada, denominada 8650, con un

reloj de 55 ns. El 8700 tiene una segmentación mucho más simple que opera a nivel de microinstrucción. La CPU del 8700 es mucho más pequeña y tiene una frecuencia de reloj más rápida, 45 ns. El resultado global es que el 8650 tiene una ventaja de CPI de, aproximadamente, el 20 por 100, pero el 8700 tiene una frecuencia de reloj que es aproximadamente el 20 por 100 más rápida. Por ello, el 8700 consigue el mismo rendimiento con mucho menos hardware.

6.11 Observaciones finales

La Figura 6.57 muestra cómo los distintos enfoques de la segmentación afectan a la velocidad del reloj y al CPI. Esta figura no contabiliza diferencias en el recuento de instrucciones. Como el rendimiento es velocidad de reloj dividido por CPI (ignorando las diferencias en el recuento de instrucciones), las máquinas de la esquina superior izquierda serán las más lentas, y las máquinas de la esquina inferior derecha serán las más rápidas. Sin embargo, las máquinas que se desplazan hacia la esquina inferior derecha, probablemente, conseguirán su máximo rendimiento en el rango más estrecho de aplicaciones.

Las máquinas *subsegmentadas* (*underpipelined*) aglutinan múltiples etapas de la segmentación de DLX en una. El reloj no puede correr tan rápido, y el CPI será sólo marginalmente inferior. La segmentación de DLX consigue un CPI muy próximo a 1 (ignorando detenciones del sistema de memoria) a una velocidad razonable de reloj. La simplicidad arquitectónica y la segmentación eficiente son dos de los atributos más importantes de las máquinas RISC

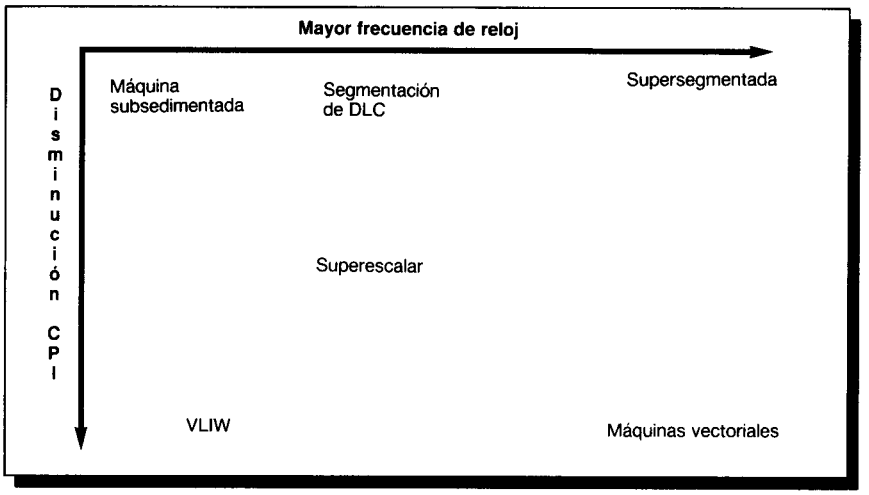


FIGURA 6.57 Incrementando la velocidad de emisión de instrucciones disminuye el CPI, mientras que una segmentación más profunda incrementa la frecuencia de reloj. Varias máquinas combinan estas técnicas.

(Computadores de Repertorio de Instrucciones Reducido —Reduced Instruction Set Computer—). DLX constituye un ejemplo de tal máquina. Hemos escogido utilizar el término arquitectura de carga/almacenamiento porque las ideas se aplican a un amplio rango de máquinas, y no sólo a las máquinas que se identifican por sí mismas como RISC. Muchas de las explicaciones de la primera parte de este capítulo se centraron en torno a las ideas clave desarrolladas por los proyectos RISC.

Las máquinas con frecuencias de reloj más altas y segmentaciones más profundas se han denominado *supersegmentadas*. Las máquinas supersegmentadas se caracterizan por la segmentación de todas las unidades funcionales. Una versión supersegmentada de DLX puede tener 10 etapas, en lugar de la segmentación de 5 etapas descrita antes. Además de incrementar la complejidad de la planificación y del control de la segmentación, las máquinas supersegmentadas no son fundamentalmente diferentes de las máquinas que ya hemos examinado en este capítulo. Debido al limitado paralelismo a nivel de instrucción, una máquina supersegmentada tendrá un CPI ligeramente mayor que una segmentación del estilo de DLX, pero su ventaja en la duración del ciclo de reloj será mayor que la desventaja en el CPI.

Los *procesadores superescalares* pueden tener duraciones del ciclo de reloj muy próximas a las de la segmentación de DLX y mantener un CPI más pequeño. Las máquinas VLIW pueden tener un CPI sustancialmente menor, pero tienden a tener una duración del ciclo de reloj significativamente mayor por las razones explicadas en este capítulo. Las máquinas vectoriales usan efectivamente ambas técnicas. Habitualmente son supersegmentadas y tienen potentes operaciones vectoriales que se pueden considerar equivalentes a emitir múltiples operaciones independientes en una máquina como DLX. Exploraremos con detalle las máquinas vectoriales en el siguiente capítulo.

Al alejarse de la esquina superior izquierda en cualquier eje, en la Figura 6.57, se incrementa el requerimiento para explotar más paralelismo a nivel de instrucción; por supuesto, al mismo tiempo, habrá menos programas que se ejecuten a la velocidad máxima.

6.12 Perspectiva histórica y referencias

Esta sección describe algunos de los avances principales en la segmentación y finaliza con parte de la literatura reciente sobre segmentación de alto rendimiento.

El Stretch, IBM 7030, es considerada como la primera máquina segmentada de propósito general. El Stretch seguía al IBM 704 y tenía el objetivo de ser 100 veces más rápida que el 704. Los objetivos fueron una extensión (stretch) del estado del arte en esa época —de aquí el apodo. El plan era obtener un factor de 1,6 a partir del solapamiento de búsquedas, decodificaciones y ejecuciones, utilizando una segmentación de 4 etapas. Bloch [1959] y Bucholtz [1962] describen los compromisos de diseño e ingeniería, incluyendo el uso de desvíos de la ALU.

En 1964, CDC sacó a la luz el primer CDC 6600. El CDC 6600 fue único en muchos aspectos. Además de introducir marcadores, el CDC 6600 fue la

primera máquina en hacer un uso amplio de múltiples unidades funcionales. También tenía procesadores periféricos que utilizaban una segmentación de tiempo compartido. Se comprendió la interacción entre segmentación y diseño del repertorio de instrucciones, y éste se mantuvo para fomentar la segmentación. El CDC 6600 también utilizaba una tecnología avanzada de encapsulamiento. Thornton [1964] describe el procesador segmentado y la arquitectura del procesador de E/S, incluyendo el concepto de ejecución de instrucciones fuera de orden. El libro de Thornton [1970] proporciona una descripción excelente de la máquina completa desde la tecnología a la arquitectura, e incluye una introducción de Cray. (Desgraciadamente, este libro no se imprime actualmente.) El CDC 6600 también tiene un planificador de instrucciones para los compiladores FORTRAN, descrito por Thorlin [1967].

El IBM 360/91 introdujo muchos conceptos nuevos, incluyendo identificación de datos, renombramiento de registros, detección dinámica de riesgos de memoria y adelantamiento *(forwarding)* generalizado. El algoritmo de Tomasulo se describe en su artículo de 1967. Anderson, Sparacio y Tomasulo [1967] describen otros aspectos de la máquina, incluyendo el uso de la predicción de saltos. Patt y sus colegas han descrito un enfoque, llamado HPSm, que es una extensión del algoritmo de Tomasulo [Hwu y Patt, 1986].

Una serie de descripciones generales sobre segmentación que aparecieron a finales de los años setenta y principios de los ochenta proporcionaron la mayor parte de la terminología y describieron la mayoría de las técnicas básicas utilizadas en las segmentaciones simples. Estos estudios incluyen: Keller [1975], Ramamoorthy y Li [1977], Chen [1980] y el libro de Kogge [1981], dedicado completamente a la segmentación. Davidson y sus colegas [1971, 1975] desarrollaron el concepto de tablas de reserva de la segmentación como una metodología de diseño para segmentaciones multiciclo con realimentación (también descrito en Kogge [1981]). Muchos diseñadores utilizan una variante de estos conceptos, como hicimos en las Figuras 6.3 y 6.4.

Las máquinas RISC refinaron la noción de segmentación planificada por compilador a principios de los años ochenta. Los conceptos de saltos retardados y cargas retardadas —comunes en microprogramación— se extendieron a la arquitectura de alto nivel. La arquitectura MIPS de Stanford hacía expresamente visible la estructura de la segmentación al compilador y permitía múltiples operaciones por instrucción. Esquemas para planificar la segmentación en el compilador se describen en Sites [1979] para el Cray, en Hennessy y Gross [1983] (y en la tesis de Gross [1983]) y en Gibbons y Muchnik [1986]. Rymarczyk [1982] describe las condiciones de interbloqueo que los programadores deberían conocer para una máquina como el 360; este artículo también muestra la interacción compleja entre segmentación y un repertorio de instrucciones no diseñado para ser segmentado.

J. E. Smith y sus colegas escribieron una serie de artículos examinando la emisión de instrucciones, manipulación de interrupciones y profundidad de la segmentación para máquinas escalares de alta velocidad. Kunkel y Smith [1986] evalúan el impacto de los gastos de la segmentación y dependencias en la elección de la profundidad óptima de la segmentación; también tienen una excelente discusión del diseño de cerrojos y su impacto en la segmentación. Smith y Plezkun [1988] evalúan una serie de técnicas para preservar interrupciones precisas, incluyendo el concepto de fichero de futuro mencionado en

la Sección 6.6. Weiss y Smith [1984] evalúan una serie de planificaciones hardware de procesadores segmentados y técnicas de emisión de instrucciones.

Los esquemas hardware de predicción dinámica de saltos son descritos por J. E. Smith [1981] y por A. Smith y Lee [1984]. Ditzel [1987] describe un nuevo buffer de destino de salto para CRISP. McFarling y Hennessy [1986] es una comparación cuantitativa de una serie de esquemas de predicción de saltos en tiempo de ejecución y en tiempo de compilación.

Una serie de artículos pioneros, incluyendo Tjaden y Flynn [1970] y Foster y Riseman [1972], concluían que sólo pequeñas cantidades de paralelismo podían estar disponibles a nivel de instrucción sin emplear gran cantidad de hardware. Estos artículos desalentaron el interés de emitir múltiples instrucciones durante más de diez años. Nicolau y Fisher [1984] publicaron un artículo afirmando la presencia de grandes cantidades de paralelismo potencial a nivel de instrucción.

Charlesworth [1981] informa sobre el AP-120B de Floating Point Systems, una de las primeras máquinas de grandes instrucciones, que contenía múltiples operaciones por instrucción. Floating Point Systems aplicó el concepto de segmentación software —aunque a mano, en lugar de con un compilador— al escribir bibliotecas en lenguaje ensamblador para utilizar la máquina eficientemente. Weiss y J. E. Smith [1987] comparan la segmentación software frente al desenrollamiento de bucles como técnicas para planificar código en una máquina segmentada. Lam [1988] presenta algoritmos para segmentación software y evalúa su uso sobre Warp, una máquina de palabra grande de instrucción. Junto con sus colegas de Yale, Fisher [1983] propuso crear una máquina con una instrucción muy grande (512 bits), y denominó este tipo de máquina VLIW. El código era generado para la máquina utilizando planificación de trazas, que Fisher [1981] había desarrollado originalmente para generar microcódigo horizontal. La implementación de la planificación de trazas para la máquina de Yale es descrita por Fisher y cols. [1984] y por Ellis [1986]. La máquina Multiflujo (Multiflow) (ver Colwell y cols. [1987]) comercializaba los conceptos desarrollados en Yale.

Algunos investigadores propusieron técnicas para emitir múltiples instrucciones. Agerwala y Cocke [1987] propusieron esta aproximación como una extensión de las ideas RISC, y acuñaron el nombre «superescalar». IBM describió una máquina basada en estas ideas a final de 1989 (ver Bakoglu y cols. [1989]). En 1990, el IBM fue anunciado como el RS/6000. La implementación puede emitir hasta cuatro instrucciones por reloj. Una buena descripción de la máquina, su estructura y software aparece en IBM [1990]. La Apollo DN 10000 y el Intel i860 ofrecen la emisión de múltiples instrucciones, aunque los requerimientos necesarios son más rígidos. El Intel i860 debería probablemente considerarse una máquina LIW, porque el programa debe indicar explícitamente si los pares de instrucciones se deben emitir en parejas. Aunque las parejas sean instrucciones ordinarias, hay limitaciones sustanciales que pueden aparecer como miembro de un par emitido en pareja. El Intel 960A y Tandem Cyclone son ejemplos de máquinas superescalares con repertorios complejos de instrucciones.

J. E. Smith y sus colegas en Wisconsin [1984] propusieron el enfoque desacoplado que incluía emisión múltiple con planificación dinámica de la seg-

mentación. El Astronautics ZS-1 descrita por Smith y cols. [1987] incorpora este enfoque y utiliza colas para conectar la unidad de carga/almacenamiento y las unidades de operación. J. E. Smith [1989] también describe las ventajas de la planificación dinámica y compara ese enfoque a la planificación estática. Dehnert, Hsu y Bratt [1989] explican la arquitectura y rendimiento de Cydra 5 de Cydrome, una máquina con una gran palabra de instrucción que proporciona renombramiento dinámico de registros. Cydra 5 es una mezcla única de hardware y software dedicada a la extracción de paralelismo a nivel de instrucción.

Recientemente, ha habido una serie de artículos explorando los compromisos entre enfoques alternativos de segmentación. Jouppi y Wall [1989] examinan las diferencias de rendimiento entre sistemas supersegmentados y superescalares, concluyendo que su rendimiento es similar, pero que las máquinas supersegmentadas pueden necesitar menos hardware para conseguir el mismo rendimiento. Sohi y Vajapeyam [1989] dan medidas de paralelismo disponibles para máquinas de grandes palabras de instrucciones. Smith, Johnson y Horowitz [1989] presentan estudios de paralelismo disponibles a nivel de instrucción en código no científico, utilizando un esquema hardware ambicioso que permite la ejecución de múltiples instrucciones.

Referencias

AGERWALA, T. AND J. COCKE [1987]. «Higt performance reduced instruction set processors», IBM Tech. Rep. (March).

ANDERSON, D. W., F. J. SPARACIO, AND R. M. TOMASULO [1967]. «The IBM 360 Model 91: Machine philosophy and instruction handling», *IBM J. of Research and Development* 11:1 (January) 8-24.

BAKOGLU, H. B., G. F. GROHOSKI, L. E. THATCHER, J. A. KAHLE, C. R. MOORE, D. P. TUTTLE, W. E. MAULE, W. R. HARDELL, D. A. HICKS, M. NGUYEN PHU, R. K. MONTOYE, W. T. GLOVER, AND S. DHAWAN [1989]. «IBM second-generation RISC machine organization», Proc. Int'l Conf. on Computer Design, IEEE (October) Rye, N. Y., 138-142.

BLOCH, E. [1959]. «The engineering design of the Stretch computer», *Proc. Fall Joint Computer Conf.*, 48-59.

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York.

CHARLESWORTH, A. E. [1981]. «An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family», *Computer* 14:12 (December) 12-30.

CHEN, T. C. [1980]. «Overlap and parallel processing» in *Introduction to Computer Architecture*, H. Stone, ed., Science Research Associates, Chicago, 427-486.

CLARK, D. W. [1987]. «Pipelining and performance in the VAX 8800 processor», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 173-177.

COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND B. K. RODMAN [1987]. «A VLIW architecture for a trace scheduling compiler», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 180-192.

DAVIDSON, E. S. [1971]. «The design and control of pipelined function generators», *Proc. Conf. on Systems, Networks, and Computers*, IEEE (January), Oaxtepec, Mexico, 19-21.

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. «Effective control for pipelined processors», *COMPCON, IEEE* (March), San Francisco, 181-184.

DEHNERT, J. C., P. Y.-T. HSU, AND J. P. BRATT [1989]. «Overlapped loop support on the Cydra 5», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (April), IEEE/ACM, Boston, 26-39.

DeRosa, J. R. Glackemeyer, and T. Knight [1985]. «Design and implementation of the VAX 8600 pipeline», *Computer* 18:5 (May) 38-48.

Digital Equipement Corporation [1987]. *Digital Technical J.* 4 (March), Hudson, Mass. (This entire issue is devoted to the VAX 8800 processor.)

Ditzel, D. R., and H. R. McLellan [1987]. «Branch folding in the CRISP microprocessor: Reducing the branch delay to zero», *Proc. 14th Symposium on Computer Architecture* (June), Pittsburgh, 2-7.

Earle. J. G. [1965]. «Latched carry-save adder», *IBM Technical Disclosure Bull.* 7 (March) 909-910.

Ellis, J. R., [1986]. *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

Emer, J. S., and D. W. Clark [1984]. «A characterization of processor performance in the VAX-11/780», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301-310.

Fisher, J. A. [1981]. «Trace Scheduling: A Technique for Global Microcode Compaction», *IEEE Trans. on Computers* 30:7 (July), 478-490.

Fisher, J. A. [1983]. «Very long instruction word architectures and ELI-512», *Proc. Tenth Symposium on Computer Architecture* (June), Stockholm, Sweden., 140-150.

Fisher, J. A., J. R. Ellis, J. C. Ruttenberg, and A. Nicolau [1984]. «Parallel processing: A smart compiler and a dumb machine», *Proc. SIGPLAN Conf. on Compiler Construction* (June), Palo Alto, CA, 11-16.

Foster, C. C., and E. M. Riseman [1972]. «Percolation of code to enhance parallel dispatching and execution», *IEEE Trans. on Computers* C-21:12 (December) 1411-1415.

Gibbons, P. B. and S. S. Muchnik [1986]. «Efficient Instruction Scheduling for a Pipelined Processor», *SIGPLAN'86 Symposium on Compiler Construction, ACM* (June), Palo Alto, 11-16.

Gross, T. R. [1983]. *Code Optimization of Pipeline Constraints*, Ph.D. Thesis (December), Computer Systems Lab., Stanford Univ.

Hennessy, J. L., and T. R. Gross [1983]. «Postpass code optimization of pipeline constraints», *ACM Trans. on Programming Languages and Systems* 5:3 (July) 422-448.

Hwu, W.-M., and Y. Patt [1986]. «HPSm, a high performance restricted data low architecture having minimum functionality», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 297-307.

IBM [1990]. «The IBM RISC System/6000 processor», collection of papers, *IBM Jour of Research and Development* 34:1 (January), 119 pages.

Jouppi N. P., and D. W. Wall [1989]. «Available instruction-level parallelism for superscalar and superpipelined machines», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272-282.

Keller, R. M. [1975]. «Look-ahead processors», *ACM Computing Surveys* 7:4 (December), 177-195.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

Kunkel, S. R. and J. E. Smith [1986]. «Optimal pipelining in supercomputers», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404-414.

Lam, M. [1988]. «Software pipelining: An effective scheduling technique for VLIW machines», *SIGPLAN Conf. on Programming Language Design and Implementation*, ACM (June), Atlanta, Ga., 318-328.

McFarling, S., and J. Hennessy [1986]. «Reducing the cost of branches», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396-403.

Nicolau, A., and J. A. Fisher [1984]. «Measuring the parallelism available for very long instruction work architectures», *IEEE Trans. on Computers* C-33:11 (November) 968-976.

Ramamoorthy, C. V., and H. F. Li [1977]. «Pipeline architecture», *ACM Computing Surveys* 9;1 (March) 61-102.

Rymarczyk, J. [1982]. «Coding guidelines for pipelined processors», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 12-19.

Sites, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego.

Smith, A., and J. Lee [1984]. «Branch prediction strategies and branch target buffer design», *Computer* 17:1 (January), 6-22.

Smith, J. E. [1981]. «A study of branch prediction strategies», *Proc. Eighth Symposium on Computer Architecture* (May), Minneapolis, 135-148.

SMITH, J. E. [1984]. «Decoupled access/execute computer architectures», *ACM Trans. on Computer Systems* 2:4 (November), 289-308.

SMITH, J. E. [1989]. «Dynamic instruction scheduling and the Astronautics ZS-1», *Computer* 22:7 (July) 21-35.

SMITH, J. E., AND A. R. PLEZKUN [1988]. «Implementing precise interrupts in pipelined processors», *IEEE Trans. on Computers* 37:5 (May), 562-573.

SMITH, J. E., G. E. DERMER, B. D. VANDERWARN, S. D. KLINGER, C. M. ROZEWSKI, D. L. FOWLER, K. R. SCIDMORE, J. P. LAUDON [1987]. «The ZS-1 central processor», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 199-204.

SMITH, M. D., M. JOHNSON, AND M. A. HOROWITZ [1989]. «Limits on multiple instruction issue», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, Mass., 290-302.

SOHI, G. S., AND S. VAJAPEYAM [1989]. «Tradeoffs in instruction format design for horizontal architectures», *Porc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, Mass. 15-25.

THORLIN, J. F. [1967]. «Code generation for PIE (parallel instruction excecution) computers», *Spring Joint Computer Conf.* (April), Atlantic City, N. J.

THORNTON, J. E. [1964]. «Parallel operation in the Control Data 6600», *Proc. Fall Joint Computer Conf.*, 26, 33-40.

THORNTON, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, Ill.

TJADEN, G. S., AND M. J. FLYNN [1970]. «Detection and parallel execution of independent instructions», *IEEE Trans. on Computers* C-19:10 (October), 889-895.

TOMASULO, R. M. [1967]. «An efficient algorithm for exploiting multiple arithmetic units», *IBM J. of Research and Development*, 11:1 (January), 25-33.

TROIANI, M. S., S. CHING, N. N. QUAYNOR, J. E. BLOEM, AND F. C. COLON OSORIO [1985]. «The VAX 8600 I Box, a pipelined implementation of the VAX architecture», *Digital Technical J.*, 1 (August), 4-19.

WEISS, S., AND J. E. SMITH [1984]. «Instruction issue logic for pipelined supercomputers», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110-118.

WEISS, S., AND J. E. SMITH [1987]. «A study of scalar compilation techniques for pipelined supercomputers», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105-109.

EJERCICIOS

6.1 [12/12/15/20/15/15] <6.2-6.4> Considerar una arquitectura con dos formatos de instrucción; un formato registro-registro y otro registro-memoria. Hay un sólo modo de direccionamiento de memoria (desplazamiento + registro base).

Hay un conjunto de operaciones de la ALU con formato:

ALUop Rdest, $Rsrc_1$, $Rsrc_2$

o

ALUop Rdest, $Rsrc_1$, MEM

Donde ALUop es una de las operaciones siguientes: Suma, Resta, And, Or, Carga ($Rsrc_1$ ignorado), Almacenamiento. Rsrc o Rdest son registros. MEM es un registro base más un desplazamiento.

Los saltos utilizan una comparación completa de dos registros y son relativos al PC. Suponer que esta máquina está segmentada para que, en cada ciclo de reloj, se comience una nueva instrucción. Se utiliza la segmentación siguiente —similar a la utilizada en la microsegmentación del VAX 8800.

```
IF   RF   ALU1 MEM  ALU2 WB
     IF   RF   ALU1 MEM  ALU2 WB
          IF   RF   ALU1 MEM  ALU2 WB
               IF   RF   ALU1 MEM  ALU2 WB
                    IF   RF   ALU1 MEM  ALU2 WB
                         IF   RF   ALU1 MEM  ALU2 WB
```

La primera etapa de la ALU se utiliza para el cálculo de la dirección efectiva para referencias a memoria y saltos. El segundo ciclo de la ALU se utiliza para operaciones y comparaciones de saltos. RF es un ciclo de decodificación y de búsqueda de registros.

a) [12] Determinar el número de sumadores necesarios, contabilizando cualquier sumador o incrementador; mostrar una combinación de instrucciones y etapas de la segmentación que justifiquen esta respuesta. Sólo se necesita dar una combinación que maximice el número de sumadores. Suponer que las lecturas en RF y escrituras en WB se presentan como en la Figura 6.8.

b) [12] Calcular el número de puertos de escritura y lectura de registros y de puertos de escritura y lectura de memoria que se necesitan. Mostrar que la respuesta es correcta presentando una combinación de instrucciones y etapas de la segmentación indicando la instrucción y el número de puertos de lectura y de escritura necesarios para esa instrucción.

c) [15] Determinar cualquier *adelantamiento de datos* (*data forwarding*) para cualquier ALU que se necesite. Suponer que hay ALU separadas para las etapas ALU1 y ALU2. Añadir todos los adelantamientos de ALU a ALU necesarios para evitar o reducir detenciones. Mostrar la relación entre las dos instrucciones involucradas en el adelantamiento.

d) [20] Mostrar algún otro requerimiento de adelantamiento de datos para las unidades listadas más abajo dando un ejemplo de la instrucción fuente y la instrucción destino del adelantamiento. Cada ejemplo debe mostrar la máxima separación de las dos instrucciones. ¿Cuántas instrucciones puede adelantar cada ejemplo? Sólo es necesario considerar las siguientes unidades: MDR_{in} (registro de datos de entrada de memoria), MDR_{out} (registro de memoria de datos para dato que sale), ALU_1 y ALU_2. Incluir cualquier adelantamiento que se necesite para evitar o reducir detenciones.

e) [15]Dar un ejemplo de todos los riesgos restantes después de que se han implementado todos los adelantamientos. ¿Cuál es el máximo número de detenciones para cada riesgo?

f) [15] Mostrar todos los tipos de riesgos de control, por ejemplo, y explicar la duración de la detención.

6.2 [12] <6.1-6.4> Una máquina se denomina «subsegmentada» si se pueden añadir niveles adicionales de segmentación sin cambiar apreciablemente el comportamiento de las detenciones de la segmentación. Suponer que la segmentación de DLX se cambió a cuatro etapas uniendo ID y EX y alargando el ciclo de reloj un 50 por 100. ¿Cuántas veces es más rápida la segmentación convencional de DLX frente a la subsegmentación de DLX sólo para código entero? Asegurarse de incluir el efecto de cualquier cambio en las detenciones de la segmentación los datos de la Figura 6.24.

6.3 [15] <6.2-6.4> Sabemos que una implementación segmentada de profundidad cuatro tiene las siguientes frecuencias de riesgos y requerimientos de detenciones entre una instrucción *i* y sus sucesores:

$i + 1$ (y no en $i + 2$)	20%	2 detenciones por ciclo
$i + 2$	5%	1 detención por ciclo

Suponer que la frecuencia de reloj de la máquina segmentada es cuatro veces la frecuencia de reloj de la implementación no segmentada. ¿Cuál es el incremento efectivo de rendimiento debido a la segmentación si ignoramos el efecto de los riesgos? ¿Cuál es el incremento de rendimiento efectivo debido a la segmentación si tenemos en cuenta el efecto de los riesgos?

6.4 [15] <6.3> Suponer que las frecuencias de saltos (como porcentajes de todas las instrucciones) son como sigue:

Saltos condicionales	20 %
Bifurcaciones y llamadas	5 %
Saltos condicionales	60 % son efectivos

Estamos examinando una segmentación de profundidad cuatro donde el salto se resuelve al final del segundo ciclo para los saltos incondicionales, y al final del tercero para los saltos condicionales. Suponiendo que sólo la primera etapa de la segmentación se puede hacer siempre independientemente del resultado del salto e ignorando las demás detenciones de la segmentación, ¿cuántas veces sería más rápida la máquina sin ningún riesgo de saltos?

6.5 [20] <6.4> Varios diseñadores han propuesto el concepto de cancelar saltos (también denominado aplastar «squashing» o anular), como una forma de mejorar el rendimiento de los saltos retardados. (Algunas de las máquinas explicadas en el Apéndice E tienen esta capacidad.) La idea es permitir que el salto indique que se puede abortar la instrucción en el hueco de retardo si el salto está mal predicho. La ventaja de cancelar saltos es que el hueco de retardo **siempre** puede llenarse, ya que el salto puede abortar el contenido del hueco de retardo si está mal predicho. El compilador no necesita preocuparse sobre si la instrucción es correcta para ejecutar cuando el salto está mal predicho.

Una sencilla versión de cancelar saltos cancela si el salto no es efectivo; suponer este tipo de cancelación de saltos. Utilizar el dato de la Figura 6.18 para la frecuencia de saltos. Suponer que el 27 por 100 de los huecos de retardo de saltos se rellena utilizando la estrategia (a) de la Figura 6.20 con saltos estándares retardados, y que el resto de los huecos se rellenan utilizando cancelación de saltos y la estrategia (b). Utilizando los datos efectivos/no efectivos de la Figura 3.22, mostrar el rendimiento de este esquema con cancelación de saltos utilizando el mismo formato que el gráfico de la Figura 6.22. ¿Cuánto más rápida sería una máquina con cancelación de saltos, suponiendo que no hay penalizaciones de velocidad de reloj, comparada con una máquina que sólo tenga saltos retardados? Utilizar la Figura 6.24 para determinar el número de detenciones de retardos de saltos y cargas.

6.6 [20/15/20] <6.2-6.4> Suponer que tenemos la siguiente segmentación de cauce:

Etapa	Función
1	Busca instrucción
2	Decodifica operando
3	Ejecución o acceso a memoria (resolución de salto)

Todas las dependencias de datos están entre el registro escrito en la Etapa 3 de la instrucción i y un registro leído en la Etapa 2 de la instrucción $i + 1$, antes de que se haya completado la instrucción i. La probabilidad de que ocurra un interbloqueo es $1/p$.

Estamos considerando un cambio en la organización de la máquina que post-escriba el resultado de una instrucción durante una cuarta etapa de la segmentación. Esto haría disminuir la duración del ciclo de reloj en d (por ejemplo, si la duración del ciclo de reloj era T, ahora es T-d). La probabilidad de una dependencia entre la instrucción i y la $i + 2$ es p^{-2}. (Suponer que el valor de p^{-1} excluye instrucciones que interbloquearían a $i + 2$.) El salto también se resolvería durante la cuarta etapa.

a) [20] Considerando sólo los riesgos por dependencias de datos, determinar el límite inferior sobre d que haga de lo anteriormente expuesto un cambio provechoso. Suponer que cada resultado tiene exactamente un uso y que el ciclo básico de reloj tiene una duración T.

b) [15] Suponer que la probabilidad de un interbloqueo entre i e $i + n$ fuera $0,3 - 0,1n$ para $1 \leqslant n \leqslant 3$. ¿Qué incremento de la frecuencia de reloj se necesitaría para que este cambio mejorase el rendimiento?

c) [20] Suponer ahora que hemos utilizado adelantamientos para eliminar los riesgos extra introducidos por el cambio. Es decir, para todos los riesgos por dependencias de *datos* la duración de la segmentación es *efectivamente* 3. Este diseño todavía puede no ser valioso debido al impacto de los riesgos de control provenientes de una segmentación de cuatro etapas frente a una de tres etapas. Suponer que sólo puede ejecutarse con seguridad la Etapa 1 de la segmentación antes que decidamos si un salto es efectivo o no. Queremos saber cómo puede ser el impacto de los riesgos de saltos antes de que esta segmentación más larga no produzca alto rendimiento. Encontrar un límite superior para el porcentaje de saltos condicionales de programas en función de la relación de d a la duración del ciclo de reloj original, para que la segmentación más larga tenga mejor rendimiento. Si d es igual a un 10 por 100 de reducción, ¿cuál es el máximo porcentaje de saltos condicionales antes de que perdamos con esta segmentación más larga? Suponer que la frecuencia de saltos efectivos para los saltos condicionales es del 60 por 100.

6.7 [12] <6.7> Una desventaja del enfoque del marcador se presenta cuando múltiples unidades funcionales que comparten los buses de entrada están esperando un resultado. Estas unidades no pueden comenzar simultáneamente, sino en serie. Esto no es cierto en el algoritmo de Tomasulo. Dar una secuencia de código que utilice no más de 10 instrucciones y muestre este problema. Utilizar las latencias de FP de la Figura 6.29 y las mismas unidades funcionales en ambos ejemplos. Indicar dónde puede continuar el enfoque de Tomasulo, pero debe detenerse el enfoque del marcador.

6.8 [15] <6.7> El algoritmo de Tomasulo también tiene una desventaja frente al marcador: sólo se puede completar un resultado por ciclo de reloj, debido al CDB. Utilizando las latencias de FP de la Figura 6.29 y las mismas unidades funcionales en ambos casos, determinar una secuencia de código de no más de 10 instrucciones donde no se detenga el marcador, pero el algoritmo de Tomasulo deba detenerse. Indicar dónde ocurre esto en la secuencia propuesta.

6.9 [15] <6.7> Suponer que tenemos una máquina segmentada profundamente, para la cual implementamos un buffer de destinos de saltos sólo para saltos condicionales. Suponer que la penalización por predicciones erróneas es siempre de 4 ciclos y la penalización de fallo del buffer es siempre de 3 ciclos. Suponer una frecuencia de aciertos del 90 por 100, y una precisión del 90 por 100, y las estadísticas de saltos de la Figura 6.18. ¿Cuántas veces es más rápida la máquina con buffer de destinos de saltos

frente a una máquina que tenga una penalización fija de salto de 2 ciclos? Suponer un CPI base sin detención de saltos de 1.

6.10 [15] <6.7> Algunos diseñadores han propuesto utilizar buffers de destino de saltos para obtener un salto incondicional de retardo cero (ver Ditzel y McLellan [1987]). El buffer, simplemente, almacena la instrucción destino en lugar del PC destino. En un salto incondicional que acierte en el buffer de destinos de saltos, se busca la instrucción destino y se envía al procesador en lugar del salto condicional. Suponiendo una frecuencia de aciertos del 90 por 100, un CPI base de 1, y los datos de la Figura 6.18, ¿cuánto se gana por esta mejora frente a una máquina cuyo CPI efectivo sea 1,1?

6.11-6.19 Para estos problemas examinaremos cómo se ejecuta un bucle común de vectores en diversas versiones segmentadas de DLX. El bucle se denomina bucle SAXPY (explicado con detalle en el Capítulo 7). El bucle implementa la operación vectorial Y = a · X + Y para un vector de longitud 100. Aquí está el código DLX para el bucle:

```
foo:  LD     F2,0(R1)    ; carga X(i)
      MULTD  F4,F2,F0    ; multiplica a · X(i)
      LD     F6,0(R2)    ; carga Y(i)
      ADDD   F6,F4,F6    ; suma aX(i) + Y(i)
      SD     0(R2),F6    ; almacena Y(i)
      ADDI   R1,R1,8     ; incrementa índice X
      ADDI   R2,R2,8     ; incrementa índice Y
      SGTI   R3,R1,done  ; test si finalizado
      BEQZ   R3,foo      ; bucle si no finalizado
```

Para estos problemas, suponer que las operaciones enteras se emiten y completan en un ciclo de reloj y que sus resultados se desvían completamente. Ignorar el retardo de salto. Se utilizarán las latencias FP mostradas en la Figura 6.29, a menos que se indique otra cosa. Suponer que las unidades FP no están segmentadas, a menos que lo indique el problema.

6.11 [20] <6.2-6.6> Para este problema utilizar las restricciones de segmentación mostradas en la Figura 6.29. Mostrar el número de ciclos de detención para cada instrucción y en qué ciclo de reloj comienza la ejecución de la instrucción (por ejemplo, entra a su primer ciclo EX) en la primera iteración del bucle. ¿Cuántos ciclos de reloj necesita cada iteración del bucle?

6.12 [22] <6.7> Utilizando el código de DLX para el bucle SAXPY anterior, mostrar el estado de las tablas del marcador (como en la Figura 6.32) cuando la instrucción SGTI alcanza Escribir *(Write)* resultado. Suponer que cada acción de emisión y leer un operando necesita un ciclo de reloj. Suponer que hay tres unidades funcionales enteras y emplean un solo ciclo de ejecución (incluyendo cargas y almacenamientos). Suponer el número de unidades funcionales descrito en la Sección 6.7 con las latencias FP de la Figura 6.29. El salto no debería incluirse en el marcador.

6.13 [22] <6.7> Utilizar el código de DLX para el bucle SAXPY anterior y las latencias de la Figura 6.29. Suponiendo el algoritmo de Tomasulo para hardware con unidades funcionales como las descritas en la Sección 6.7, mostrar el estado de las estaciones de reserva y de las tablas de estado de registros (como en la Figura 6.37) cuando SGTI escriba su resultado en el CDB. Hacer las mismas hipótesis sobre latencias y unidades funcionales que en el Ejercicio 6.12.

6.14 [22] <6.7> Utilizando el código de DLX para el bucle SAXPY anterior, suponer un marcador con las unidades funcionales descritas en el algoritmo para el hardware, más tres unidades funcionales enteras (también utilizadas para carga/almacenamiento). Suponer las siguientes latencias de ciclos de reloj:

Multiplicación FP	10
Suma FP	6
Carga/almacenamiento FP	2
Todas operaciones enteras	1

Mostrar el estado del marcador (como en la Figura 6.32) cuando el salto se emita por segunda vez. Suponer que el salto se predijo correctamente y necesitó un ciclo. ¿Cuántos ciclos de reloj se necesitan en cada iteración del bucle? Se puede ignorar cualquier conflicto en los puestos/buses de registros.

6.15 [25] <6.7> Utilizar el código de DLX para el bucle SAXPY anterior. Suponer el algoritmo de Tomasulo para el hardware utilizando el número de unidades funcionales mostrado en la Sección 6.7. Suponer las siguientes latencias en ciclos de reloj:

Multiplicación FP	10
Suma FP	6
Carga/almacenamiento FP	2
Todas operaciones enteras	1

Mostrar el estado de las estaciones de reserva y tablas de estado de los registros (como en la Figura 6.37) cuando se ejecuta el salto por segunda vez. Suponer que el salto se predijo correctamente. ¿Cuántos ciclos de reloj emplea cada iteración del bucle?

6.16 [22] <6.8> Desenrollar el código de DLX para el bucle SAXPY tres veces, y planificarlo para la segmentación estándar de DLX. Suponer las latencias FP de la Figura 6.29. Al desenrollar, se debería optimizar el código como en la Sección 6.8. Para maximizar el rendimiento será necesario reordenar significativamente el código. ¿Cuál es la aceleración sobre el bucle original?

6.17 [25] <6.8> Suponer una arquitectura superescalar que pueda emitir dos operaciones independientes cualesquiera en un ciclo de reloj (incluyendo dos operaciones enteras). Desenrollar tres veces el código de DLX para el bucle SAXPY y planificarlo suponiendo las latencias FP de la Figura 6.29. Suponer una copia completamente segmentada de cada unidad funcional (por ejemplo, sumador FP, multiplicador FP). ¿Cuántos ciclos de reloj empleará cada iteración en el código original? Al desenrollar, se debería optimizar el código como en la Sección 6.8. ¿Cuál es la aceleración con respecto al código original?

6.18 [25] <6.8> En una máquina supersegmentada, en lugar de tener múltiples unidades funcionales, deberíamos segmentar completamente todas las unidades. Suponer que diseñamos una DLX supersegmentada, que tenga dos veces la frecuencia de reloj de nuestra segmentación estándar DLX y pueda emitir dos operaciones cualesquiera no relacionadas en el mismo tiempo que la segmentación DLX normal emitía una operación. Desenrollar el código SAXPY de DLX tres veces y planificarlo para esta máquina supersegmentada suponiendo las latencias FP de la Figura 6.29. ¿Cuántos ciclos de reloj emplea cada iteración del bucle? Recordar que estos ciclos de reloj son la mitad de largos que los de una segmentación estándar DLX o un DLX superescalar.

6.19 [20] <6.8> Comenzar con el código SAXPY y la máquina utilizada en la Figura 6.49. Desenrollar tres veces el bucle SAXPY realizando optimizaciones sencillas. Rellenar una tabla como la de la Figura 6.49 para el bucle desenrollado. ¿Cuántos ciclos de reloj emplea cada iteración del bucle?

6.20 [35] <6.1-6.4> Cambiar el simulador de instrucciones de DLX para que sea segmentado. Medir la frecuencia de los huecos de retardo de salto vacíos, la frecuencia de los retardos de carga, y la frecuencia de detenciones FP para una serie de programas enteros y de FP. Medir también la frecuencia de las operaciones de adelantamiento. Determinar cuál sería el impacto del rendimiento al eliminar los adelantamientos y detenciones.

6.21 [35] <6.6> Utilizando un simulador DLX, crear un simulador segmentado DLX. Explorar el impacto de alargar la segmentación FP, suponiendo unidades FP no segmentadas y completamente segmentadas. ¿Cómo afectan a los resultados los agrupamientos de operaciones FP? ¿Qué unidades FP son más susceptibles para cambios de la longitud de la segmentación FP?

6.22 [40] <6.4-6.6> Escribir un planificador de instrucciones para DLX que opere sobre el lenguaje ensamblador de DLX. Evaluar el planificador utilizando perfiles de programas o con un simulador del procesador segmentado. Si el compilador C de DLX realiza la optimización, evaluar el rendimiento del planificador con y sin optimización.

6.23 [35] <6.4-6.6> Escribir un simulador de la segmentación de DLX que utilice el algoritmo de Tomasulo con las unidades funcionales descritas. Evaluar el rendimiento de esta máquina comparada con la segmentación básica de DLX.

6.24 [Discusión] <6.7> La planificación dinámica de instrucciones requiere una considerable inversión en hardware. En contrapartida, esta capacidad permite al hardware ejecutar programas que no podrían ejecutarse a velocidad máxima con sólo planificación estática en tiempo de compilación. ¿Qué compromisos deberán tenerse en cuenta al tratar de decidir entre un esquema planificado estáticamente y otro dinámicamente? ¿Qué tipo de situaciones en tecnología hardware y en características de los programas están a favor de un enfoque o de otro?

6.25 [Discusión] <6.7> Hay un sutil problema que se debe considerar cuando se implemente el algoritmo de Tomasulo. ¿Qué ocurre si una instrucción pasa a una estación de reserva durante el mismo período de reloj que uno de sus operandos va al bus común de datos? Antes de que una instrucción esté en una estación de reserva, los operandos son extraidos del fichero de registros; pero una vez que está en la estación, los operandos se obtienen siempre del CDB. Ya que la instrucción y su etiqueta de operando están en tránsito a la estación de reserva, la etiqueta no puede compararse con la etiqueta en el CDB. Así hay una posibilidad de que la instrucción esté en la estación de reserva esperando de forma indefinida un operando. ¿Cómo podría resolverse este problema? Se puede considerar el subdividir uno de los pasos del algoritmo en múltiples partes. (Este problema es cortesía de J. E. Smith.)

6.26 [Discusión] <6.8> Discutir las ventajas y desventajas de una implementación superescalar, una implementación supersegmentada y un enfoque VLIW en el contexto de DLX. ¿Qué niveles de paralelismo a nivel de instrucción favorecen cada enfoque? ¿Qué otros aspectos se podrían considerar al escoger el tipo de máquina a construir?

Ciertamente, no estoy inventando máquinas vectoriales. Hay tres tipos que sé que existen hoy en día. Están representadas por el Illiac-IV, la máquina Star (CDC) y la máquina TI (ASC). Esas tres fueron máquinas pioneras... Uno de los problemas de ser pionero es que siempre se cometen errores y yo nunca, nunca quiero ser pionero. Siempre es mejor ir de segundo cuando se pueden observar los errores que cometieron los pioneros.

Seymour Cray,
Conferencia pública en «Lawrence Livermore Laboratories» sobre Introducción del CRAY-1 (1976)

7.1 ¿Por qué máquinas vectoriales?

7.2 Arquitectura vectorial básica

7.3 Dos aspectos del mundo real: longitud del vector y separación entre elementos

7.4 Un modelo sencillo para el rendimiento vectorial

7.5 Tecnología de compiladores para máquinas vectoriales

7.6 Mejorando el rendimiento vectorial

7.7 Juntando todo: evaluación del rendimiento de los procesadores vectoriales

7.8 Falacias y pifias

7.9 Observaciones finales

7.10 Perspectiva histórica y referencias

Ejercicios

7 Procesadores Vectoriales

7.1 ¿Por qué máquinas vectoriales?

En el último capítulo examinamos con detalle la segmentación y vimos que gestionar los segmentos, emitir varias instrucciones por ciclo de reloj e incrementar el número de segmentos o etapas en un procesador podría, como mucho, duplicar el rendimiento de una máquina. Sin embargo, existen límites en la mejora del rendimiento que la segmentación puede conseguir. Estos límites están impuestos por dos factores principales:

- Duración del ciclo de reloj. La duración del ciclo de reloj se puede hacer más pequeña aumentando el número de etapas o segmentos, pero hará aumentar el número de dependencias, y con ello el valor CPI. A partir de cierto momento, cualquier incremento en el número de etapas lleva consigo un incremento en el CPI. Como vimos en la Sección 6.10, un número elevado de etapas puede hacer más lento al procesador.
- Velocidad en la búsqueda y decodificación de las instrucciones. Esta limitación, conocida a veces como *cuello de botella*, fue detectada por Flynn (Flynn [1966]) e imposibilita la búsqueda y emisión de más de unas pocas instrucciones por ciclo de reloj. Vimos que, en la mayoría de los procesadores segmentados, el número promedio de instrucciones emitidas, por ciclo de reloj, fue inferior a una.

Las limitaciones duales impuestas por un mayor número de segmentos y por la emisión de varias instrucciones puede considerarse desde el punto de vista de la frecuencia de reloj o del CPI: es tan difícil gestionar un procesador que

tenga *n* veces más segmentos como gestionar un procesador que emita *n* instrucciones por ciclo.

Los procesadores segmentados, de alta velocidad, son particularmente útiles para grandes aplicaciones científicas y de ingeniería. Un procesador segmentado de alta velocidad, habitualmente, utilizará una cache, para no permitir que las instrucciones con referencia a memoria tengan una latencia muy alta. Sin embargo, los programas científicos grandes, cuya ejecución es larga, tienen con frecuencia conjuntos de datos activos muy grandes que son accedidos, a menudo, con baja localidad, consiguiendo un rendimiento pobre de la jerarquía de memoria. El impacto resultante es una disminución en el rendimiento de la cache. Este problema podría superarse sin utilizar caches en estas estructuras si fuese posible determinar los patrones de acceso a memoria y segmentar dichos accesos eficientemente. Los compiladores pueden ayudar a resolver este problema en el futuro (ver Sección 10.7).

Las *máquinas vectoriales* proporcionan operaciones de alto nivel que trabajan sobre *vectores* —arrays lineales de números. Una operación vectorial normal puede sumar dos vectores de 64 elementos en punto flotante para obtener como resultado un vector de 64 elementos. La instrucción vectorial es equivalente a un bucle completo, donde en cada iteración se calcula uno de los 64 elementos del resultado, actualizando los índices y saltando al comienzo.

Las operaciones vectoriales tienen algunas propiedades importantes que resuelven la mayoría de los problemas antes mencionados:

- El cálculo de cada resultado es independiente de los cálculos de los resultados anteriores, permitiendo un gran nivel de segmentación *sin* generar ningún riesgo por dependencias de datos. Esencialmente, la ausencia de riesgos por dependencias de datos la determina el compilador o el programador cuando deciden que se puede utilizar una instrucción vectorial.
- Una simple instrucción vectorial especifica una gran cantidad de trabajo —es equivalente a ejecutar un bucle completo—. Por tanto, el requerimiento de anchura de banda de las instrucciones es reducido, y el cuello de botella de Flynn se reduce considerablemente.
- Las instrucciones vectoriales que acceden a memoria tienen un patrón de acceso conocido. Si los elementos del vector son todos adyacentes, entonces extraer el vector de un conjunto de bancos de memoria entrelazados funciona muy bien. La alta latencia de iniciar un acceso a memoria principal, en comparación con acceder a una cache se amortiza porque se inicia un acceso para el vector completo en lugar de para un único elemento. Por ello, el coste de la latencia a memoria principal se paga sólo una vez para el vector completo, en lugar de una vez por cada elemento del vector.
- Como se sustituye un bucle completo por una instrucción vectorial cuyo comportamiento está predeterminado, los riesgos de control que normalmente podían surgir del salto del bucle son inexistentes.

Por estas razones, las operaciones vectoriales pueden hacerse más rápidas que una secuencia de operaciones escalares sobre el mismo número de elementos

de datos, y los diseñadores están motivados para incluir unidades vectoriales si el conjunto de las aplicaciones las puede usar frecuentemente.

Como mencionamos antes, los procesadores vectoriales segmentan las operaciones sobre los elementos de un vector. La segmentación, no sólo incluye las operaciones aritméticas (multiplicación, suma, etc.), sino también los accesos a memoria y el cálculo de direcciones efectivas. Además, la mayor parte de las máquinas vectoriales de altas prestaciones permiten que se hagan múltiples operaciones vectoriales a la vez, creando paralelismo entre las operaciones sobre diferentes elementos. En este capítulo, nos centraremos sobre máquinas vectoriales que mejoran el rendimiento gracias a la segmentación y al solapamiento de las instrucciones. En el Capítulo 10, explicaremos máquinas paralelas que operan sobre muchos elementos en paralelo en lugar de aplicar técnicas de segmentación.

7.2 Arquitectura vectorial básica

Una máquina vectorial, normalmente, consta de una unidad escalar segmentada más una unidad vectorial. Todas las unidades funcionales de la unidad vectorial tienen una latencia de varios ciclos de reloj. Esto permite un ciclo de reloj de menor duración y es compatible con operaciones vectoriales de larga ejecución que pueden ser segmentadas a nivel alto sin generar riesgos. La mayoría de las máquinas vectoriales permiten que los vectores sean tratados como números en punto flotante (FP), como enteros, o como datos lógicos, aunque nos centraremos en el punto flotante. La unidad escalar no es básicamente diferente del tipo de CPU segmentada explicada en el Capítulo 6.

Hay dos tipos principales de arquitecturas vectoriales: máquinas vectoriales con registros y máquinas vectoriales memoria-memoria. En una *máquina vectorial con registros*, todas las operaciones vectoriales —excepto las de carga y almacenamiento— operan con vectores almacenados en los registros. Estas máquinas son el equivalente vectorial de una arquitectura escalar de carga/almacenamiento. Todas las máquinas vectoriales importantes construidas en 1990 utilizan una arquitectura vectorial con registros; éstas incluyen las máquinas de Cray Research (CRAY-1, CRAY-2, X-MP e Y-MP), los supercomputadores japoneses (NEC SX/2, Fujitsu VP200 y el Hitachi S820) y los minisupercomputadores (Convex C-1 y C-2). En una *máquina vectorial memoria-memoria* todas las operaciones vectoriales son de memoria a memoria. Las primeras máquinas vectoriales fueron de este tipo, como por ejemplo las máquinas de CDC. A partir de este punto nos centraremos sólo en arquitecturas vectoriales con registros; volveremos brevemente a las arquitecturas vectoriales memoria-memoria al final del capítulo (Sección 7.8) para explicar por qué no han tenido el éxito de las arquitecturas vectoriales con registros.

Comenzamos con una máquina vectorial con registros que consta de los componentes principales mostrados en la Figura 7.1. Esta máquina, que está aproximadamente basada en el CRAY-1, es la base de las explicaciones de la mayor parte de este capítulo. La llamaremos DLXV; su parte entera es DLX, y su parte vectorial es la extensión vectorial lógica de DLX. El resto de esta

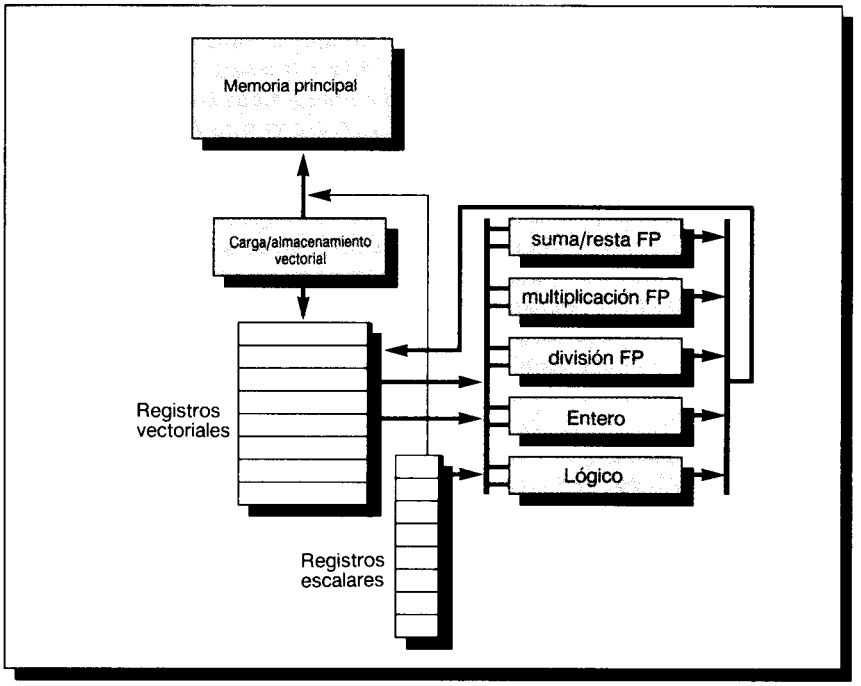


FIGURA 7.1 Estructura básica de una arquitectura con registros vectoriales, DLXV. Esta máquina tiene una arquitectura escalar como DLX. También hay ocho registros vectoriales de 64 elementos, y todas las unidades funcionales son vectoriales. Operaciones vectoriales especiales y cargas y almacenamientos de vectores están definidos. Mostramos unidades vectoriales para operaciones lógicas y enteras. Estas se incluyen para que DLXV sea como una máquina vectorial estándar, que eventualmente incluye estas unidades. Sin embargo, no explicaremos aquí estas unidades excepto en los ejercicios. En la Sección 7.6 añadimos el encadenamiento, que requerirá capacidad adicional de interconexión.

sección examina cómo la arquitectura básica de DLX está relacionada con otras máquinas.

Los componentes principales del conjunto de instrucciones de la máquina de DLXV son:

- Registros vectoriales. Cada registro vectorial es un banco de longitud fija que contiene un solo vector. DLXV tiene ocho registros vectoriales, y cada registro vectorial contiene 64 dobles palabras. Cada registro vectorial debe tener como mínimo dos accesos de lectura y un acceso de escritura en DLXV. Esto permitirá un alto grado de solapamiento entre las operaciones vectoriales que usan diferentes registros vectoriales. (El CRAY-1 gestiona la implementación del conjunto de registros con un sólo acceso por registro utilizando algunas técnicas de implementación inteligentes.)
- Unidades funcionales vectoriales. Cada unidad está completamente segmentada y puede comenzar una operación nueva en cada ciclo de reloj. Se

necesita una unidad de control para detectar riesgos, sobre conflictos en las unidades funcionales (riesgos estructurales) y sobre conflictos en los accesos a registros (riesgos por dependencias de datos). DLXV tiene cinco unidades funcionales, como se muestra en la Figura 7.1. Por simplicidad, nos centraremos exclusivamente en las unidades funcionales de punto flotante.

- Unidad de carga/almacenamiento de vectores. Es una unidad que carga o almacena un vector en o desde memoria. Las cargas y almacenamientos vectoriales de DLXV están completamente segmentadas, para que las palabras puedan ser transferidas entre los registros vectoriales y memoria con un ancho de banda de una palabra por ciclo de reloj, después de una latencia inicial.
- Un conjunto de registros escalares. Estos también pueden proporcionar datos como entradas a las unidades funcionales vectoriales, así como calcular direcciones para pasar a la unidad de carga/almacenamiento de vectores. Estos son los 32 registros normales de propósito general y los 32 registros de punto flotante de DLX.

La Figura 7.2 muestra las características de algunas máquinas vectoriales típicas, incluyendo el tamaño y número de registros, el número y tipos de unidades funcionales y el número de unidades de carga/almacenamiento.

En DLXV, la operación vectorial tiene el mismo nombre que en DLX añadiéndole la letra «V». Estas son operaciones vectoriales de punto flotante y doble precisión. (Por simplicidad hemos omitido operaciones FP en simple precisión y operaciones lógicas y enteras.) Por tanto, ADDV es una suma de dos vectores en doble precisión. Las operaciones vectoriales toman como entrada o un par de registros vectoriales (ADDV) o un registro vectorial y un registro escalar , lo que se designa añadiendo «SV» (ADDSV). En el último caso, el valor del registro escalar se utiliza como entrada para todas las operaciones —la operación ADDSV sumará el contenido de un registro escalar a cada elemento de un registro vectorial. Las operaciones vectoriales siempre tienen como destino un registro vectorial. Los nombres LV y SV denotan cargar vector y almacenar vector, y cargar o almacenar un vector completo de datos en doble precisión. Un operando es el registro vectorial que se va a cargar o almacenar; el otro operando, que es un registro de propósito general de DLX, tiene la dirección de comienzo del vector en memoria. La Figura 7.3 lista las instrucciones vectoriales de DLXV. Además de los registros vectoriales, necesitamos dos registros adicionales de propósito general: los registros de longitud de vector y de máscara de vector. Explicaremos estos registros y sus propósitos en las Secciones 7.3 y 7.6, respectivamente.

Una máquina vectorial se comprende mejor examinando un bucle vectorial en DLXV. Examinemos un problema vectorial típico, que se utilizará a través de este capítulo:

$$Y = a \cdot X + Y$$

X e Y son vectores que, inicialmente residen en memoria, y a es un escalar. Este es el bucle llamado SAXPY o DAXPY (Simple precisión o Doble precisión A · X Más *(Plus)* Y) que forma el bucle interno del benchmark de Lin-

Máquina	Anunciada año	Registros vectoriales	Elementos por registro vectorial (elementos de 64-bit)	Unidades funcionales vectoriales	Unidades vectoriales de carga/ almacenamiento
CRAY-1	1976	8	64	6: suma, multiplicación, recíproco, suma entera, lógica, desplazamiento	1
CRAY X-MP CRAY Y-MP	1983 1988	8	64	8: suma FP, multiplicación FP, recíproco FP, suma entera, 2 lógicas, desplazamiento, población cuenta/ paridad	2 cargas 1 almacenamiento
CRAY-2	1985	8	64	5: suma FP, multiplicación FP, recíproco raíz cuadrada FP, (suma desplazamiento, cuenta población) entera, lógica	1
Fujitsu VP100/200	1982	8-256	32-1024	3: suma/lógica entera o FP	2
Hitachi S810/820	1983	32	256	4: 2 suma entera/lógica, 1 multiplicación-suma y 1 multiplicación/división-suma	4
Convex C-1	1985	8	128	4: multiplicación, suma, división, entera/lógica	1
NEC SX/2	1984	8 + 8192	256 variable	16: 4 suma entera/lógica, 4 multiplicación/división FP, 4 suma FP, 4 desplazamientos	8
DLXV	1990	8	64	5: multiplicación, división, suma, suma entera, lógica	1

FIGURA 7.2 Características de algunas arquitecturas vector-registro. Las unidades funcionales incluyen todas las unidades de operación utilizadas por las instrucciones vectoriales. Las unidades vectoriales son de punto flotante a menos que se indique otra cosa. Si la máquina es un multiprocesador, las entradas corresponden a las características de un procesador. Cada unidad vectorial de carga/almacenamiento representa la posibilidad de realizar una transferencia independiente solapada a o desde registros vectoriales. Los registros vectoriales de Fujitsu VP200 son configurables. El tamaño y número de las 8K entradas de 64 bits puede variarse inversamente (por ejemplo, 8 registros cada uno de 1K elementos, o 128 registros cada uno de 64 elementos). El NEC SX/2 tiene 8 registros fijos de longitud 256, más 8K registros configurables de 64 bits. La unidad recíproca en las máquinas CRAY se utiliza para hacer divisiones (y raíces cuadradas en el CRAY-2). La unidad de suma realiza la suma y resta en punto flotante. La unidad de multiplicación/división-suma en la Hitachi S810/200 realiza una multiplicación o división FP seguida por una suma o resta (mientras que la unidad de multiplicación realiza una multiplicación seguida por una suma o resta). Observar que la mayoría de las máquinas utilizan las unidades vectoriales de multiplicación y división FP para la multiplicación y división vectoriales enteras, justo como DLX, y algunas máquinas utilizan las mismas unidades para las operaciones vectoriales y escalares FP.

pack. Linpack es una colección de rutinas de álgebra lineal; la parte de eliminación Gaussiana de Linpack es la parte utilizada como benchmark. SAXPY representa una pequeña parte del programa, aunque consume la mayor parte del tiempo en el benchmark.

Por ahora, supongamos que el número de elementos, o longitud, de un registro vectorial (64) coincide con la longitud de la operación vectorial en la que estamos interesados. (Esta restricción se dejará dentro de poco.)

Instrucción vectorial	Operandos	Función
ADDV	V1,V2,V3	Suma elementos de V2 y V3, después pone cada resultado en V1
ADDSV	V1,F0,V2	Suma F0 a cada elemento de V2, después pone cada resultado en V1
SUBV	V1,V2,V3	Resta elementos de V3 desde V2, después pone cada resultado en V1.
SUBVS	V1,V2,F0	Resta F0 de los elementos de V2, después pone cada resultado en V1.
SUBSV	V1,F0,V2	Resta elementos de V2 de F0, después pone cada resultado en V1
MULTV	V1,V2,V3	Multiplica elementos de V2 y V3, después pone cada resultado en V1.
MULTSV	V1,F0,V2	Multiplica F0 por cada elemento de V2, después pone cada resultado en V1.
DIVV	V1,V2,V3	Divide elementos de V2 por V3, después pone cada resultado en V1.
DIVVS	V1,V2,F0	Divide elementos de V2 por F0, después pone cada resultado en V1.
DIVSV	V1,F0,V2	Divide F0 por elementos de V2, después pone cada resultado en V1.
LV	V1,R1	Carga registro vectorial V1 desde memoria comenzando en la dirección R1.
SV	R1,V1	Almacena registro vectorial V1 en memoria comenzando en la dirección R1.
LVWS	V1,(R1,R2)	Carga V1 desde la dirección en R1 con separación en R2, p. ej., R1+i·R2.
SVWS	(R1,R2),V1	Almacena V1 desde dirección en R1 con separación en R2, p. ej., R1+i·R2
LVI	V1,(R1+V2)	Carga V1 con un vector cuyos elementos están en R1+V2(i), es decir, V2 es un índice.
SVI	(R1+V2),V1	Almacena V1 con un vector cuyos elementos están en R1+V2(i), es decir, V2 es un índice.
CVI	V1,R1	Crea un vector de índices almacenando en V1 los valores 0, 1·R1, 2·R1,...,63·R1 en V1.
S_V S_SV	V1,V2 F0,V1	Compara (EQ, NE, GT, LT, GE, LE) los elementos de V1 y V2. Si la condición es cierta pone un 1 en el bit correspondiente del vector; en cualquier otro caso pone 0. Pone el vector de bits resultante en un registro de máscara vectorial (VM). La instrucción S_SV realiza la misma comparación pero utilizando un valor escalar como operando.
POP	R1,VM	Cuenta los 1s en el registro de máscara vectorial y almacena la cuenta en R1.
CVM		Pone todo el registro de máscara vectorial a 1.
MOVI2S	VLR,R1	Transfiere el contenido de R1 al registro de longitud vectorial.
MOVS2I	R1,VLR	Transfiere el contenido de registro de longitud vectorial a R1.
MOVF2S	VM,F0	Transfiere el contenido de F0 al registro de máscara vectorial.
MOVS2F	F0,VM	Transfiere el contenido del registro de máscara vectorial a F0.

FIGURA 7.3 Las instrucciones vectoriales DLXV. Sólo se muestran las operaciones FP en doble precisión. Además de los registros vectoriales hay dos registros especiales VLR (explicado en la Sección 7.3) y VM (explicado en la Sección 7.6). Las operaciones con separación entre elementos se explican en la Sección 7.3, y el uso de la creación de índices y operaciones indexadas de carga/almacenamiento se explican en la Sección 7.6.

Ejemplo Mostrar el código para DLX y DLXV para el bucle DAXPY. Suponer que las direcciones de comienzo de X e Y están en Rx y Ry, respectivamente.

Respuesta Este es el código de DLX

```
      LD      F0,a
      ADDI    R4,Rx,#512   ;última dirección a cargar
loop:
      LD      F2,0(Rx)     ;carga X(i)
      MULTD   F2,F0,F2     ;a·X(i)
      LD      F4,0(Ry)     ;carga Y(i)
      ADDD    F4,F2,F4     ;a·X(i) + Y(i)
      SD      F4,0(Ry)     ;almacena en Y(i)
      ADDI    Rx,Rx,#8     ;incrementa índice a X
      ADDI    Ry,Ry,#8     ;incrementa índice a Y
      SUB     R20,R4,Rx    ;calcula límite
      BNZ     R20,loop     ;comprobación si se ha terminado
```

Aquí está el código DLXV para DAXPY.

```
      LD      F0,a         ;carga escalar a
      LV      V1,Rx        ;carga vector X
      MULTSV  V2,F0,V1     ;multiplicación vector-escalar
      LV      V3,Ry        ;carga vector Y
      ADDV    V4,V2,V3     ;suma
      SV      Ry,V4        ;almacena el resultado
```

Hay algunas comparaciones interesantes entre los dos segmentos de código del ejemplo anterior. Lo más espectacular es que la máquina vectorial reduce enormemente el número de instrucciones que realmente se ejecutan, ejecutando sólo 6 instrucciones frente a casi 600 para DLX. Esta reducción ocurre porque las operaciones vectoriales trabajan sobre 64 elementos, y porque las instrucciones de sobrecarga adicional, que constituyen aproximadamente la mitad del bucle en DLX no están presentes en el código de DLXV.

Otra diferencia importante es la frecuencia de interbloqueos de las etapas. En el sencillo código de DLX cada ADDD debe esperar por un MULTD, y cada SD debe esperar por un ADDD. En la máquina vectorial, cada instrucción vectorial opera sobre todos los elementos del vector independientemente. Entonces las detenciones de la segmentación se requieren sólo una vez por operación vectorial, en lugar de una vez por elemento del vector. En este ejemplo, la frecuencia de detenciones de la segmentación en DLX será aproximadamente 64 veces mayor que en DLXV. Las detenciones de la segmentación se pueden eliminar en DLX utilizando segmentación software o desenrollamiento de bucles (como vimos en el Capítulo 6, Sección 6.8). Sin embargo, no se puede reducir la gran diferencia del ancho de banda de las instrucciones.

Velocidad de iniciación y tiempo de arranque vectorial

Investiguemos el tiempo de ejecución de este código vectorial en DLXV. El tiempo de ejecución de cada operación vectorial del bucle tiene dos compo-

nentes —el *tiempo de arranque (start-up)* y la *velocidad de iniciación.* El tiempo de arranque depende de la latencia de las etapas o segmentos de la operación vectorial y está determinado principalmente por el número de etapas necesarias de la unidad funcional utilizada. Por ejemplo, una latencia de 10 ciclos de reloj significa que la operación tarda 10 ciclos de reloj y que se necesita pasar por 10 etapas. (En discusiones de rendimiento de operaciones vectoriales, es costumbre utilizar como métrica los ciclos de reloj.) La velocidad de iniciación es el tiempo por resultado una vez que una instrucción vectorial está en ejecución; esta frecuencia, habitualmente, es uno por ciclo de reloj para operaciones individuales, aunque algunos supercomputadores tienen operaciones vectoriales que pueden producir dos o más resultados por reloj, y otros tienen unidades que no pueden admitir menos datos en cada ciclo. La *velocidad de terminación* debe igualar como mínimo a la velocidad de iniciación —en otro caso no hay sitio para poner resultados. Por consiguiente, el tiempo para completar una operación vectorial de longitud n es:

$$\text{Tiempo de arranque} + n \cdot \text{Velocidad de iniciación}$$

Ejemplo

Supongamos que el tiempo de arranque para multiplicar un vector es de 10 ciclos de reloj. Después de arrancar, la velocidad de iniciación es de uno por ciclo de reloj. ¿Cuál es el número de ciclos de reloj por resultado (p. e., un elemento del vector) para un vector de 64 elementos?

Respuesta

$$\text{Ciclos de reloj por resultado} = \frac{\text{Tiempo total}}{\text{Longitud del vector}}$$

$$= \frac{\text{Tiempo de arranque} + 64 \cdot \text{Velocidad de iniciación}}{64}$$

$$= \frac{10 + 64}{64} = 1{,}16 \text{ ciclos de reloj}$$

La Figura 7.4 muestra el efecto del tiempo de arranque y de la velocidad de iniciación sobre el rendimiento vectorial. El efecto de incrementar el tiempo de arranque en un vector de ejecución lenta es pequeño, mientras que el mismo incremento en el tiempo de arranque en un sistema con una velocidad de iniciación de 1 por ciclo de reloj disminuye el rendimiento en un factor de aproximadamente 2.

¿Qué determina las velocidades de arranque e iniciación? Consideremos primero las operaciones que no involucran accesos a memoria. Para operaciones registro-registro el tiempo de arranque (en ciclos de reloj) es igual al número de etapas de la unidad funcional, ya que éste es el tiempo para obtener el primer resultado. En el ejemplo anterior, la profundidad 10 daba un tiempo de arranque de 10 ciclos de reloj. En las siguientes secciones, veremos que hay otros costes involucrados que incrementan el tiempo de arranque. La velocidad de iniciación está determinada por la frecuencia con que la unidad funcional vectorial correspondiente pueda aceptar nuevos operandos. Si está totalmente segmentada, entonces puede empezar una operación sobre nuevos

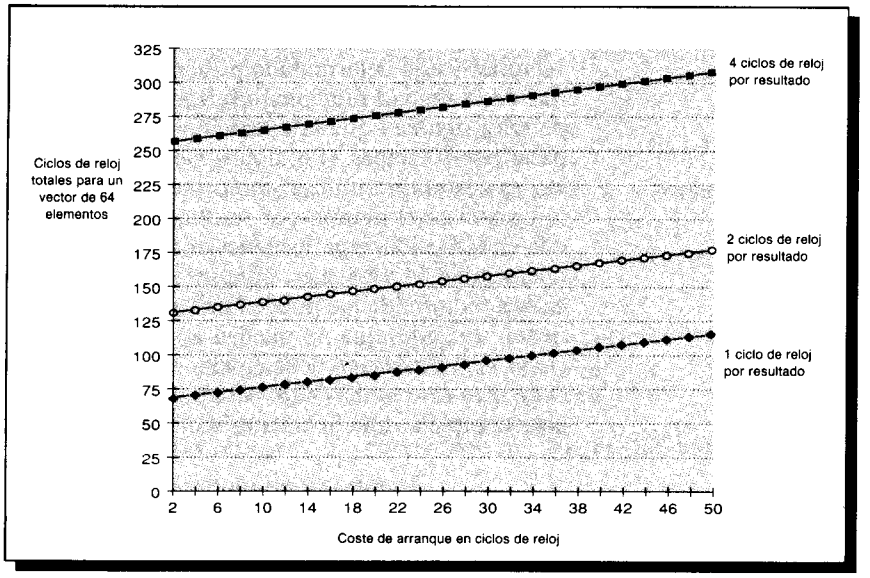


FIGURA 7.4 El tiempo total de ejecución se incrementa con el coste de arranque desde 2 a 50 ciclos de reloj por operación en el eje x. El impacto del tiempo de arranque es mucho mayor para vectores de ejecución rápida que para vectores de ejecución lenta. La ejecución de operaciones a un ciclo de reloj por resultado incrementa su tiempo de ejecución un 75 por 100, mientras que la ejecución de una operación a cuatro ciclos de reloj por resultado incrementa en menos del 20 por 100.

operandos cada ciclo de reloj, dando una velocidad de iniciación de 1 por ciclo de reloj (como en el ejemplo anterior).

El tiempo de arranque para una operación comprende la latencia total de la unidad funcional implementando esa operación. Si la velocidad de iniciación se mantiene a un ciclo de reloj por resultado, entonces:

$$\text{Profundidad de la segmentación} = \left\lceil \frac{\text{Tiempo total de la unidad funcional}}{\text{Duración del ciclo de reloj}} \right\rceil$$

Por ejemplo, si una operación necesita 10 ciclos de reloj, debe tener una profundidad de la segmentación de 10 para lograr una velocidad de iniciación de 1 por reloj. La profundidad de la segmentación, se determina entonces por la complejidad de la operación y la duración del ciclo de reloj de la máquina. El número de segmentos o etapas de las unidades funcionales varían ampliamente —de 2 a 20 etapas no es infrecuente— aunque las unidades más intensamente utilizadas tengan tiempos de arranque de 4 a 8 ciclos de reloj.

Para DLXV, elegiremos los mismos valores de número de la CRAY-1. Todas las unidades funcionales están totalmente segmentadas. Los tiempos requeridos son seis ciclos de reloj para sumas en punto flotante y de siete ciclos de reloj para multiplicaciones en punto flotante. Si un cálculo vectorial depende de un cálculo incompleto y necesitase ser detenido, hay que añadir

una penalización extra de arranque de 4 ciclos de reloj. Esta penalización es normal en máquinas vectoriales y surge debido a la falta de cortocircuitos entre unidades funcionales: la penalización es el tiempo para escribir y después leer los operandos y existe sólo cuando hay una dependencia. Así, las operaciones vectoriales que están próximas y tienen dependencias, verán la latencia completa de una operación vectorial. En DLXV, como en muchas máquinas vectoriales, las operaciones vectoriales independientes, que utilizan diferentes unidades funcionales, pueden realizarse sin ninguna penalización o retardo. Las operaciones vectoriales independientes también pueden estar completamente solapadas, y la emisión de cada instrucción sólo necesita un ciclo de reloj. Por tanto, cuando las operaciones son independientes y diferentes, DLXV puede solapar operaciones vectoriales, al igual que DLX puede solapar operaciones enteras y en punto flotante.

Como DLXV está totalmente segmentado, la velocidad de iniciación para una instrucción vectorial es siempre 1. Sin embargo, una secuencia de operaciones vectoriales no se podrá ejecutar a esa velocidad, debido a los costes de arranque. Debido a esto, se usa el término de velocidad sostenida, que significa el tiempo por elemento para un conjunto de operaciones vectoriales relacionadas. Aquí, un elemento no es el resultado de una única operación vectorial, sino un resultado de una serie de operaciones vectoriales. El tiempo por elemento, entonces, es el tiempo requerido para que cada operación produzca un elemento. Por ejemplo, en el bucle SAXPY, la velocidad sostenida será el tiempo para calcular y almacenar un elemento del vector resultado Y.

Ejemplo

Para un vector de longitud 64 en DLXV y las dos siguientes instrucciones vectoriales, ¿cuál es la velocidad sostenida de la secuencia, y el número efectivo de operaciones de punto flotante por ciclo reloj para la secuencia?

```
MULTV V1,V2,V3
ADDV  V4,V5,V6
```

Respuesta

Examinemos los tiempos de inicialización y finalización de estas operaciones independientes (recordar que los tiempos de arranque son 7 ciclos para la multiplicación y 6 ciclos para la suma):

Operación	Comienza	Finaliza
MULTV	0	7 + 64 = 71
ADDV	1	1 + 6 + 64 = 71

La velocidad sostenida es un elemento por reloj —recordar que la velocidad sostenida requiere que todas las operaciones vectoriales produzcan un resultado. La secuencia ejecuta 128 FLOP (OPeraciones en punto FLotante) en 71 ciclos de reloj, dando una velocidad de 1,8 FLOP por reloj. Una máquina vectorial puede sostener una productividad de más de una operación por ciclo de reloj al emitir operaciones vectoriales independientes en diferentes unidades funcionales vectoriales.

El comportamiento de la unidad vectorial de carga/almacenamiento es significativamente más complicado. El tiempo de arranque para una carga es el tiempo necesario para leer la primera palabra de memoria y escribirla en un registro. Si el resto del vector se puede suministrar sin detenciones, entonces la velocidad de iniciación del vector es igual a la velocidad a la que se extraen o almacenan las siguientes palabras. Normalmente, las penalizaciones para arranques en las unidades de carga o almacenamiento son mayores que para las unidades funcionales —hasta 50 ciclos de reloj en algunas máquinas. Para DLXV supondremos un tiempo de arranque de lectura bajo de 12 ciclos de reloj, ya que el CRAY-1 y el CRAY X-MP tienen tiempos de arranque de carga/almacenamiento entre 9 y 17 ciclos de reloj. Para las escrituras en memoria, habitualmente, no cuidaremos el tiempo de arranque, ya que no producen directamente resultados. Sin embargo, cuando una instrucción debe esperar que una escritura en memoria finalice (como puede ser el caso de una lectura de memoria, ya que no sólo existe un bus entre procesador y memoria), la lectura puede ver parte o toda la latencia de 12 ciclos de una escritura en memoria. La Figura 7.5 resume las penalizaciones de arranque para las operaciones vectoriales de DLXV.

Para mantener una velocidad de iniciación de una palabra leída o escrita de/en memoria por ciclo de reloj, el sistema de memoria debe ser capaz de producir o aceptar esa cantidad de datos. Esto se hace, habitualmente, teniendo varios *bancos de memoria.* Cada banco de memoria permite realizar una lectura o escritura de una palabra independiente de los otros bancos. Así pues, las palabras se transfieren desde memoria a la máxima frecuencia (una por ciclo de reloj en DLXV).

La primera de ellas es sincronizar todos los bancos, de forma que se accede a ellos en paralelo y al misma instante. Una vez han sido, por siempre, leídos, se guardan juntos todos los resultados y, mientras se transfieren uno a uno al procesador, los bancos comienzan a servir otro grupo de peticiones. La segunda es que el acceso a los bancos sea asíncrono o desfasado. En esta forma, y después del primer acceso en el que todos los bancos se acceden en paralelo, las palabras leídas son enviadas al procesador una a una desde cada banco.

Operación	Penalización de arranque
Suma vectorial	6
Multiplicación vectorial	7
División vectorial	20
Carga vectorial	12

FIGURA 7.5 Penalizaciones de arranque en DLXV. Estas son las penalizaciones de arranque en ciclos de reloj para las operaciones vectoriales de DLXV. Cuando una instrucción vectorial depende de otra que no se ha completado en el instante que aparece la segunda instrucción vectorial, la penalización de arranque aumenta en 4 ciclos de reloj.

Una vez un banco ha transmitido o almacenado su dato, comienza el próximo servicio inmediatamente. La primera aproximación (accesos sincronizados) necesita más elementos para memorizar los resultados, pero tiene un control más simple que la aproximación que utilice acceso independiente a los bancos. El concepto de banco de memoria es similar, pero no idéntico, al de entrelazado, como veremos en la Figura 7.6. Explicaremos ampliamente el entrelazado en el Capítulo 8, Sección 8.4.

Suponiendo que cada banco tiene un ancho de una palabra en doble precisión, si se mantiene una velocidad de iniciación de uno por reloj, debe cumplirse que:

$$\text{Número de bancos de memoria} \geqslant \begin{array}{c}\text{Tiempo de acceso al banco de memoria}\\ \text{en ciclos de reloj}\end{array}$$

Para ver por qué existe esta relación, pensar en leer un vector de 64 palabras de doble precisión. Suponer las direcciones de los elementos del vector dadas por k_i, donde

k_i = Dirección del primer elemento del vector + $(i - 1)$
· Distancia entre elementos consecutivos del vector

Para los elementos del vector de doble precisión que son adyacentes, la distancia entre elementos será de 8 bytes. Las direcciones de los elementos del vector que van a ser accedidos por un banco serán los valores de k_i tales que

$$k_i \bmod \text{Número de banco} = 0$$

Examinemos el primer acceso para cada banco. Después de un tiempo igual al tiempo de acceso a memoria, todos los bancos de memoria habrán extraido una palabra en doble precisión, y las palabras pueden empezar a volver a los registros del vector. (Esto requiere, por supuesto, que los accesos estén alineados en límites de dobles palabras.) Las palabras se envían en serie desde los bancos, comenzando con la extraida del banco con la dirección más baja. Si los bancos están sincronizados, los accesos siguientes comienzan inmediatamente. Si los bancos funcionan de manera asíncrona, entonces el acceso siguiente comienza después que un elemento se transmite desde el banco. En cualquier caso, un banco comienza su siguiente acceso en una dirección de bytes que es (8 · número de bancos) veces mayor que la dirección del último byte. Debido a que el tiempo de acceso a memoria en ciclos de reloj es menor que el número de bancos de memoria, y a que las palabras se transfieren desde los bancos en orden de petición a la velocidad de una transferencia por ciclo de reloj, un banco completará el acceso siguiente antes que empiece de nuevo su turno para la transmisión de datos. Para simplificar direccionamientos, el número de bancos de memoria, normalmente, es una potencia de dos. Como veremos dentro de poco, los diseñadores, en general, quieren tener un número de bancos superior al mínimo necesario para minimizar los retardos, al acceder a memoria.

Ejemplo

Suponer que se desea leer un vector de 64 elementos, comenzando en el byte de dirección 136, y un acceso a memoria necesita 6 ciclos de reloj. ¿Cuántos bancos de memoria debe haber? ¿Con qué direcciones se accede a los bancos de memoria? ¿Cuándo llegan a la CPU los diversos elementos?

Respuesta

Seis ciclos de reloj por acceso requieren al menos 6 bancos, pero como queremos que el número de bancos sea una potencia de dos, elegimos entonces 8 bancos. La Figura 7.6 muestra las direcciones a nivel de byte que cada banco accede en cada período de tiempo. Recordar que un banco comienza un nuevo acceso tan pronto como ha completado el anterior acceso.

La Figura 7.7 muestra el diagrama de tiempos para los primeros conjuntos

Comienza en núm. de reloj	Banco 0	1	2	3	4	5	6	7
0	192	136	144	152	160	168	176	184
6	256	200	208	216	224	232	240	248
14	320	264	272	280	288	296	304	312
22	384	328	336	344	352	360	368	376

FIGURA 7.6 Direcciones de memoria (en bytes) por número de bancos e instante de tiempo en que comienza el acceso. El instante exacto en que un banco transmite sus datos está dado por la dirección a la que éste accede menos la dirección de comienzo dividida por 8 más la latencia de memoria (6 ciclos de reloj). Es importante observar que el Banco 0 accede una palabra del bloque siguiente (p. e., accede 192 en lugar de 128 y después 256 en lugar de 192, y así sucesivamente). Si el Banco 0 comenzase en la dirección más baja necesitaríamos un ciclo extra para transmitir el dato, y transmitiríamos un valor innecesariamente. Aunque este problema no es severo para este ejemplo, si tuviésemos 64 bancos, podrían presentarse hasta 63 ciclos de reloj y transferencias innecesarias. El hecho de que el Banco 0 no acceda a una palabra en el mismo bloque de 8 distingue a este tipo de sistemas de memoria de la memoria entrelazada. Normalmente, los sistemas de memoria entrelazada combinan la dirección del banco y la dirección base de comienzo por concatenación en lugar de por adición. También, las memorias entrelazadas se implementan casi siempre con accesos sincronizados. Los bancos de memoria requieren cerrojos de dirección para cada banco, que normalmente no son necesarios en un sistema con solo entrelazado.

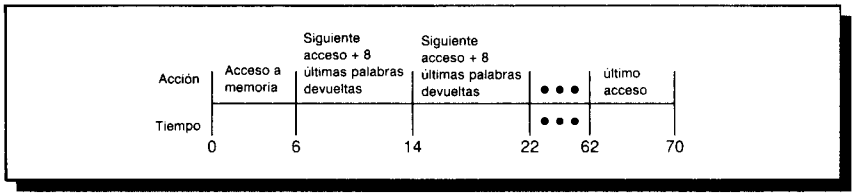


FIGURA 7.7 Temporización de accesos para las 64 primeras palabras de la carga en doble precisión. Después de la latencia inicial de 6 ciclos de reloj, cada 8 ciclos de reloj se devuelven 8 palabras en doble precisión.

de accesos en un sistema de 8 bancos con una latencia de acceso de 6 ciclos de reloj. Dos observaciones importantes sobre estas dos figuras son: primero, observar que la dirección exacta extraida por un banco está determinada por los bits de orden inferior del número de banco; sin embargo, el acceso inicial a un banco está siempre en 8 dobles palabras de la dirección inicial. Segundo, observar que una vez que se supera la latencia inicial (6 ciclos de reloj en este caso), el patrón es acceder un banco cada n ciclos de reloj, donde n es el número total de bancos ($n = 8$ en este caso).

El número de bancos del sistema de memoria y el número de segmentos usados en las unidades funcionales son esencialmente conceptos equivalentes, ya que determinan las velocidades de iniciación para las operaciones que utilizan estas unidades. El procesador no puede acceder a memoria con más rapidez que la duración del ciclo de memoria. Por tanto, si la memoria se construye con DRAM, donde la duración del ciclo es aproximadamente dos veces el tiempo de acceso, el procesador habitualmente necesitará dos veces tantos bancos como el obtenido en los cálculos realizados antes. Esta característica de las DRAM se explica posteriormente, en el Capítulo 8, Sección 8.4.

7.3 Dos aspectos del mundo real: longitud del vector y separación entre elementos

Esta sección trata con dos aspectos que se presentan en los programas reales. Estos son: qué hacer cuando la longitud de los vectores de un programa no es exactamente 64, y qué hacer cuando los elementos consecutivos de un vector no están adyacentes en memoria después de que la matriz se ha almacenado en memoria. Primero, trataremos con el aspecto de la longitud del vector.

Control de la longitud del vector

Una máquina con registros vectoriales o bancos de registros tiene una longitud natural de los vectores determinada por el número de elementos de cada registro vectorial. Esta longitud, que es 64 para DLXV, es improbable que coincida con la longitud real de los vectores en un programa. Además, en un programa real, la longitud de una operación vectorial particular es, con frecuencia, desconocida en tiempo de compilación. En efecto, una parte de código puede requerir diferentes longitudes de vectores. Por ejemplo, considerar el código:

```
      do 10 i = 1,n
10      Y(i) = a · X(i) + Y(i)
```

El tamaño de todas las operaciones vectoriales depende de `n`, ¡que puede incluso no conocerse hasta el momento de la ejecución! El valor de `n` puede también ser un parámetro para el procedimiento y, por tanto, estar sujeto a cambio durante la ejecución.

La solución a estos problemas es crear un *registro de longitud vectorial* (VLR). El VLR controla la longitud de cualquier operación vectorial, incluyendo la carga o almacenamiento de un vector. Sin embargo, el valor en el VLR no puede ser mayor que la longitud de los registros vectoriales. Esto resuelve nuestro problema mientras que la longitud real sea menor que la *longitud máxima del vector* (MVL) definida por la máquina.

¿Qué ocurre si el valor de n no se conoce en tiempo de compilación, y, además, puede ser mayor que MVL? Para abordar este problema, se utiliza una técnica denominada *seccionamiento* (*strip mining*). El seccionamiento es la generación de código tal que cada operación vectorial se realiza para un tamaño menor o igual que la MVL. La versión seccionada del bucle SAXPY escrito en FORTRAN, el principal lenguaje utilizado para aplicaciones científicas, se muestra con comentarios estilo C:

```
      low = 1
      VL = (n mod MVL)  /*encontrar remanente del vector*/
      do 1 j = 0,(n / MVL)  /*bucle exterior*/
         do 10 i = low,low+VL-1  /*corre para longitud VL*/
            Y(i) = a·X(i) + Y(i)  /*operación principal*/
10       continue
         low = low+VL  /*comienzo de siguiente vector*/
         VL = MVL  /*reinicializa la longitud a max*/
1        continue
```

El término n / MVL representa la división entera truncada (que es lo que hace FORTRAN) y se usa en toda esta sección. El efecto de este bucle es dividir al vector en secciones que después son procesadas por el bucle interior. La longitud de la primera sección (ship) es (n *módulo* MVL) y la de todos los segmentos siguientes MVL. Esto se indica en la Figura 7.8.

El bucle interior del código anterior es vectorizable con longitud VL, que es igual a (n mod MVL) o MVL. El registro VLR se debe inicializar dos veces —una vez en cada lugar donde se asigna la variable VL del código. Con múltiples operaciones vectoriales ejecutándose en paralelo, el hardware debe co-

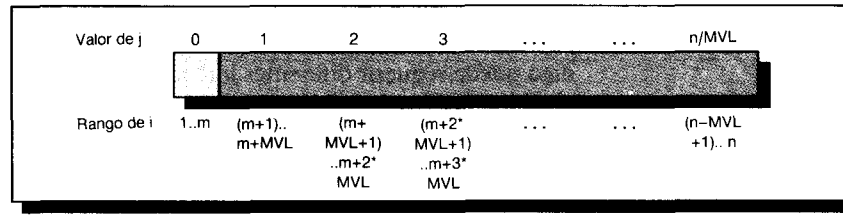


FIGURA 7.8 Un vector de longitud arbitraria procesado con seccionamiento (strip mining). Todos los bloques, excepto el primero, son de longitud MVL, utilizando la potencia completa de la máquina vectorial. En esta figura, la variable *m* se utiliza para la expresión (n mod MVL).

piar el valor de VLR cuando se emita una operación vectorial, en el caso que VLR sea cambiado por una operación vectorial posterior.

En la sección anterior, los costes de arranque podían calcularse independientemente para cada operación vectorial. Con el seccionamiento de bandas, un porcentaje significativo de los costes de arranque estará en los costes de seccionamiento y, por tanto, calcular los costes de arranque será más complejo.

Veamos lo significativo que son estos costes añadidos. Considerar un simple bucle:

```
      do 10 i  =  1,n
10       A(i)  =  B(i)
```

El compilador generará dos bucles anidados para este código, exactamente como hace nuestro ejemplo. El bucle interior contiene una secuencia de dos operaciones vectoriales, LV (cargar vector) seguido por SV (almacenar vector). Cada iteración del bucle de la operación original del vector requerirá dos ciclos de reloj si no hubiera penalizaciones de arranque de ningún tipo. Las penalizaciones de arranque son de dos tipos: costes de arranque del vector y costes de seccionamiento. Para DLXV, el coste de arranque del vector es de 12 ciclos de reloj, para la carga del vector, más un retardo de 4 ciclos de reloj, porque el almacenamiento depende de la carga, dando un total de 16 ciclos de reloj. Podemos ignorar la latencia de almacenamiento, ya que nada depende de ello. La Figura 7.9 muestra el impacto sólo del coste de arranque

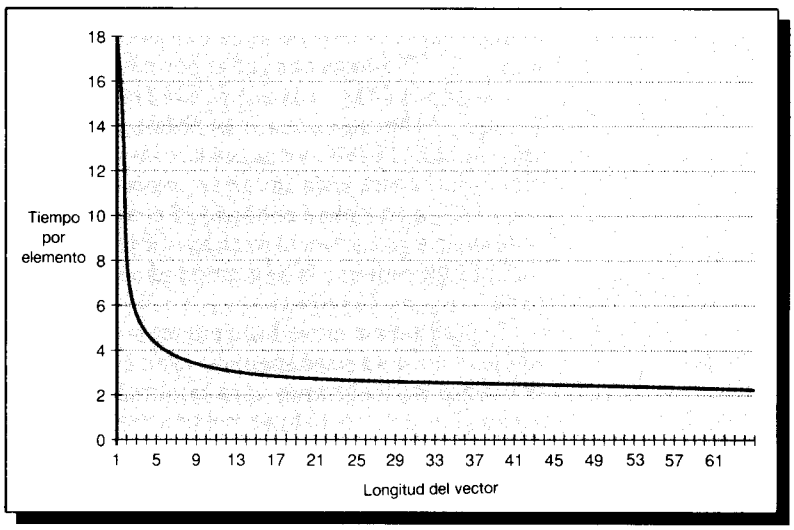


FIGURA 7.9 Impacto del coste de arranque del vector en un bucle consistente en una asignación vectorial. Para vectores cortos, el impacto del coste de arranque de 16 ciclos es enorme, decreciendo el rendimiento hasta nueve veces. No se ha incluido el gasto de seccionamiento.

vectorial cuando el vector crece de longitud 1 a longitud 64. Este coste de arranque puede disminuir la velocidad *(throughput)* en un factor de 9, dependiendo de la longitud del vector.

En la Sección 7.4, veremos un modelo de rendimiento unificado que incorpora todos los costes de arranque y adicionales *(overhead)*. Primero, examinemos cómo implementar vectores con accesos no secuenciales a memoria.

Separación entre elementos de un vector

El segundo problema que se trata en esta sección es cuando la posición en memoria de los elementos adyacentes de un vector no es secuencial. Considerar el código para multiplicar matrices:

```
      do 10 i = 1,100
         do 10 j = 1,100
            A(i,j) = 0.0
            do 10 k = 1,100
10             A(i,j) = A(i,j)+B(i,k)·C(k,j)
```

En la sentencia cuya etiqueta es 10, podríamos vectorizar la multiplicación de cada fila de B con cada columna de C y seccionar el bucle interior usando k como variable índice. Para hacer eso, debemos considerar cómo se direccionan los elementos adyacentes en B y en C. Cuando un array se ubica en memoria, se linealiza y debe organizarse por filas o columnas *(row-major or column major)*. El almacenamiento por filas, utilizado por muchos lenguajes excepto por FORTRAN, consiste en asignar posiciones consecutivas a elementos consecutivos de cada fila, haciendo adyacentes a los elementos B(i,j) y B(i,j+1). El almacenamiento por columnas, utilizado por FORTRAN, hace adyacentes a B(i,j) y B(i+1,j). La Figura 7.10 ilustra estas dos alternativas. Veamos los accesos a B y C en el bucle interior de la multiplicación de matrices. En FORTRAN, los accesos a los elementos de B serán no adyacentes en memoria, y para cada iteración se accederá a un elemento que esté separado por una fila completa del array. En este caso, los elementos de B que son accedidos por iteraciones en el lazo interior están separados por el tamaño de fila multiplicado por 8 (el número de bytes por elemento) para un total de 800 bytes.

Esta distancia entre los elementos consecutivos que van a formar un vector se denomina separación (*stride*). En el ejemplo actual, usando el almacenamiento por columnas para las matrices significa que la matriz C tiene una separación de 1, ó 1 doble palabra (8 bytes), separando a los elementos consecutivos, y la matriz B tiene una separación de 100, ó 100 dobles palabras (800 bytes).

Una vez que un vector se carga en un registro vectorial, actúa como si tuviera los elementos lógicamente adyacentes. Esto posibilita que una máquina con registros vectoriales maneje separaciones entre elementos mayores que uno, denominadas *separaciones no unitarias*, haciendo las operaciones de almacenamiento y carga de los vectores más generales. Por ejemplo, si pudié-

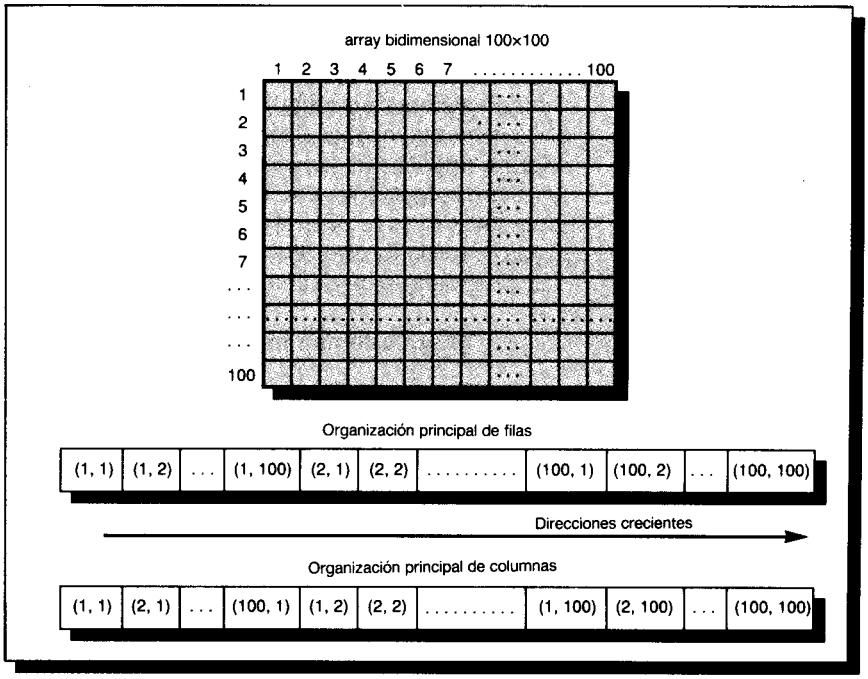


FIGURA 7.10 Matriz para un array bidimensional y organizaciones correspondientes en un almacenamiento de una dimensión. Si se almacena por filas, los elementos consecutivos en las filas son adyacentes en memoria, mientras que si se almacena por columnas, los elementos consecutivos en las columnas son adyacentes. Es fácil imaginar extender esto a arrays con más dimensiones.

ramos cargar una fila de B en un registro vectorial, entonces podríamos tratar la fila como lógicamente adyacente.

Por ello, es deseable que las operaciones de carga y almacenamiento de vectores especifiquen una separación entre elementos además de una dirección de comienzo. En la DLXV, donde la unidad direccionable es el byte, la separación entre elementos de nuestro ejemplo sería 800. El valor se debe calcular dinámicamente, ya que el tamaño de la matriz puede no conocerse en tiempo de compilación, o —como la longitud del vector— puede cambiar para diferentes ejecuciones de la misma sentencia. La separación del vector, como su dirección de comienzo puede ponerse en un registro de propósito general, donde se utiliza durante la duración de la operación vectorial. Entonces, la instrucción LVWS de DLXV (Cargar Vector con Separación) puede utilizarse para buscar el vector en un registro vectorial. De igual forma, cuando se está almacenando un vector de separación no unidad se puede utilizar SVWS (Almacenar Vector con Separación). En algunas máquinas vectoriales las cargas y almacenamientos siempre tienen un valor de separación almacenado en un registro, para que sólo haya una única instrucción.

Se pueden presentar complicaciones en la unidad de memoria al soportar separaciones entre elementos mayores que uno. Antes, veíamos que una ope-

ración memoria-registro vectorial podía proceder a velocidad completa si el número de bancos de memoria era como mínimo tan grande como el tiempo de acceso a memoria en ciclos de reloj. Sin embargo, una vez que se introducen separaciones no unitarias es posible pedir accesos al mismo banco a una frecuencia mayor que el tiempo de acceso a memoria. Esta situación se denomina *conflicto del banco de memoria* y hace que cada carga necesite un mayor tiempo de acceso a memoria. Un conflicto del banco de memoria se presenta cuando se le pide al mismo banco que realice un acceso antes que se haya completado otro. Por tanto, un conflicto del banco, y por consiguiente una detención, se presentará si:

$$\frac{\text{Mínimo común múltiplo (separación, número de bancos)}}{\text{Separación}} <$$

$$< \text{Latencia de acceso de memoria}$$

Ejemplo

Supongamos que tenemos 16 bancos de memoria con un tiempo de acceso de 12 ciclos de reloj. ¿Cuánto se tardará en completar la lectura de un vector de 64 elementos con una separación de 1? ¿y con una separación de 32?

Respuesta

Como el número de bancos es mayor que la latencia de cada módulo, para una separación de 1, la lectura empleará 12 + 64 = 76 ciclos de reloj, o 1,2 ciclos de reloj por elemento. La peor separación posible es un valor que sea múltiplo del número de bancos de memoria, como en este caso con una separación de 32 y 16 bancos de memoria. Cada acceso a memoria colisionará con el anterior. Esto nos lleva a un tiempo de acceso de 12 ciclos de reloj por elemento y un tiempo total para la carga del vector de 768 ciclos de reloj.

Los conflictos en los bancos de memoria no se presentarán si la separación entre elementos y número de bancos son relativamente primos entre sí y hay suficientes bancos para evitar conflictos en el caso de separación unidad. Aumentar el número de bancos de memoria, a un número mayor del mínimo, para prevenir detenciones con una separación de longitud 1, disminuirá la frecuencia de detenciones para las demás separaciones. Por ejemplo, con 64 bancos, una separación de 32 parará cada dos accesos, en lugar de en cada acceso. Si originalmente tuviésemos una separación de 8 y 16 bancos, pararía cada dos accesos; mientras que con 64 bancos, una separación de 8 parará en cada ocho accesos. Si tenemos acceso a varios vectores simultáneamente, también necesitaremos más bancos para prevenir conflictos. En los años noventa, la mayoría de los supercomputadores vectoriales tienen como mínimo 64 bancos, y algunos tienen 512.

7.4 Un modelo sencillo para el rendimiento vectorial

En esta sección se presenta un modelo para comprender el rendimiento de un bucle vectorizado. Hay tres componentes clave del tiempo de ejecución de un

bucle seccionado cuyo cuerpo es una secuencia de instrucciones vectoriales:

1. El tiempo de cada operación vectorial en el bucle para procesar un elemento, ignorando los costes de arranque, que llamamos $T_{elemento}$. La secuencia vectorial tiene, con frecuencia, un solo resultado, en cuyo caso $T_{elemento}$ es el tiempo en producir un elemento de ese resultado. Si la secuencia vectorial produce múltiples resultados, $T_{elemento}$ es el tiempo en producir un elemento de cada resultado. Este tiempo depende solamente de la ejecución de las instrucciones del vector. En breve veremos un ejemplo.

2. El coste adicional *(overhead)* de las instrucciones vectoriales para cada bloque seccionado. Este coste está formado por el coste de ejecución del código escalar para seccionamiento de cada bloque, T_{bucle}, más el coste de arranque del vector para cada bloque, $T_{arranque}$.

3. Los costes adicionales *(overhead)* del cálculo de las direcciones de comienzo y la escritura del vector de control. Esto se presenta una vez para la operación completa del vector. Este tiempo, T_{base}, consta únicamente de instrucciones escalares.

Estos componentes se pueden utilizar para determinar el tiempo total de ejecución para una secuencia vectorial operando sobre un vector de longitud n, que llamaremos T_n:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot (T_{bucle} + T_{arranque}) + n \cdot T_{elemento}$$

Los valores de $T_{arranque}$ y T_{bucle} son dependientes de la máquina y del compilador, mientras que el valor de $T_{elemento}$ depende principalmente del hardware. La secuencia vectorial concreta afecta a los tres valores; el efecto sobre $T_{elemento}$ es probablemente el más pronunciado, con $T_{arranque}$ y T_{bucle} menos afectados.

Por simplicidad, utilizaremos valores constantes para T_{base} y T_{bucle} en DLXV. En base a una serie de medidas de ejecución vectorial en la CRAY-1, los valores escogidos son 10 para T_{base} y 15 para T_{bucle}. En un primer vistazo, se puede pensar que estos valores, especialmente T_{bucle}, son muy pequeños. El coste de cada bucle requiere: inicializar las separaciones y las direcciones de comienzo del vector, incrementar los contadores, y ejecutar un salto del bucle. Sin embargo, estas instrucciones escalares pueden estar solapadas con las instrucciones vectoriales, minimizando el tiempo empleado en estas funciones de coste adicional. Los valores de T_{base} y T_{bucle}, por supuesto, dependen de la estructura del bucle, pero la dependencia es pequeña comparada con la conexión entre el código vectorial y los valores de $T_{elemento}$ y $T_{arranque}$.

Ejemplo

¿Cuál es el tiempo de ejecución para la operación vectorial $A = B \cdot s$, donde s es un escalar y la longitud de los vectores A y B es 200?

Respuesta

Aquí está el código seccionado de DLXV, suponiendo que las direcciones de A y B están inicialmente en `Ra` y `Rb` y `s` está en `Fs`:

```
        ADDI    R2,R0,# 1600 ;no. bytes en vector
        ADD     R2,R2,Ra     ;fin de vector A
        ADDI    R1,R0,#8     ;longitud de seccionamiento
        MOVI2S  VLR,R1       ;longitud de vector
        ADDI    R1,R0,#64    ;longitud en bytes
        ADDI    R3,R0,#64    ;longitud de vector
                             ;de otras piezas
loop:   LV      V1,Rb        ;carga B
        MULTVS  V2,V1,Fs     ;vector · escalar
        SV      Ra,V2        ;almacena A
        ADD     Ra,Ra,R1     ;siguiente sección de A
        ADD     Rb,Rb,R1     ;siguiente sección de B
        ADDI    R1,R0,#512   ;longitud total de vector
                             ; (en bytes)
        MOVI2S  VLR,R3       ;inicializa longitud a 64
        SUB     R4,R2,Ra     ;en el fin de A?
        BNZ     R4,LOOP      ;si no, volver atrás
```

A partir de este código, podemos ver que $T_{elemento} = 3$, debido a la carga, multiplicación y almacenamiento de cada elemento del vector. Además, nuestras suposiciones para DLXV son $T_{bucle} = 15$ y $T_{base} = 10$. Usemos nuestra fórmula básica:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot (T_{bucle} + T_{arranque}) + n \cdot T_{elemento}$$

$$T_{200} = 10 + (4) \cdot (15 + T_{arranque}) + 200 \cdot 3$$

$$T_{200} = 10 + 4 \cdot (15 + T_{arranque}) + 600 = 670 + 4 \cdot T_{arranque}$$

El valor de $T_{arranque}$ es la suma de:

- El arranque de la carga del vector de 12 ciclos de reloj,
- La detención de 4 ciclos de reloj, debido a la dependencia entre la carga y la multiplicación,
- Un arranque de 7 ciclos de reloj para la multiplicación, más
- Una parada de 4 ciclos de reloj debido a la dependencia entre la multiplicación y el almacenamiento.

Entonces, el valor de $T_{arranque}$ está dado por:

$$T_{arranque} = 12 + 4 + 7 + 4 = 27$$

Así, el valor total es

$$T_{200} = 670 + 4 \cdot 27 = 778$$

El tiempo de ejecución por elemento con todos los costes de arranque es entonces $\frac{778}{200} = 3{,}9$, comparado con un caso ideal de 3.

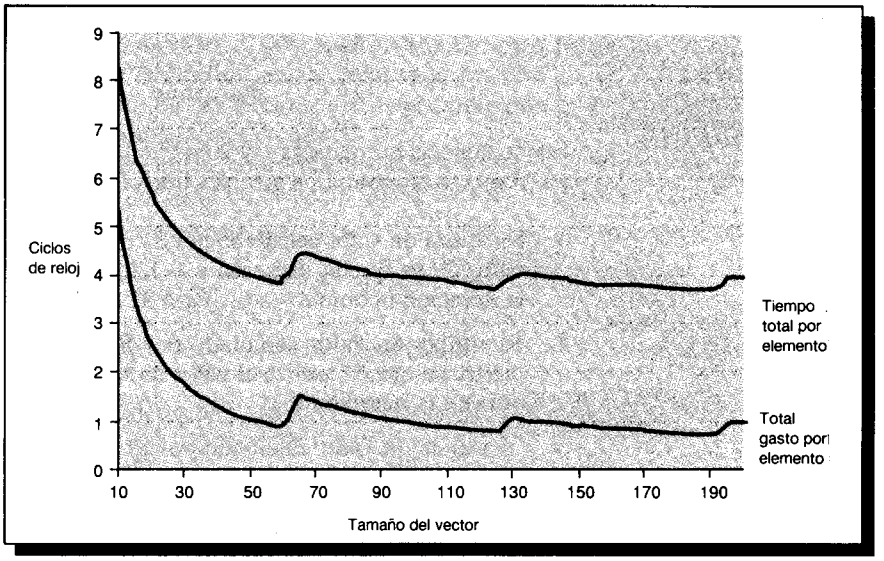


FIGURA 7.11 Tiempo total de ejecución por elemento y tiempo total de gasto por elemento, frente a la longitud vectorial para el ejemplo de la página anterior. Para vectores cortos el tiempo total de arranque es más de la mitad del tiempo total, mientras que para vectores grandes se reduce aproximadamente una tercera parte del tiempo total. Las bifurcaciones repentinas se presentan cuando la longitud del vector cruza un múltiplo de 64, forzando a otra iteración del código seccionado y ejecución de un conjunto de instrucciones vectoriales. Estas operaciones incrementan T_n en $T_{bucle} + T_{arranque}$.

La Figura 7.11 muestra el coste adicional y velocidades efectivas por elemento para el ejemplo anterior ($A = B \cdot s$) con varias longitudes vectoriales. Comparado con el modelo más simple de arranque, ilustrado en la Figura 7.9, vemos que la contabilidad de costes para todas las fuentes es mayor. En este ejemplo, el coste de arranque del vector, que está dibujado en la Figura 7.9 contabiliza sólo aproximadamente la mitad del coste adicional total por elemento.

7.5 Tecnología de compiladores para máquinas vectoriales

Para poder utilizar con efectividad una máquina vectorial el compilador debe poder reconocer que un bucle (o parte de un bucle) es vectorizable y generar el código vectorial apropiado. Esto implica determinar las dependencias que existen entre los operandos en el bucle. Por ahora, consideraremos sólo las dependencias que se presentan cuando un operando se escribe en un punto y se lee en un punto posterior. Estas corresponden a riesgos RAW (lectura después de escritura). Considerar un bucle como:

```
        do 10 i=1,100
1          A(i+1) = A(i) + B(i)
2          B(i+1) = B(i) + A(i+1)
10      continue
```

Las sentencias numeradas `1` y `2` en el cuerpo del bucle las denominamos S1 y S2, respectivamente. Los posibles tipos de dependencias diferentes son:

1. S1 utiliza un valor calculado por S1 en una iteración anterior. Esto es cierto para S1, ya que la iteración i + 1 utiliza el valor `A(i)` que se calculó en la iteración i como `A(i+1)`. Para S2 ocurre lo mismo con `B(i)` y `B(i+1)`.
2. S1 utiliza un valor calculado por S2 en una iteración anterior. Esto es cierto, ya que S1 utiliza el valor de `B(i+1)` en la iteración i+1 que S2 calcula en la iteración i.
3. S2 utiliza un valor calculado por S1 en la misma iteración. Esto es cierto para el valor de `A(i+1)`.

Como las operaciones vectoriales están segmentadas y la latencia puede ser bastante grande, una iteración anterior puede no completarse antes de que comience una iteración posterior: entonces, los valores que debe escribir la iteración anterior pueden no estar escritos antes que comience la iteración posterior. Consiguientemente, si existe la situación 1 ó 2, la vectorización del bucle introducirá un riesgo RAW —un riesgo que una máquina vectorial no comprueba. Esto significa que si existe alguna de las tres dependencias en las situaciones 1 y 2, el bucle no es vectorizable, y el compilador no generará instrucciones vectoriales para este código. En la situación 3, el hardware normal de detección de riesgos podría manejar la situación. Por tanto, un bucle que contenga sólo dependencias como las de la situación 3 puede vectorizarse, como veremos pronto. Las dependencias de las dos primeras situaciones, que involucran el uso de valores calculados en iteraciones anteriores del bucle, se denominan *dependencias entre iteraciones del bucle* (*loop-carried dependences*).

La primera tarea del compilador es determinar si en el cuerpo del bucle hay dependencias entre iteraciones del bucle. El compilador hace esto utilizando un algoritmo de análisis de dependencias. Como las sentencias del cuerpo del bucle involucran arrays, el análisis de dependencias es complejo. (Si no hubiese arrays, no habría nada que vectorizar.) El caso más simple se presenta cuando aparece un nombre de array sólo en una parte de la sentencia de asignación. Tomar, por ejemplo, esta variación del bucle anterior:

```
        do 10 i=1,100
           A(i) = B(i) + C(i)
           D(i) = A(i) · E(i)
10      continue
```

Si los arrays `A`, `B`, `C`, `D` y `E` son diferentes, entonces no pueden existir dependencias entre iteraciones del bucle. Hay una dependencia entre las dos sentencias para el vector `A`. Si el compilador cayó en la cuenta que había dos accesos a `A`, podría tratar de no releer `A` en la segunda sentencia haciendo en su lugar

la multiplicación vectorial, utilizando el registro de resultados de la suma vectorial. En este caso, el procesador vería el riesgo potencial RAW y pararía la emisión de la multiplicación de los vectores. Si el compilador almacenó A y lo releyó, entonces las lecturas y almacenamientos se presentarán en orden, lográndose una ejecución correcta.

Con frecuencia, en un bucle aparece el mismo nombre como fuente y destino, como ocurría en el bucle SAXPY. Allí, Y aparece a ambos lados de la asignación:

```
      do 10 i=1,100
         Y(i) = a·X(i) + Y(i)
10    continue
```

En este caso, no hay todavía dependencias arrastradas por el bucle, porque la asignación a Y no depende del valor de Y calculado en una iteración anterior. Sin embargo, el bucle siguiente, que se denomina *recurrencia*, contiene una dependencia entre iteraciones del bucle:

```
      do 10 i=2,100
         Y(i) = Y(i-1) + Y(i)
10    continue
```

La dependencia puede verse desenrollando el bucle: en la iteración j se utiliza el valor de Y(j–1), pero ese elemento se calcula en la iteración j–1, creando una dependencia entre iteraciones.

En general, ¿cómo detecta el compilador las dependencias? Suponer que hemos escrito un elemento del array con un valor índice $a \cdot \mathrm{i} + b$ y es accedido con el valor índice $c \cdot \mathrm{i} + d$, donde i es la variable índice del bucle que varía de m a n. Existe una dependencia si se mantienen dos condiciones:

1. Hay dos índices de iteración, j y k, ambos dentro de los límites del bucle.
2. El bucle almacena un elemento del array indexado por $a \cdot \mathrm{j} + b$ y más tarde busca ese **mismo** elemento del array cuando este es indexado por $c \cdot \mathrm{k} + d$. Esto es, $a \cdot \mathrm{j} + b = c \cdot \mathrm{k} + d$.

En general, a veces no se puede determinar si existe dependencia en tiempo de compilación. Por ejemplo, los valores de a, b, c y d pueden no ser conocidos, haciendo imposible decir si existe alguna dependencia. En otros casos, la comprobación de dependencias puede ser muy cara pero decidible en tiempo de compilación. Por ejemplo, los accesos pueden depender de los índices de iteración de múltiples bucles anidados. Muchos programas no contienen estas estructuras complejas, pero en su lugar contienen simples índices donde a, b, c y d son constantes. Para estos casos, es posible imaginar tests razonables de dependencia.

Un test sencillo y suficiente utilizado para detectar dependencias es el *máximo común divisor*, o GCD. Está basado en la observación de que si existe una dependencia entre iteraciones del bucle, entonces GCD (c,a) debe **dividir** a (d-b). (Recordar que un entero x, *divide* a otro entero y, si no hay resto

cuando hacemos la división $\frac{y}{x}$ y obtenemos un resultado entero.) La prueba GCD es suficiente para garantizar que no existe dependencia (ver Ejercicio 7.10); sin embargo, hay casos donde la prueba GCD tiene éxito, pero no existe dependencia. Por ejemplo, esto puede surgir porque el test GCD no tiene en cuenta los límites o extremos del bucle. Un test más complejo es el de Banerjee, llamado después de U. Barnerjee [1979], que tiene en cuenta los límites o extremos del bucle, pero todavía no es exacto. Siempre puede hacerse un test exacto resolviendo ecuaciones con valores enteros, pero esto puede ser caro para estructuras complejas de bucles.

Ejemplo

Usar la prueba GCD (máximo común divisor) para determinar si existen dependencias en el siguiente bucle:

```
      do 10 i=1,100
10       X(2·i+3) = X(2·i) · 5.0
```

Respuesta

Dados los valores $a=2$, $b=3$, $c=2$ y $d=0$, entonces GCD$(a,c) = 2$, y $d-b = -3$. Como 2 no divide -3, no hay dependencia posible.

Una *dependencia verdadera de datos* surge de un riesgo RAW y prevendrá la vectorización del bucle como una simple secuencia vectorial. Hay casos donde el bucle se puede vectorizar como dos secuencias vectoriales separadas (ver Ejercicio 7.11). Hay también dependencias correspondientes a riesgos WAR (escritura después de lectura), denominadas *antidependencias*, y a riesgos WAW (escritura después de escritura), denominadas *dependencias de salida.* Antidependencias y dependencias de salida no son verdaderas dependencias de datos. Son conflictos de nombres y se pueden eliminar renombrando los registros en el compilador con un método similar al que usa el algoritmo de Tomasulo cuando renombra los registros en tiempo de ejecución (ver Sección 6.7 del Capítulo 6). Los compiladores que vectorizan utilizan con frecuencia renombramiento en tiempo de compilación para eliminar antidependencias y dependencias de salida.

Ejemplo

El siguiente bucle tiene una antidependencia (WAR) y una dependencia de salida (WAW). Determinar todas las dependencias verdaderas, dependencias de salida y antidependencias, y eliminar las dependencias de salida y antidependencias renombrándolas.

```
      do 10 i=1,100
1        Y(i) = X(i) / s
2        X(i) = X(i) + s
3        Z(i) = Y(i) + s
4        Y(i) = s - Y(i)
10    continue
```

Respuesta Hay dependencias verdaderas entre la sentencia 1 y la sentencia 3, y entre la sentencia 1 y la sentencia 4 a causa de `Y(i)`. Estas no son arrastradas por el bucle, así que no evitarán la vectorización. Sin embargo, las dependencias forzarán a que las sentencias 3 y 4 esperen a que se complete la sentencia 1, aun cuando las sentencias 3 y 4 utilicen unidades funcionales diferentes a las de la sentencia 1. En la siguiente sección veremos una técnica para eliminar esta serialización.

Hay una antidependencia entre la sentencia 1 y la sentencia 2, y una dependencia de salida entre la sentencia 1 y la 4. La siguiente versión del bucle elimina estas falsas (o pseudo) dependencias.

```
      do 10 i=1,100
C     Y renombrado a T para eliminar dependencia de salida
1        T(i) = X(i) / s
C     X renombrado a X1 para eliminar antidependencia
2        X1(i) = X(i) + s
3        Z(i) = T(i) + s
4        Y(i) = s - T(i)
10     continue
```

Después del bucle, la variable `X` ha sido renombrada como `X1`. En el código que sigue al bucle, el compilador puede sencillamente sustituir el nombre `X` por `X1`. Para renombrar no se requiere una operación real de copia; puede hacerse por sustitución de nombres o por asignación de registros.

Además de decidir qué bucles son vectorizables, el compilador debe generar código de seccionamiento y asignar los registros vectoriales. Muchas transformaciones de vectorización se hacen a nivel fuente, aunque algunas optimizaciones involucren coordinar transformaciones fuente de alto nivel con transformaciones dependientes de la máquina de nivel más bajo. La asignación eficiente de los registros vectoriales es una optimización y es quizá la optimización más difícil —la que muchos compiladores que vectorizan no intentan.

Eficacia de las técnicas de vectorización

Dos factores afectan al éxito con que se puede ejecutar un programa en modo vectorial. El primer factor es la estructura del mismo programa: ¿tienen los bucles dependencias verdaderas de datos, o se pueden reestructurar para que no tengan dichas dependencias? Este factor está influenciado por los algoritmos escogidos y, en alguna extensión, por la forma que están codificados. El segundo factor es la capacidad del compilador. Aunque el compilador no pueda vectorizar un bucle donde no exista paralelismo entre las iteraciones del bucle, hay una tremenda variación en las posibilidades de los compiladores para determinar si se puede vectorizar un bucle.

Como indicación del nivel de vectorización que se puede conseguir en programas científicos, examinaremos los niveles de vectorización observados

Nombre de benchmark	Operaciones FP	Operaciones FP ejecutadas en modo vectorial
ADM	23 %	68 %
DYFESM	26 %	95 %
FLO52	41 %	100 %
MDG	28 %	27 %
MG3D	31 %	86 %
OCEAN	28 %	58 %
QCD	14 %	1 %
SPICE	16 %	7 %
TRACK	9 %	23 %
TRFD	22 %	10 %

FIGURA 7.12 Nivel de vectorización entre los benchmarks de Perfect Club cuando se ejecutan en el CRAY X-MP. La primera columna contiene el porcentaje de operaciones de punto flotante, mientras que la segunda contiene el porcentaje de operaciones FP ejecutadas en instrucciones vectoriales. Observar que esta ejecución de Spice con diferentes entradas muestra una mayor relación de vectorización.

en los benchmarks de «Perfect Club», explicados en la Sección 2.7 del Capítulo 2. Estos benchmarks son grandes aplicaciones científicas reales. La Figura 7.12 muestra el porcentaje de operaciones en punto flotante en cada benchmark y el porcentaje ejecutado en modo vectorial en la CRAY X-MP. La amplia variación en el nivel de vectorización se ha observado en varios estudios de rendimiento de aplicaciones en máquinas vectoriales. Mientras los mejores compiladores pueden mejorar el nivel de vectorización de algunos de estos programas, la mayoría necesitará reescribirse para conseguir incrementos significativos de vectorización. Por ejemplo, veamos con detalle nuestra versión del benchmark Spice. En Spice, con la entrada escogida encontramos que sólo el 3,7 por 100 de las operaciones en punto flotante se ejecutan en modo vectorial en la CRAY X-MP, y la versión vectorial corre solamente el 0,5 por 100 más rápido que la versión escalar. Claramente será necesario un nuevo programa o una reescritura significativa para obtener sobre Spice los beneficios de una máquina vectorial.

También hay una tremenda variación en la forma en que los compiladores vectorizan los programas. Como resumen del estado de los compiladores que vectorizan, considerar los datos de la Figura 7.13, que muestra la extensión de la vectorización para diferentes máquinas utilizando un grupo de pruebas de 100 «kernels» de FORTRAN escritos a mano. Los «kernels» fueron diseñados para probar la capacidad de vectorización y todos se podían vectorizar a mano. En los ejercicios veremos algunos ejemplos de estos bucles.

Máquina	Compilador	Completa-mente vectorizados	Parcial-mente vectorizados	No vectorizados
Ardent Titan-1	FORTRAN V1.0	62	6	32
CDC CYBER-205	VAST-2 V2.21	62	5	33
Series Convex C	FC5.0	69	5	26
CRAY X-MP	CFT77 V3.0	69	3	28
CRAY X-MP	CFT V1.15	50	1	49
CRAY-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN 77/ SX V.040	66	5	29
Stellar GS 1000	F77 preliberación	48	11	41

FIGURA 7.13 Resultado de usar compiladores que vectorizan a 100 núcleos (kernels) FORTRAN de test. Para cada máquina indicamos el número de bucles completamente vectorizados, parcialmente vectorizados, y no vectorizados. Estos bucles fueron colectados por Callahan, Dongarra y Levine [1988]. Las máquinas mostradas son las mencionadas en algún punto de este capítulo. Dos compiladores diferentes para el CRAY X-MP muestran la gran dependencia en tecnología de compiladores.

7.6 Mejorando el rendimiento vectorial

En esta sección se explican tres técnicas para mejorar el rendimiento de las máquinas vectoriales. La primera trata de hacer que una secuencia de operaciones vectoriales dependientes se ejecute con más rapidez. Las otras dos tratan con la ampliación de la clase de bucles que se pueden ejecutar en modo vectorial. La primera técnica, encadenamiento, se empezó a usar en el CRAY-1, pero ahora está soportada en muchas máquinas vectoriales. Las técnicas explicadas en la segunda y tercera parte de esta sección se han tomado de diversas máquinas y, en general, van más allá de las capacidades proporcionadas en las arquitecturas Cray-1 o Cray X-MP.

Encadenamiento-El concepto de adelantamiento extendido a los registros de vectores

Considerar la secuencia vectorial:

```
MULTV    V1,V2,V3
ADDV     V4,V1,V5
```

En el estado actual de DLXV estas dos instrucciones corren en un tiempo igual a

$$\begin{aligned} T_{\text{elemento}} \cdot \text{Longitud de vector} &+ \text{Tiempo de arranque}_{\text{ADDV}} + \text{tiempo de detención} + \text{Tiempo de arranque}_{\text{MULTV}} \\ &= 2 \cdot \text{Longitud de vector} + 6 + 4 + 7 \\ &= 2 \cdot \text{Longitud de vector} + 17 \end{aligned}$$

Debido a las dependencias, MULTV se debe completar antes que ADDV pueda comenzar. Sin embargo, si el registro del vector, V1 en este caso, se trata no como una única entidad, sino como un grupo de registros individuales, entonces el concepto segmentado del adelantamiento se puede extender para que funcione sobre los elementos individuales de un vector. Esta idea que permitirá a ADDV comenzar antes, en este ejemplo, se denomina *encadenamiento* (*chaining*). El encadenamiento permite que una operación vectorial comience tan pronto como los elementos individuales de su operando vectorial fuente estén disponibles: los resultados de la primera unidad funcional de la cadena se adelantan a la segunda unidad funcional. (¡Por supuesto, deben ser unidades diferentes para evitar utilizar dos veces la misma unidad por ciclo de reloj!) En una secuencia encadenada, la velocidad de iniciación es igual a un ciclo por reloj si las unidades funcionales en las operaciones encadenadas están todas totalmente segmentadas. Aun cuando las operaciones dependan entre sí, el encadenamiento permite que las operaciones procedan en paralelo sobre diferentes elementos del vector. Una velocidad sostenida (ignorando arranque) de dos operaciones en punto flotante por ciclo de reloj puede conseguirse, ¡aun cuando las operaciones sean dependientes!

El tiempo total de ejecución para la secuencia anterior se convierte en

$$\begin{aligned} \text{Longitud de vector} &+ \text{Tiempo de arranque}_{\text{ADDV}} + \\ &+ \text{Tiempo de arranque}_{\text{MULTV}} \end{aligned}$$

La Figura 7.14 muestra los tiempos de una versión encadenada, y de otra no encadenada del par anterior de instrucciones vectoriales con un vector de longitud 64. En la Figura 7.14, el tiempo total de la operación encadenada es de 77 ciclos de reloj. Con 128 operaciones en punto flotante realizadas en ese tiempo, se obtiene 1,7 FLOP por ciclo de reloj, frente a un tiempo total de 145 ciclos de reloj o 0,9 FLOP por ciclo de reloj para la versión no encadenada.

Veremos en la Sección 7.7 que el encadenamiento juega un papel importante en aumentar el rendimiento vectorial.

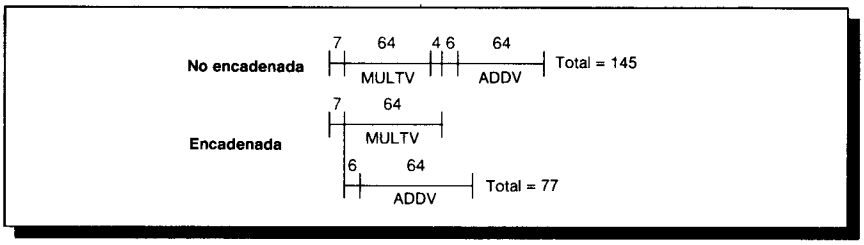


FIGURA 7.14 Temporización para una secuencia de operaciones vectoriales ADDV y MULTV dependientes, no encadenadas y encadenadas. El retardo de 4 ciclos de reloj proviene de una detención por la dependencia, descrita antes; los retardos de 6 y 7 ciclos de reloj son la latencia del sumador y multiplicador.

Sentencias ejecutadas condicionalmente y matrices dispersas

En la última sección, veíamos que muchos programas sólo conseguían moderados niveles de vectorización. Debido a la Ley de Amdahl, el aumento de velocidad de estos programas estaba muy limitado. Dos razones por las que no se consiguen altos niveles de vectorización son la presencia de condicionales (sentencias «if») dentro de los bucles y el uso de matrices dispersas. Los programas que contienen sentencias if en los bucles no se pueden ejecutar en modo vectorial utilizando las técnicas que hemos explicado porque las sentencias if introducen control del flujo en un bucle. De igual forma, las matrices dispersas no se pueden implementar eficientemente utilizando algunas de las capacidades que hemos visto; esto es un factor importante en la falta de vectorización de Spice. Esta sección explica técnicas que permiten que se ejecuten en modo vectorial programas con estas estructuras. Comencemos con una ejecución condicional.

Considerar el siguiente bucle:

```
      do 100 i = 1, 64
           if (A(i) .ne. 0) then
              A(i) = A(i) - B(i)
           endif
100   continue
```

Este bucle no puede vectorizarse normalmente a causa de la ejecución condicional del cuerpo. Sin embargo, si el bucle interior se pudiera ejecutar en las iteraciones para las cuales A(i) $\neq$ 0, entonces se podría vectorizar la sustracción.

El control de una máscara sobre un vector nos ayuda a hacer esto. El *control de máscara vectorial* necesita un vector Booleano de longitud MVL. Cuando se carga el *registro de máscara vectorial* con el resultado de un test del vector, cualquier instrucción vectorial que se vaya a ejecutar solamente opera sobre los elementos del vector cuyas entradas correspondientes en el re-

gistro de máscara vectorial sean 1. Las entradas del registro vectorial destino que corresponden a un 0 en el registro de máscara no se modifican por la operación del vector. Para que no actúe, el registro de máscara vectorial se inicializa todo a 1, haciendo que las instrucciones posteriores al vector operen con todos los elementos del vector. Ahora se puede utilizar el siguiente código para el bucle anterior, suponiendo que las direcciones de comienzo de A y B están en `Ra` y `Rb` respectivamente:

```
LV     V1,Ra       ;carga vector A en V1
LV     V2,Rb       ;carga vector B
LD     F0,#0       ;carga F0 con cero en punto flotante
SNESV  F0,V1       ;inicializa VM a 1 si V1(i)≠F0
SUBV   V1,V1,V2    ;resta bajo el control de la máscara
CVM                ;pone la máscara a todo unos
SV     Ra,V1       ;almacena el resultado en A
```

La mayoría de las máquinas vectoriales modernas tienen control de vectores, basado en el uso de máscaras. La capacidad de enmascarar descrita aquí está disponible en algunas máquinas, pero otras permiten el uso de máscaras vectoriales con sólo un número pequeño de instrucciones.

Sin embargo, utilizar un registro de máscara vectorial tiene desventajas. Primero, el tiempo de ejecución no decrece, aun cuando no operen algunos elementos del vector. Segundo, en algunas máquinas vectoriales la máscara sólo sirve para inhabilitar el almacenamiento del resultado en el registro destino, mientras todavía se realiza la operación real. Por ello, si la operación del ejemplo anterior fuese una división en lugar de una resta y el test fuese sobre B en vez de sobre A, podrían presentarse excepciones de punto flotante falsas, ya que la operación se estaba haciendo realmente. Las máquinas que enmascaran la operación y el almacenanamiento en memoria evitan este problema.

Ahora, examinemos las matrices dispersas; más tarde veremos otro método para tratar con la ejecución condicional. Hemos tratado con vectores cuyos elementos están separados por una distancia constante. En una aplicación con matrices dispersas, podríamos tener un código como el siguiente:

```
      do   100 i = 1,n
100        A(K(i)) = A(K(i)) + C(M(i))
```

Este código implementa una suma de vectores dispersos sobre los arrays A y C, utilizando los vectores índice K y M para designar los elementos distintos de cero de A y de C. (A y C deben tener el mismo número de elementos distintos de cero —`n` de ellos—.) Otra representación común para matrices dispersas utiliza un vector de bits para decir qué elementos existen, y a menudo ambas representaciones existen en el mismo programa. Las matrices dispersas se encuentran en muchos códigos, y hay muchas formas de implementarlas, dependiendo de las estructuras de datos utilizados en el programa.

El mecanismo principal para tratar con matrices dispersas son las operaciones de dispersar y agrupar usando vectores de índices. Una instrucción de

agrupar *(gather)* usa un *vector de índices*, y busca en memoria el vector cuyos elementos están en las direcciones obtenidas al sumar una dirección base a los desplazamientos dados en el vector de índices. El resultado es un vector no disperso en un registro vectorial. Después de que con estos elementos se opera en forma densa, el vector disperso se puede almacenar en forma expandida por un almacenamiento *disperso* (*scatter*), utilizando el mismo vector de índices. El soporte hardware para estas operaciones se denomina de *dispersar-agrupar* (*scatter-gather*) y se usó ya en el STAR-100 de CDC. Las instrucciones `LVI` (Cargar Vector Indexado) y `SVI` (Almacenar Vector Indexado) proporcionan estas operaciones en DLXV. Por ejemplo, suponer que `Ra`, `Rc`, `Rk` y `Rm` contienen las direcciones de comienzo de los vectores de la secuencia anterior. El bucle interior de la secuencia se puede codificar con instrucciones vectoriales como:

```
LV   Vk,Rk        ;carga K
LVI  Va,(Ra+Vk)   ;carga A(K(I))
LV   Vm,Rm        ;carga M
LVI  Vc,(Rc+Vm)   ;carga C(M(I))
ADDV Va,Va,Vc     ;los suma
SVI  (Ra+Vk),Va   ;almacena A(K(I))
```

Esta técnica permite que se ejecute en modo vectorial el códigos con matrices dispersas. El código fuente anterior **nunca** lo vectorizaría, automáticamente, un compilador porque el compilador no puede saber si los elementos de K tienen valores diferentes, y, por ello, que no existen dependencias. En su lugar, una directiva del programador podría indicar al compilador que ejecute el bucle en modo vectorial.

La capacidad de dispersar/agrupar se incluye en muchos de los supercomputadores más modernos. Estas operaciones raramente se ejecutan a un elemento por ciclo de reloj, pero son todavía mucho más rápidas que la alternativa, que puede ser un bucle escalar. Si cambian los elementos distintos de cero de una matriz se debe calcular un nuevo vector de índices. Muchas máquinas tienen soporte para calcular rápidamente el vector de índices. La instrucción `CVI` (Crear Vector Indice) en DLXV crea un vector de índices dada una separación (m), donde los valores del vector de índices son 0, m, $2 \cdot m$, ..., $63 \cdot m$. Algunas máquinas proporcionan una instrucción para crear un vector de índices comprimido cuyas entradas corresponden a las posiciones con un 1 en el registro de máscara. Otras arquitecturas vectoriales proporcionan un método para comprimir un vector. En DLXV, definimos la instrucción `CVI` para crear siempre un vector comprimido de índices utilizando el vector de máscara. Cuando el vector de máscara tenga todos los bits a 1, se creará un vector de índices estándar.

Las cargas/almacenamientos indexados y la instrucción CVI proporcionan un método alternativo para soportar la ejecución condicional. Aquí se da una secuencia vectorial que implementa el bucle que vimos en la página 407:

```
LV    V1,Ra       ;carga vector A en V1
LD    F0,#0       ;carga F0 con cero en punto flotante
```

```
SNESV   F0,V1       ;pone VM(i) a 1 si V1(i)≠F0
CVI     V2,#8       ;genera índices en V2
POP     R1,VM       ;calcula el número de 1 de VM
MOVI2S  VLR,R1      ;carga registro de longitud vectorial
CVM
LVI     V3,(Ra+V2)  ;carga los elementos de A
                     distintos de cero
LVI     V4,(Rb+V2)  ;carga los elementos
                    ; correspondientes de B
SUBV    V3,V3,V4    ;hace la resta
SVI     (Ra+V2),V3  ; almacena A
```

El que la implementación usando agrupar/dispersar sea mejor que la versión ejecutada de manera condicional, depende de la frecuencia con que se cumpla la condición y del coste de las operaciones. Ignorando encadenamiento, el tiempo de ejecución de la primera versión (en la página 407) es $5n + c_1$. El tiempo de ejecución de la segunda versión, utilizando cargas y almacenamientos indexados con un tiempo de ejecución de un elemento por ciclo de reloj es $4n + 4 \cdot f \cdot n + c_2$, donde f es la fracción de elementos para los cuales la condición es cierta (p. e., $A \neq 0$). Si suponemos que los valores de c_1 y c_2 son parecidos, o que son mucho menores que n, podemos determinar cuándo es mejor esta segunda técnica.

$$\text{Tiempo}_1 = 5n$$
$$\text{Tiempo}_2 = 4n + 4 \cdot f \cdot n$$

Queremos $\text{Tiempo}_1 \geq \text{Tiempo}_2$, así

$$5n \geq 4n + 4 \cdot f \cdot n$$
$$\frac{1}{4} \geq f$$

Es decir, el segundo método es más rápido si menos de la cuarta parte de los elementos son distintos de cero. En muchos casos, la frecuencia de ejecución es mucho menor. Si el vector de índices puede ser reusado, o si crece el número de sentencias vectoriales con la sentencia if, la ventaja de la aproximación de «dispersar/agrupar» aumentará claramente.

Reducción vectorial

Como vimos en la Sección 7.5, algunas estructuras de bucle no se vectorizan fácilmente. Una estructura común es una *reducción* —un bucle que reduce un array a un simple valor por aplicación repetida de una operación. Este es un caso especial de recurrencia. Un ejemplo común ocurre en el producto escalar de dos vectores *(dot product)*.

```
      dot = 0.0
      do 10 i=1,64
10          dot = dot + A(i) · B(i)
```

Este bucle tiene una clara dependencia entre iteraciones del bucle (en dot) y no puede ser vectorizado de una forma correcta. La primera cosa que haría un buen compilador de vectorización sería dividir el bucle para separar la parte vectorizable y la recurrencia y quizá reescribir el bucle como:

```
      do 10 i=1,64
10          dot(i) = A(i) · B(i)
      do 20 i=2,64
20          dot(1) = dot(1) + dot(i)
```

La variable dot se ha expandido en un vector; esta transformación se denomina *expansión escalar.*

Un esquema sencillo para compilar el bucle con la recurrencia es añadir secuencias de vectores progresivamente más cortos —dos vectores de 32 elementos, después dos vectores de 16 elementos, y así sucesivamente. Esta técnica se denomina *doblamiento recursivo.* Es más rápida que hacer todas las operaciones en modo escalar. Muchas máquinas vectoriales proporcionan hardware para hacer reducciones, como veremos a continuación.

Ejemplo

Mostrar cómo sería el código FORTRAN para la ejecución del segundo bucle en el fragmento de código anterior utilizando doblamiento recursivo.

Respuesta

Aquí está el código:

```
      len = 32
      do 100 j=1,6
            do 10 i=1,len
10                dot(i) = dot(i) + dot(i+len)
            len = len / 2
100   continue
```

Cuando se acaba el bucle, la suma está en dot(1).

En algunas máquinas vectoriales, los registros vectoriales son direccionables, y se pueden utilizar otras técnicas, a veces denominadas sumas parciales. Esto se explica en el Ejercicio 7.12. Hay una importante advertencia en el uso de técnicas vectoriales para reducción. Para hacer la reducción, estamos contando con la asociatividad del operador que se está utilizando para la reducción. Sin embargo, debido al redondeo y al rango finito, la aritmética de punto flotante no es estrictamente asociativa. Por esta razón, la mayoría de los compiladores necesitan que el programador indique si se puede utilizar la asociatividad para compilar más eficientemente las reduccciones.

7.7 Juntando todo: evaluación del rendimiento de los procesadores vectoriales

En esta sección examinamos diferentes medidas del rendimiento de máquinas vectoriales y lo que nos dicen respecto a la máquina. Para determinar el rendimiento de una máquina sobre un problema vectorial, debemos examinar el coste de arranque y la velocidad sostenida. La forma más sencilla y mejor de informar del rendimiento de una máquina vectorial sobre un bucle, es dar el tiempo de ejecución del bucle vectorial. Para bucles vectoriales a menudo se da la velocidad en MFLOPS (Millones de Operaciones en punto FLotante Por Segundo) en lugar del tiempo de ejecución. Usaremos la notación R_n para la velocidad en MFLOPS sobre un vector de longitud n. Utilizar las medidas T_n (tiempo) o R_n (velocidad) es equivalente si el número de FLOPS es el correcto (ver Capítulo 2, Sección 2.2, para una extensa explicación de los MFLOPS). En cualquier evento, cualquier otra medida debería incluir los costes adicionales.

En esta sección examinamos el rendimiento de DLXV sobre nuestro bucle SAXPY considerándolo desde diferentes puntos de vista. Continuaremos para calcular el tiempo de ejecución de un bucle vectorial utilizando la ecuación desarrollada en la Sección 7.4. Al mismo tiempo, examinaremos diferentes formas de medir el rendimiento utilizando el tiempo calculado. Los valores constantes de T_{bucle} y T_{base} utilizados en esta sección introducen algún pequeño error, que será ignorado.

Medidas de rendimiento vectorial

Como la longitud del vector es tan importante al establecer el rendimiento de una máquina, con frecuencia se aplican medidas relacionadas con la longitud, además del tiempo y los MFLOPS. Estas medidas relacionadas con la longitud tienden a variar de manera espectacular a través de diferentes máquinas y son interesantes de comparar. (Recordar, sin embargo, que el **tiempo** es siempre la medida de interés cuando se compara la velocidad relativa de dos máquinas.) Tres de las medidas más importantes relacionadas con la longitud son:

R_∞—La velocidad en MFLOPS sobre un vector de longitud infinita. Aunque esta medida puede ser de interés cuando se estiman rendimientos máximos, los problemas reales no tienen longitudes de vectores ilimitadas, y las penalizaciones de los costes adicionales encontradas en problemas reales serán mayores. (R_n es la velocidad en MFLOPS para un vector de longitud n.)

$N_{1/2}$—La longitud del vector necesaria para alcanzar la mitad de R_∞. Esto es una buena medida del impacto de los costes adicionales.

N_v—La longitud del vector necesaria para hacer el modo vectorial más rápido que el modo escalar. Esto mide el coste adicional y la velocidad de los escalares relativa a los vectores.

Examinemos estas medidas para nuestro problema SAXPY ejecutándose en DLXV. Cuando se encadenan operaciones, el bucle interior del código SAXPY es de la forma (suponiendo que Rx y Ry contienen direcciones iniciales):

```
LV      V1,Rx       ;carga el vector X
MULTSV  V2,S1,V1    ;vector · escalar-encadenado a LV X
LV      V3,Ry       ;carga el vector Y
ADDV    V4,V2,V3    ;suma aX + Y, encadenado a LV Y
SV      Ry,V4       ;almacena el vector Y
```

Recordar nuestra ecuación de rendimiento para el tiempo de ejecución de un bucle vectorial con n elementos, T_n:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot (T_{bucle} + T_{arranque}) + n \cdot T_{elemento}$$

Como hay tres referencias a memoria y sólo un camino con memoria, el valor de $T_{elemento}$ debe ser como mínimo 3, y el encadenamiento permite que sea exactamente 3. Si $T_{elemento}$ fuese una indicación completa del rendimiento, el bucle se ejecutaría a una velocidad en MFLOPS igual a $\frac{2}{3}$ · frecuencia de reloj (ya que hay 2 FLOPS por iteración). Por tanto, basado sólo en el tiempo de $T_{elemento}$, una DLXV a 80 MHz ejecutaría este bucle en 53 MFLOPS. Pero el benchmark Linpack, cuyo núcleo es este cálculo, corre sólo a 13-MFLOPS (sin ninguna optimización de un compilador sofisticado, como explicamos en los ejercicios) en un CRAY-1 de 80 MHz, ¡similar a DLXV! Veamos qué justifica la diferencia.

Rendimiento máximo de DLXV en SAXPY

Primero, deberemos determinar qué es realmente el rendimiento máximo R_∞, ya que sabemos que difiere de la velocidad ideal de 53 MFLOPS. La Figura 7.15 muestra los tiempos dentro de cada bloque de código seccionado.

De los datos de la Figura 7.15 y del valor de $T_{elemento}$ sabemos que

$$T_{arranque} = 241 - 64 \cdot T_{elemento} = 241 - 192 = 49$$

Este valor es igual a la suma de las latencias de las unidades funcionales:

$$12 + 7 + 12 + 6 + 12 = 49$$

Usando MVL = 64, $T_{bucle} = 15$, $T_{base} = 10$, y $T_{elemento} = 3$ en la ecuación del rendimiento, el tiempo para una operación de n elementos es

$$T_n = 10 + \left\lceil \frac{n}{64} \right\rceil \cdot (15 + 49) + 3n$$

$$T_n = 10 + n + 64 + 3n = 4n + 74$$

Operación		Comienza en número de reloj	Completa en número de reloj	Comentario
LV	V1,Rx	0	12 + 64 = 76	Latencia simple
MULTV	a,V1	12 + 1 = 13	13 + 7 + 64 = 84	Encadenada a LV
LV	V2,Ry	76 + 1 = 77	77 + 12 + 64 = 153	Comienzan después del primer LV hecho (contención de memoria)
ADDV	V3,V1,V2	77 + 1 + 12 = 90	90 + 6 + 64 = 160	Encadenada a MULTV y LV
SV	Ry,V3	160 + 1 + 4 = 165	165 + 12 + 64 = 241	Debe esperar en ADDV; no encadenado (contención de memoria)

FIGURA 7.15 El bucle SAXPY encadenado en DLXV. Hay tres tipos distintos de retardo: retardo de 4 *ciclos de reloj cuando se presenta una dependencia no encadenada, retardo de latencia que se presenta cuando* se espera un resultado de la unidad funcional (6 para la suma, 7 para la multiplicación y 12 para accesos a memoria), y retardo debido a la contención para el canal de acceso a memoria. La última causa es la que hace el tiempo por elemento de 3 relojes como mínimo.

La velocidad sostenida realmente es de unos 4 ciclos de reloj por iteración, en lugar de la velocidad teórica de 3 ciclos por iteración, que ignora costes adicionales. La parte principal de la diferencia es el coste del gasto para cada bloque de 64 elementos. El gasto básico de arranque, T_{base}, suma sólo $\frac{10}{n}$ al tiempo para cada elemento. Este coste desaparece con vectores grandes.

Ahora podemos calcular R_∞ para un reloj de 80 MHz como

$$R_\infty = \lim_{n\to\infty}\left(\frac{\text{Operaciones por iteración} \cdot \text{Frecuencia de reloj}}{\text{Ciclos de reloj por iteración}}\right)$$

El numerador es independiente de n, por consiguiente

$$R_\infty = \frac{\text{Operaciones por iteración} \cdot \text{Frecuencia de reloj}}{\lim\limits_{n\to\infty}(\text{Ciclos de reloj por iteración})}$$

$$\lim_{n\to\infty}(\text{Ciclos de reloj por iteración}) = \lim_{n\to\infty}\left(\frac{T_n}{n}\right) = \lim_{n\to\infty}\left(\frac{4n + 74}{n}\right) = 4$$

$$R_\infty = \frac{2 \cdot 80\ \text{MHz}}{4} = 40\ \text{MFLOPS}$$

Rendimiento sostenido de Linpack en DLXV

El benchmark Linpack es una eliminación Gaussiana sobre una matriz 100×100. Por tanto, las longitudes de los elementos vectoriales varían desde 99 hasta 1. Un vector de longitud k se utiliza k veces. Por tanto, la longitud vectorial media está dada por:

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66{,}3$$

Ahora podemos obtener una estimación precisa del rendimiento de SAXPY utilizando un vector de longitud 66.

$$T_{66} = 10 + 2 \cdot (15 + 49) + 66 \cdot 3 = 10 + 128 + 198 = 336$$

$$R_{66} = \frac{2 \cdot 66 \cdot 80}{336} \text{ MFLOPS} = 31{,}4 \text{ MFLOPS}$$

En realidad, Linpack no emplea todo su tiempo en el bucle interior. El rendimiento real del benchmark puede calcularse tomando la media armónica ponderada de las velocidades en MFLOPS dentro del bucle interior (31,4 MFLOPS) y fuera de ese bucle (aproximadamente 0,5 MFLOPS). Podemos calcular los factores de ponderación conociendo el porcentaje de tiempo dentro del bucle interior después de la vectorización.

El porcentaje del bucle interior después de la vectorización se puede obtener utilizando la Ley de Amdahl si conocemos el porcentaje escalar y el aumento de velocidad por la vectorización. En modo escalar, aproximadamente el 75 por 100 del tiempo de ejecución se emplea en el bucle interior, y el aumento por la vectorización es aproximadamente cinco veces. Con esta información el porcentaje de tiempo en el bucle interior después de la vectorización se puede calcular:

$$\text{Tiempo total relativo después de la vectorización} = \frac{0{,}75}{5} + 0{,}25$$

$$= 0{,}15 + 0{,}25 = 0{,}40$$

Porcentaje de tiempo en el bucle interior después de la vectorización

$$= \frac{0{,}15}{0{,}40} = 37{,}5\ \%$$

El restante 62,5 por 100 del tiempo se emplea fuera del bucle principal. Por tanto, la velocidad global en MFLOPS es

$$\text{Porcentaje}_{\text{interior}} \cdot \text{MFLOPS}_{\text{interior}} + \text{Porcentaje}_{\text{resto}} \cdot \text{MFLOPS}_{\text{restos}}$$

$$= 37{,}5\,\% \cdot 31{,}4 + 62{,}5\,\% \cdot 0{,}5 = 12{,}1 \text{ MFLOPS}$$

Esto es comparable a la velocidad a la cual el CRAY-1 corre este benchmark.

Ejemplo

¿Cuál es $N_{½}$ para el bucle interior de SAXPY y DLXV con un reloj de 80 MHz?

Respuesta

Utilizando R_∞ como velocidad máxima, queremos conocer la longitud del vector que consiga aproximadamente 20 MFLOPS. Así,

$$\frac{\text{Ciclos de reloj}}{\text{Iteración}} = \frac{\dfrac{\text{FLOPS}}{\text{Iteración}} \cdot \dfrac{\text{Relojes}}{\text{Segundo}}}{\dfrac{\text{FLOPS}}{\text{Segundo}}}$$

$$= \frac{2 \cdot 80 \text{ MHz}}{20 \text{ MFLOPS}} = 8$$

Por consiguiente, una velocidad de 20 MFLOPS significa que una iteración del bucle se completa cada 8 ciclos de reloj en promedio, o que $\frac{T_n}{n} = 8$. Utilizando nuestra ecuación y suponiendo que $n \leqslant 64$,

$$T_n = 10 + 1 \cdot 64 + 3 \cdot n$$

Sustituyendo T_n en la primera ecuación, obtenemos

$$8n = 74 + 3 \cdot n$$
$$5n = 74$$
$$n = 14{,}8$$

Así $N_{½} = 15$; es decir, un vector de longitud 15 da aproximadamente la mitad del rendimiento máximo para el bucle SAXPY sobre DLXV.

Ejemplo

¿Cuál es la longitud del vector N_v, tal que la operación vectorial se ejecute con más rapidez que la escalar?

Respuesta

De nuevo sabemos que $N_v < 64$. El tiempo para hacer una iteración en modo escalar puede estimarse como $10 + 12 + 12 + 7 + 6 = 47$ ciclos de reloj, donde 10 es la estimación de gastos del bucle, sabiendo que es algo menor que los gastos de seccionamiento del bucle. En el último problema demostrábamos que este bucle vectorial corre en modo vectorial en un tiempo $T_n = 74 + 3 \cdot n$ ciclos de reloj para un vector de longitud $\leqslant 64$. Por tanto,

$$74 + 3n = 47n$$

$$n = \frac{74}{44}$$

$$N_v = 2$$

Para el bucle SAXPY, el modo vectorial es más rápido que el escalar, mientras que el vector tenga como mínimo dos elementos. Este número es sorprendentemente pequeño, como veremos en la sección siguiente (Falacias y pifias).

Rendimiento de SAXPY en una DLXV mejorada

SAXPY, como muchos problemas vectoriales, está limitado por el acceso a memoria. Consecuentemente, el rendimiento podría mejorarse añadiendo más canales para acceder a memoria. Esta es la principal diferencia arquitectónica entre el CRAY X-MP y el CRAY-1. El CRAY X-MP tiene tres canales de acceso a memoria, comparado con el único del CRAY-1, y un encadenamiento más flexible. ¿Cómo afecta esto al rendimiento?

Ejemplo

¿Cuál sería el valor de T_{66} para SAXPY sobre DLXV si añadimos dos canales más a memoria?

Respuesta

La Figura 7.16 es una versión de la Figura 7.15, ajustada para múltiples canales de acceso a memoria.

Con tres canales con memoria, el rendimiento mejora enormemente. Aquí está nuestra ecuación estándar del rendimiento:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot (T_{bucle} + T_{arranque}) + n \cdot T_{elemento}$$

Con tres canales con memoria, el valor de $T_{elemento}$ se hace 1, así que

$$T_{arranque} = 104 - 64 \cdot T_{elemento} = 104 - 64 = 40$$

Operación	Comienza en número de reloj	Completa en número de reloj	Comentario
`LV V1,Rx`	0	12 + 64 = 76	Latencia sencilla
`MULTV a,V1`	12 + 1 = 13	13 + 7 + 64 = 84	Encadenado a `LV`
`LV V2,Ry`	2	2 + 12 + + 64 = 78	Comienza inmediatamente
`ADDV V3,V1,V2`	13 + 1 + 7 = 21	21 + 6 + 64 = 91	Encadenado a `MULTV` y `LV`
`SV Ry,V3`	21 + 1 + 6 = 28	28 + 12 + 64 = 104	Encadenado a `ADDV`

FIGURA 7.16 El bucle SAXPY encadenado en DLXV con tres canales de acceso con memoria. Los únicos retardos son los de latencia que se presentan cuando se espera un resultado desde una unidad funcional (6 para la suma, 7 para la multiplicación y 12 para cada acceso a memoria).

La reducción de detenciones reduce la penalización de arranque para cada secuencia. Los valores de T_{bucle} y T_{base}, 15 y 10, permanecen iguales. Además, para un vector de longitud media 66, tenemos:

$$T_{66} = T_{base} + \left\lceil \frac{66}{64} \right\rceil \cdot (T_{bucle} + T_{elemento}) + 66 \cdot T_{elemento}$$

$$T_{66} = 10 + 2 \cdot (15 + 40) + 66 \cdot 1 = 186$$

Con tres canales de acceso con memoria, hemos reducido el número de ciclos de reloj para rendimiento sostenido desde 336 a 186, un factor de 1,8. Notar el efecto de la Ley de Amdahl: mejorábamos la velocidad teórica máxima, que se medía por $T_{elemento}$, en un factor de 3, pero sólo conseguíamos una mejora global, en un factor de 1,8, del rendimiento sostenido. Debido a que la velocidad fuera del bucle interior es probablemente menor que 1,8, la mejora global en tiempo de ejecución para el benchmark también será menor.

Otra mejora podría provenir al permitir que comience una iteración del bucle antes que se complete otra. Esto requiere que se permita que una operación vectorial comienza a utilizar una unidad funcional, antes de que se haya completado otra operación. Esto complica sustancialmente la lógica que emite instrucciones, pero tiene la ventaja de que el coste adicional de arranque sólo se presenta una vez, independientemente de la longitud del vector. En un vector largo, el coste por bloque ($T_{bucle} + T_{arranque}$) se puede amortizar completamente. De esta forma, una máquina con registros vectoriales puede tener gastos de arranque bajos para vectores cortos y un alto rendimiento de pico para vectores muy largos.

Ejemplo

¿Cuáles serán los valores de R_∞ y T_{66} para SAXPY en DLXV si añadiésemos dos canales de acceso más a memoria y permitiésemos que se solapasen completamente los costes de arranque y de seccionamiento.

Respuesta

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operaciones por iteración} \cdot \text{Frecuencia de reloj}}{\text{Ciclos de reloj por iteración}} \right)$$

$$\lim_{n \to \infty} (\text{Ciclos de reloj por iteración}) = \lim_{n \to \infty} \left(\frac{T_n}{n} \right)$$

Como $T_n = n + 40 + 10 + 15 = n + 65$,

$$\lim_{n \to \infty} \left(\frac{T_n}{n} \right) = \lim_{n \to \infty} \left(\frac{n + 65}{n} \right) = 1$$

$$R_\infty = \frac{2 \cdot 80 \text{ MHz}}{1} = 160 \text{ MFLOPS}$$

Por ello, añadiendo canales de acceso extra con memoria y lógica de emisión de instrucciones más flexible se obtiene una mejora en el rendimiento máximo de un factor de 4. Sin embargo, $T_{66} = 131$, de forma que, para vectores más cortos, la mejora del rendimiento sostenido es aproximadamente el 40 por 100.

En resumen, hemos examinado diversas medidas de rendimiento vectorial. El rendimiento teórico máximo se puede calcular en base puramente del valor de $T_{elemento}$ como

$$\frac{\text{Número de FLOPS por iteración} \cdot \text{Frecuencia de reloj}}{T_{elemento}}$$

Al incluir el gasto del bucle, podemos calcular valores del rendimiento de pico (o máximo) para un vector de longitud infinita (R_∞), y también para el rendimiento sostenido R_n para un vector de longitud n, que se calcula como:

$$R_n = \frac{\text{Número de FLOPS por iteración} \cdot n \cdot \text{Frecuencia de reloj}}{T_n}$$

Utilizando estas medidas podemos encontrar también $N_{1/2}$ y N_v, que nos dan otra forma de examinar los costes de arranque para vectores y la relación entre la velocidad vectorial y velocidad escalar. Una amplia variedad de medidas sobre rendimiento de máquinas vectoriales son útiles para comprender el amplio rango de rendimientos que las aplicaciones pueden ver en una máquina vectorial.

7.8 Falacias y pifias

Pifia: Concentrarse en el rendimiento máximo e ignorar los gastos de arranque.

Las primeras máquinas vectoriales, como la ASC de TI y la STAR-100 de CDC, tenían tiempos de arranque grandes. Para algunos problemas vectoriales, ¡N_v podría ser mayor que 100! Hoy día, los supercomputadores japoneses, con frecuencia, tienen mayores velocidades sostenidas que las máquinas de Cray Research. Pero con gastos de arranque que son del 50 al 100 por 100 más altos, las velocidades sostenidas más rápidas no proporcionan, con frecuencia, una ventaja real. En la CYBER-205 el gasto de arranque para SAXPY es de 158 ciclos de reloj, incrementando sustancialmente el punto de cruce de los costes. Con una sola unidad vectorial, que contiene dos canales de acceso con memoria, el CYBER-205 puede sostener una velocidad de 2 ciclos de reloj por iteración. El tiempo de SAXPY para un vector de longitud n es, por tanto, aproximadamente $158 + 2n$. Si las frecuencias de reloj del CRAY-1 y del CYBER-205 fuesen idénticas, el CRAY-1 sería más rápida hasta $n > 64$. Como el reloj del CRAY-1 es también más rápido (aunque el 205 es más nuevo), el punto de cruce está sobre 100. Comparando un CYBER-205 de

cuatro canales de acceso vectoriales (la máquina de tamaño máximo) con el CRAY X-MP que apareció poco después que el 205, éste completa dos resultados por ciclo de reloj —dos veces más rápida que la X-MP. Sin embargo, los vectores deben ser mayores que 200 aproximadamente para que el CYBER 205 sea más rápido. El problema de los costes de arranque ha sido la principal dificultad para las arquitecturas vectoriales memoria-memoria.

Pifia: Incrementar el rendimiento vectorial, sin incrementos comparables en rendimiento escalar.

Esta es otra área donde Seymour Cray reescribió las reglas. Muchas de las primeras máquinas vectoriales tenían unidades escalares comparativamente lentas (así como grandes costes de arranque). Incluso hoy día, máquinas con mayores rendimientos vectoriales máximos, pueden ser superadas por una máquina con menor rendimiento vectorial, pero mejor rendimiento escalar. El buen rendimiento escalar mantiene bajos los costes (seccionamiento, por ejemplo) y reduce el impacto de la Ley de Amdahl. Un buen ejemplo proviene de comparar una máquina escalar rápida y una máquina vectorial con menor rendimiento escalar. Los «kernels» FORTRAN de Livermore son una colección de 24 «kernels» científicos con varios grados de vectorización (ver Capítulo 2; Sección 2.2). La Figura 7.17 muestra el rendimiento de dos máquinas diferentes sobre este benchmark. A pesar del mayor rendimiento máximo de la máquina vectorial, es el bajo rendimiento escalar el que la hace más lenta que una máquina escalar rápida. La falacia siguiente está muy relacionada con esta pifia.

*Falacia: El rendimiento escalar de los **mejores** supercomputadores es bajo.*

Los supercomputadores de Cray Research han tenido siempre buen rendimiento escalar. Medidas del CRAY Y-MP ejecutando el benchmark de Spice (el no vectorizable) muestran esto. Cuando el benchmark de Spice está corriendo en el CRAY Y-MP en modo escalar ejecuta 665 millones de instrucciones, con un CPI de 4,1. Por comparación, la DECstation 3100 ejecuta 738 millones de instrucciones con un CPI de 2,1. Aunque la DECstation utiliza

Máquina	Velocidad mínima para cualquier bucle	Velocidad máxima para cualquier bucle	Media armónica de los 24 bucles
MIPS M/120-5	0,80 MFLOPS	3,89 MFLOPS	1,85 MFLOPS
Stardent-1500	0,41 MFLOPS	10,08 MFLOPS	1,72 MFLOPS

FIGURA 7.17 Medidas de rendimiento para los kernels (núcleos) FORTRAN de Livermore en dos máquinas diferentes. MIPS M/120-5 y Stardent-1500 (antiguamente Ardent Titan-1) utilizan un chip MIPS R2000 de 16,7 MHz para la CPU principal. Stardent-1500 utiliza su unidad vectorial para FP escalar y tiene aproximadamente la mitad del rendimiento escalar (como se midió por la mínima frecuencia de la MIPS M/120, que utiliza el chip MIPS R2010 FP. La máquina vectorial es más de 2,5 veces más rápida para un bucle altamente vectorizable (frecuencia máxima). Sin embargo, el rendimiento más bajo escalar de Stardent-1500 anula el rendimiento vectorial más alto cuando el rendimiento total se mide por la media armónica en los 24 bucles.

menos ciclos de reloj, la Y-MP utiliza menos instrucciones y es mucho más rápida globalmente, ya que tiene un ciclo de reloj de un-décimo de duración.

Falacia: Se puede obtener rendimiento vectorial sin proporcionar suficiente ancho de banda de memoria.

Como vimos con el bucle SAXPY, la anchura de banda de memoria es bastante importante. SAXPY requiere 1,5 referencias a memoria por operación en punto flotante, y esta relación es normal en muchos códigos científicos. Incluso si las operaciones en punto flotante no consumiesen tiempo, un CRAY-1 no podría incrementar el rendimiento de la secuencia vectorial, ya que está limitado por el acceso a memoria. Recientemente, el rendimiento de CRAY-1 sobre Linpack ha saltado porque el compilador utilizaba transformaciones inteligentes al cambiar el cálculo para que los valores pudieran mantenerse en registros vectoriales. ¡Esto bajó el número de referencias a memoria por FLOP y mejoró el rendimiento aproximadamente en un factor de 2! Por tanto, el ancho de banda de memoria en el CRAY-1 llegó a ser suficiente para un bucle que, en principio, requería más anchura de banda.

7.9 Observaciones finales

A finales de los años ochenta, el rápido incremento del rendimiento en las máquinas escalares con buenas segmentaciones, condujo a una disminución espectacular del espacio vacío entre los supercomputadores vectoriales, que costaban millones de dólares, y los microprocesadores rápidos VLSI segmentados, que costaban menos de 100 000 dólares. La razón básica fue el rápido decrecimiento del CPI de las máquinas escalares.

Para los programas científicos, una contrapartida interesante al CPI es la de *ciclos de reloj por FLOP*, o CPF. Vimos en este capítulo que, para máquinas vectoriales, este número estaba normalmente en el rango de 2 (para una máquina estilo CRAY X-MP) a 4 (para una máquina estilo CRAY-1). En el último capítulo, vimos que las máquinas segmentadas variaban desde aproximadamente 6 (para DLX) disminuyendo hasta 1,5 (para un DLX superescalar sin pérdidas del sistema de memoria ejecutando un bucle tipo SAXPY).

Las tendencias recientes en el diseño de máquinas vectoriales se han centrado en el alto rendimiento vectorial y multiprocesamiento. Mientras tanto, las máquinas escalares de alta velocidad se concentran en mantener la relación del rendimiento máximo al sostenido cercana a uno. Por ello, si las velocidades máximas (de pico) avanzan comparablemente, las velocidades sostenidas de las máquinas escalares avanzarán más rápidamente, y las máquinas escalares continuarán aproximando las separaciones actuales del CPF. Estas máquinas escalares puede rivalizar o superar el rendimiento de las máquinas vectoriales con frecuencias de reloj comparables, especialmente para niveles de vectorización inferiores al 70 por 100. Además, las diferencias en la frecuencia de reloj están conducidas enormemente por la tecnología —las máquinas vectoriales basadas en microprocesador, de bajas prestaciones, tienen frecuencias de reloj comparables a las máquinas segmentadas que utilizan tec-

nología de microprocesadores. (En efecto, ¡a menudo utilizan los mismos microprocesadores!) En el futuro, podemos esperar máquinas escalares segmentadas de alta velocidad construidas con frecuencias de reloj que rivalizarán con las de los supercomputadores vectoriales actuales. Sin embargo, las máquinas vectoriales deberán conservar ventaja en el rendimiento para problemas con vectores muy largos que puedan utilizar múltiples accesos a memoria y conseguir rendimientos próximos al máximo.

En los años noventa será interesante ver cuando las máquinas escalares segmentadas, que explotan más paralelismo a nivel de instrucción y habitualmente son más baratas (debido a que su rendimiento máximo y por tanto el hardware total es mucho menor), comenzaran a ofrecer niveles de rendimiento para muchas aplicaciones que sean difíciles de distinguir de los de las máquinas vectoriales.

7.10 Perspectiva histórica y referencias

Las primeras máquinas vectoriales fueron el STAR-100 de CDC (ver Hintz y Tate [1972]) y la ASC de TI (ver Watson [1972]), ambas anunciadas en 1972. Ambas eran máquinas vectoriales memoria-memoria. Tenían unidades escalares relativamente lentas —la STAR utilizó las mismas unidades para escalares y vectores—, haciendo la longitud de las operaciones escalares extremadamente profundo. Ambas máquinas tenían altos gastos de arranque y trabajaban sobre vectores de varios cientos a varios miles de elementos. El cruce entre escalar y vector podría estar sobre 50 elementos. Parece ser que en estas dos máquinas no se prestó suficiente atención al papel de la Ley de Amdahl.

Cray, que trabajó en el 6600 y en el 7600 en CDC, fundó Cray Research e introdujo el CRAY-1 en 1976 (ver Russell [1978]). El CRAY-1 utilizaba una arquitectura con registros vectoriales para disminuir significativamente los costes de arranque. También tenía un soporte eficiente para separación entre elementos no unidad e inventó el encadenamiento. Más importante, el CRAY-1 era también la máquina escalar más rápida del mundo en esa época. Esta combinación de buen rendimiento escalar y vectorial fue probablemente el factor más significativo que contribuyó al éxito del CRAY-1. Algunos clientes compraron la máquina principalmente por su excepcional rendimiento escalar. Muchas máquinas vectoriales posteriores están basadas en la arquitectura de esta primera máquina vectorial de éxito comercial. Baskett y Keller [1977] es una buena evaluación del CRAY-1.

En 1981, CDC comenzó a construir el CYBER-205 (ver Lincoln [1982]). El 205 tenía la misma arquitectura básica que el STAR, pero ofrecía mejores rendimientos, así como expandibilidad de la unidad vectorial con hasta cuatro conjuntos para cálculo vectorial, cada uno con varias unidades funcionales y un canal ancho de carga/almacenamiento que proporcionaba varias palabras por ciclo de reloj. El rendimiento pico del CYBER-205 excedía enormemente al rendimiento del CRAY-1. Sin embargo, en programas reales, la diferencia de rendimiento era mucho menor.

La máquina STAR de CDC y su descendiente, CYBER-205, fueron máquinas vectoriales memoria-memoria. Para mantener el hardware sencillo y

soportar los requerimientos de ancho de banda elevado (hasta 3 referencias a memoria por FLOP), estas máquinas no manipulaban con eficacia las separaciones entre elementos no unitarios. Aunque muchos bucles tienen separaciones entre elementos unidad, un bucle de separación entre elementos no unidad tiene pobre rendimiento en estas máquinas porque las transferencias de datos memoria-memoria necesitaban agrupar (y después dispersar) los elementos no adyacentes de los vectores.

Schneck [1987] describió algunas de las primeras máquinas segmentadas (p. e., Stretch) a través de las primeras máquinas vectoriales incluyendo el 205 y el CRAY-1. Dongarra [1986] realizó otra buena visión general (survey), centrándose en máquinas más recientes.

En 1983, Cray construyó el primer CRAY X-MP (ver Chen [1983]). Con una frecuencia de reloj mejorada (9,5 ns frente a 12,5 en el CRAY-1), mejor soporte de encadenamiento y varios canales de acceso a memoria, esta máquina mantuvo a Cray Research en el liderato de los supercomputadores. El CRAY-2, un nuevo diseño completamente configurable con hasta cuatro procesadores, se introdujo más tarde. Tiene un reloj mucho más rápido que el X-MP, pero también muchos segmentos. El CRAY-2 carece de encadenamiento; tiene una enorme latencia de memoria, y tiene sólo un canal con memoria por procesador. En general, sólo es más rápida que el CRAY X-MP en problemas que requieren su enorme memoria principal.

En 1983, los vendedores japoneses de computadores entraron en el mercado de los supercomputadores, comenzando con la VP100 y VP200 de Fujitsu (Miura y Uchida [1983]), y en una expansión posterior incluyeron la S810 de Hitachi, y la SX/2 de NEC (ver Watanabe [1987]). Estas máquinas han probado estar próximas a el CRAY X-MP en rendimiento. En general, estas tres máquinas tienen mucha mayor velocidad pico que el CRAY X-MP, aunque, debido a los grandes costes de arranque, su rendimiento normal, con frecuencia, es menor que el del CRAY X-MP (ver Fig. 2.24 en el Cap. 2). El CRAY X-MP favoreció el uso de varios procesadores, ofreciendo primero una versión de dos procesadores y más tarde una máquina de cuatro procesadores. En contraste, las tres máquinas japonesas tenían capacidades vectoriales expandibles. En 1988, Cray Research introdujo el CRAY Y-MP —una versión mayor y más rápida que el X-MP. El Y-MP permitía hasta ocho procesadores y disminuía la duración del ciclo a 6 ns—. Con un juego completo de ocho procesadores, el Y-MP es, generalmente, el supercomputador más rápido, aunque los supercomputadores japoneses de un procesador pueden ser más rápidos que un Y-MP de un procesador. A finales de 1989 Cray Research se dividió en dos compañías, ambas dedicadas a construir las máquinas de altas prestaciones, disponibles a principio de los años noventa. Seymour Cray continúa a la cabeza de una gran empresa, que ahora se llama Cray Computer Corporation.

A primeros de los años ochenta, CDC creó un grupo, llamado ETA, para construir un nuevo supercomputador, el ETA-10, capaz de 10 GigaFLOP. La máquina ETA que apareció a finales de los años ochenta (ver Fazio [1987]) utilizaba CMOS de baja temperatura en una configuración con hasta 10 procesadores. Cada procesador conservaba la arquitectura memoria-memoria basada en el CYBER-205. Aunque el ETA-10 consiguió una gran velocidad pico, su velocidad escalar no era comparable. En 1989, CDC, el primer vendedor

de supercomputadores, cerró ETA y dejo el negocio de diseño de supercomputadores.

En 1986, IBM introdujo la arquitectura vectorial System/370 (ver Moore y cols. [1987]) y su primera implementación en la 3090 Vector Facility. La arquitectura amplía la arquitectura de System/370 con 171 instrucciones vectoriales. La 3090/VF está integrada en la CPU 3090. De forma distinta a muchas otras máquinas vectoriales, la 3090/VF canaliza sus vectores a través de la cache.

Los años ochenta también vieron la llegada de máquinas vectoriales de menor escala, llamadas mini-supercomputadores. Su precio era aproximadamente la décima parte del coste de un supercomputador (0,5 a 1 millón de dólares frente a 5 a 10 millones de dólares), estas máquinas cayeron rápidamente. Aunque muchas compañías accedieron al mercado, las dos que han tenido más éxito son Convex y Alliant. Convex comenzó con una máquina vectorial uniprocesador (C-1) y ahora ofrece un pequeño multiprocesador (C-2); enfatizando la capacidad de software de Cray. Alliant [1987] se ha concentrado más en los aspectos de multiprocesadores. Construyeron una máquina de ocho procesadores, cada uno de los cuales ofrecía capacidad vectorial.

La base para la tecnología de modernos compiladores que vectorizan y la noción de dependencia de datos fue desarrollada por Kuck y sus colegas [1974] en la Universidad de Illinois. Banerjee [1979] desarrolló un test, que hoy día se conoce con su nombre; Padua y Wolf [1986] dieron una buena visión general de la tecnología de los compiladores que vectorizan.

Los estudios de benchmarks de varios supercomputadores incluyendo intentos de comprender las diferencias de rendimiento han sido realizados por Lubeck, Moore y Mendez [1985], Bucher [1983] y Jordan [1987]. En el Capítulo 2, explicamos varios benchmarks dedicados a utilización científica y, con frecuencia, empleados para realizar pruebas de rendimiento en supercomputadores, incluyendo Linpack, los «kernels» FORTRAN de Lawrence Livermore Laboratories, y la colección de Perfect Club.

A finales de los ochenta, los supercomputadores gráficos llegaron al mercado con Stellar [Sporer, Moss y Mathais, 1988] y Ardent [Miranker, Rubenstein y Sanguinetti, 1988]. La máquina Stellar utilizaba una segmentación de tiempo compartido para permitir tratamiento vectorial de alta velocidad y multitarea eficiente. Esta aproximación se utilizó antes en la máquina diseñada por B. J. Smith [1981] llamada HEP y construida por Denelcor a mediados de los años ochenta. Este enfoque no tiene rendimiento escalar de alta velocidad, como evidencian los benchmarks escalares de la máquina Stellar. La máquina Ardent combina un procesador RISC (el MIPS R2000) con una unidad vectorial a medida. Estas máquinas vectoriales, que cuestan aproximadamente 100K dólares, tienen capacidades vectoriales para un nuevo mercado potencial. A finales de 1989, Stellar y Ardent se unieron para formar Stardent, y la arquitectura Ardent se está construyendo desde la compañía conjunta.

Desde esta panorámica podemos ver el progreso que han hecho las máquinas vectoriales. En menos de veinte años han evolucionado desde nuevas arquitecturas no probadas a jugar un papel significante con el objetivo de proporcionar a los ingenieros y científicos aún mayores cantidades de potencia de cálculo.

Referencias

ALLIANT COMPUTER SYSTEMS CORP. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass.

BANERJEE, U. [1979]. *Speedup of Ordinary Programs*, Ph. D. Thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (October).

BASKETT, F., AND T. W. KELLER [1977]. «An Evaluation of the CRAY-1 Computer», in *High Speed Computer and Algorithm Organization*, Kuck, D. J., Lawrie, D. H. and A. H. Sameh, eds., Academic Press, 71-84.

BUCHER, I. Y. [1983]. «The computational speed of supercomputers», *Proc. SIGMETRICS Conf. on Measuring and Modeling of Computer Systems*, ACM (August), 151-165.

CALLAHAN, D., J. DONGARRA, AND D. LEVINE [1988]. «Vectorizing compilers: A test suite and results», *Supercomputing'88*, ACM/IEEE (November), Orlando, Fla., 98-105.

CHEN, S. [1983]. «Large-scale and high-speed multiprocessor system for scientific applications», *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., «Supercomputers: Design and applications», *IEEE* (August), 1984.

DONGARRA, J. J. [1986]. «A survey of high performance computers», *COMPCON, IEEE* (March), 8-11.

FAZIO, D. [1987]. «It's really much more fun building a supercomputer than it is simply inventing one», *COMPCON, IEEE* (February), 102-105.

FLYNN, M. J. [1966]. «Very high-speed computing systems», *Proc. IEEE*, 54:12 (December), 1901-1909.

HINTZ, R. G., AND D. P. TATE [1972]. «Control data STAR-100 processor design», *COMPCON, IEEE* (September), 1-4.

JORDAN, K. E. [1987]. «Performance comparison of large-scale scientific computers: Scalar mainframes, mainframes with vector facilities, and supercomputers», *Computer* 20:3 (March), 10-23.

KUCK, D., P. P. BUDNIK, S.-C. CHEN, D. H. LAWRIE, R. A. TOWLE, R. E. STREBENDT, E. W. DAVIS, JR, J. HAN, P. W. KRASKA, Y. MURAOKA [1974]. «Measurements of parallelism in ordinary FORTRAN programs», *Computer*, 7:1 (January), 37-46.

LINCOLN, N. R. [1982]. «Technology and design trade offs in the creation of a modern supercomputer», *IEEE Trans. on Computers* C-31:5 (May), 363-376.

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. «A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2», *Computers*, 18:1 (January), 10-29.

MIRANKER, G. S., J. RUBENSTEIN, AND J. SANGUINETTI [1988]. «Squeezing a Cray-class supercomputer into a single-user package», *COMPCON, IEEE* (March), 452-456.

MIURA, K., AND K. UCHIDA [1983]. «FACOM vector processing system: VP100/200», *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., «Supercomputers: Design and applications», *IEEE* (August 1984), 59-73.

MOORE, B., A. PADEGS, R. SMITH, AND W. BUCHOLZ [1987]. «Concepts of the System/370 vector architecture», *Proc. 14th Tymposium on Computer Architecture* (June), ACM/IEEE, Pittsburgh, Pa., 282-292.

PADUA, D., AND M.WOLFE [1986]. «Advanced compiler optimizations for supercomputers», *Comm. ACM*, 29:12 (December), 1184-1201.

RUSSELL, R. M. [1978]. «The CRAY-1 computer system», *Comm. of the ACM*, 21:1 (January), 63-72.

SCHNECK, P. B. [1987]. *Supercomputer Architecture*, Kluwer Academic Publishers, Norwell, Mass.

SMITH, B. J. [1981]. «Architecture and applications of the HEP multiprocessor system», *Real-Time Signal Processing IV*, 298 (August), 241-248.

SPORER, M., F. H. MOSS AND C. J. MATHAIS [1988]. «An introduction to the architecture of the Stellar Graphics supercomputer», *COMPCON, IEEE* (March), 464-467.

WATANABE, T. [1987]. «Architecture and performance of the NEC supercomputer SX system», *Parallel Computing*, 5, 247-255.

WATSON, W. J. [1972]. «The TI ASC-A highly modular and flexible super computer architecture», *Proc. AFIPS Fall Joint Computer Conf.*, 221-228.

EJERCICIOS

En estos ejercicios suponer que DLXV tiene una frecuencia de reloj de 80 MHz y que $T_{base} = 10$ y $T_{bucle} = 15$. Suponer también que la latencia de almacenamiento está siempre incluida en el tiempo de ejecución.

7.1 [10] <7.1-7.2> Escribir una secuencia vectorial de DLXV que consiga el rendimiento máximo (de pico) en MFLOPS de la máquina (utilizar la unidad funcional y descripción de instrucciones de la Sección 7.2). Suponiendo una frecuencia de reloj de 80 MHz, ¿cuál es la velocidad pico en MFLOPS?

7.2 [20/15/15] <7.1-7.6> Considerar el siguiente código vectorial ejecutándose en una versión de 80 MHz de DLXV para un vector de longitud 64:

```
LV    V1,Ra
MULTV V2,V1,V3
ADDV  V4,V1,V3
SV    Rb, V2
SV    Rc,V4
```

Ignorar todos los gastos de seccionamiento, pero suponer que la latencia del almacenamiento debe estar incluida en el tiempo de realizar el bucle. La secuencia completa produce 64 resultados.

a. [20] Suponiendo no encadenamiento y un solo canal de acceso con memoria, ¿cuántos ciclos de reloj por resultado (incluyendo ambos almacenamientos como un resultado) necesita esta secuencia vectorial?

b. [15] Si la secuencia vectorial está encadenada, ¿cuántos ciclos de reloj por resultado necesita esta secuencia?

c. [15] Suponer que DLXV tuviese tres canales de acceso con memoria y encadenamiento. Si no hubiese conflicto de bancos en los accesos para el bucle anterior, ¿cuántos ciclos de reloj se necesitarían por resultado para esta secuencia?

7.3 [20/20/15/15/20/20/20] <7.2-7.7> Considerar el siguiente código FORTRAN:

```
      do 10 i=1,n
         A(i) = A(i) + B(i)
         B(i) = x · B(i)
10    continue
```

Utilizar las técnicas de la Sección 7.7 para estimar el rendimiento en este ejercicio, suponiendo una versión de 80 MHz de DLXV.

a. [20] Escribir el mejor código vectorial de DLXV para la parte interior del bucle. Suponer que x está en F0 y que las direcciones de `A` y `B` están en `Ra` y `Rb`, respectivamente.

b. [20] Calcular el tiempo total para este bucle en DLXV (T_{100}). ¿Cuál es la velocidad en MFLOP para el bucle (R_{100})?

c. [15] Calcular R_∞ para este bucle.

d. [15] Calcular $N_{1/2}$ para este bucle.

e. [20] Calcular N_v para este bucle. Suponer que el código escalar se ha planificado en los segmentos para que cada referencia a memoria use seis ciclos y cada operación FP 3 ciclos. Suponer que el gasto escalar también es T_{bucle}.

f. [20] Suponer que DLXV tiene dos canales de acceso con memoria. Escribir el código vectorial que aprovecha el segundo canal con memoria.

g. [20] Calcular T_{100} y R_{100} para DLX con dos cauces a memoria.

7.4 [20/10] <7.3> Suponer que tenemos una versión de DLXV con ocho bancos de memoria (cada uno con una anchura de doble palabra) y un tiempo de acceso a memoria de ocho ciclos.

a. [20] Si se ejecuta una lectura de un vector de longitud 64 con una separación entre elementos de 20 dobles palabras, ¿cuántos ciclos tardará en completarse la carga?

b. [10] ¿Qué porcentaje de la anchura de banda de memoria se conseguirá con una carga de 64 elementos con una separación entre elementos de 20 en comparación con una separación entre elementos de 1?

7.5 [12/12/20] <7.4-7.7> Considerar el siguiente bucle:

```
      C = 0.0
      do 10 i=1,64
         A(i) = A(i) + B(i)
         C = C + A(i)
   10 continue
```

a. [12] Dividir el bucle en dos: uno sin dependencia y otro con dependencia: escribir estos bucles en FORTRAN —como una transformación fuente a fuente—. Esta optimización se denomina *fisión del bucle.*

b. [12] Escribir el código vectorial de DLXV para el bucle sin dependencia.

c. [20] Escribir el código de DLXV para evaluar el bucle dependiente utilizando doblamiento recursivo.

7.6 [20/15/20/20] <7.5-7.7> El rendimiento del Linpack compilado del CRAY-1 (diseñado en 1976) fue casi doblado por un compilador mejor en 1989. Veamos un sencillo ejemplo de cómo puede ocurrir esto. Considerar el bucle «como-SAXPY» (donde k es un parámetro para el procedimiento que contiene el bucle):

```
      do 10 i=1,64
         do 10 j=1,64
         Y(k,j) = a*X(i,j) + Y(k,j)
10    continue
```

a. [20] Escribir la secuencia de código **correcta** para el bucle interior en instrucciones vectoriales de DLXV.

b. [15] Utilizando las técnicas de la Sección 7.7, estimar el rendimiento de este código en DLXV, calculando T_{64} en ciclos de reloj. Se puede suponer que T_{base} se aplica una vez y que un gasto de T_{bucle} se contrae en cada iteración del bucle externo. ¿Qué limita el rendimiento?

c. [20] Reescribir el código DLXV para reducir la limitación de rendimiento; mostrar el bucle interior resultante en instrucciones vectoriales de DLXV. (Sugerencia: pensar sobre qué establece $T_{elemento}$; ¿puedes modificarla?) Encontrar el tiempo total para la secuencia resultante.

d. [20] Estimar el rendimiento de la nueva versión utilizando las técnicas de la Sección 7.7 y calculando T_{64}.

7.7 [15/15/25] <7.6> Considerar el siguiente código.

```
      do 10 i=1,64
          if (B(i) .ne. 0) then
             A(i) = A(i) / B(i)
             endif
10    continue
```

Suponer que las direcciones de A y B están en Ra y Rb, respectivamente, y que F0 contiene 0.

a. [15] Escribir el código de DLXV para este bucle utilizando la capacidad del vector de máscara.

b. [15] Escribir el código de DLXV para este bucle utilizando el método de dispersar/agrupar.

c. [25] Estimar el rendimiento (T_{100} en ciclos de reloj) de estos dos bucles vectoriales suponiendo una latencia de la división de 20 ciclos. Suponer que todas las instrucciones vectoriales se ejecutan a un resultado por ciclo de reloj, independiente de la inicialización del registro de máscara vectorial. Suponer que el 50 por 100 de las entradas de B son 0. Considerando los costes hardware, ¿qué se construiría si el bucle anterior **fuese** típico?

7.8 [15/20/15/15] <7.1-7.7> En la Figura 2.24 del Capítulo 2, veíamos que la diferencia entre rendimiento máximo y sostenido podía ser grande: para un problema, un S810 de Hitachi tenía una velocidad máxima doble que el CRAY X-MP, mientras que para otro problema más realista el CRAY X-MP era el doble de rápido que la máquina de Hitachi. Examinemos por qué puede ocurrir esto utilizando dos versiones de DLXV y las siguientes secuencias de código:

```
C     Secuencia de código 1
      do 10 i=1,10000
         A(i+1) = x · A(i) + y · A(i)
10    continue

C     Secuencia de código 2
      do 10 i=1,100
         A(i+1) = x · A(i)
10    continue
```

Suponer que hay una versión de DLXV (llamada DLXVII) que tiene dos copias de cada unidad funcional de punto flotante con encadenamiento completo entre ellas. Suponer que DLXV y DLXVII tienen dos unidades de carga/almacena-

miento. Debido a las unidades funcionales extra y al aumento de complejidad en las operaciones de asignación a unidades, todos los gastos (T_{base}, T_{bucle}, y gastos de arranque por operación vectorial) se doblan.

a. [15] Calcular el número de ciclos de reloj para la secuencia de código 1 en DLXV.

b. [20] Calcular el número de ciclos de reloj en la secuencia de código 1 para DLXVII. ¿Cómo se compara esto con DLXV?

c. [15] Calcular el número de ciclos de reloj en la secuencia del código 2 para DLXV.

d. [15] Calcular el número de ciclos de reloj en la secuencia de código 2 para DLXVII. ¿Cómo se compara esto con DLXV?

7.9 [15/15/20] <7.5> En este problema examinaremos algunos de los tests de bucles vectoriales explicados en la Sección 7.5 y en la Figura 7.13.

a. [15] Aquí hay un fragmento de código sencillo:

```
          do 400 i = 2,100,2
          a(i-1) = a(50·i+1)
400       continue
```

Para utilizar la prueba de GCD (máximo común divisor) este bucle primero debe ser «normalizado» —escrito para que el índice comience en 1 y se incremente en 1 en cada iteración. Escribir una versión normalizada del bucle (cambiar los índices cuando sea necesario), después utilizar la prueba GCD para ver si puede vectorizar.

b. [15] Aquí se da otro bucle:

```
          do 400 i = 2,100,2
             a(i) = a(i-1)
400       continue
```

Normalizar el bucle y utilizar el test GCD para detectar una dependencia. ¿Hay una dependencia real en el bucle?

c. [20] Ahora daremos un código, un poco complicado, con arrays de dos dimensiones. ¿Puede vectorizarse? Si es así, ¿cómo? Reescribir el código **fuente** para que esté claro que el bucle se pueda vectorizar, si es posible.

```
          do 290 j = 2,n
             do 290 i = 2,j
                aa(i,j)=aa(i-1,j)·aa(i-1,j)+bb(i,j)
290       continue
```

7.10 [25] <7.5> Mostrar que si existe para dos elementos del array A(a·i+b) y A(c·i+d) una dependencia verdadera, entonces GCD(c,a) divide a (d-b).

7.11 [12/15] <7.5> Considerar el siguiente bucle:

```
          do 10 i = 2,n
             A(i) = B
10           C(i) = A(i-1)
```

a. [12] Demostrar que hay dependencia entre iteraciones en este fragmento de código.

b. [15] Reescribir el código en Fortran para que pueda vectorizarse en dos secuencias vectoriales separadas.

7.12 [25] <7.6> Como la diferencia entre los modos escalar y vectorial es tan grande en un supercomputador y las máquinas cuestan, con frecuencia decenas de millones de dólares, los programadores están deseando realizar un esfuerzo extraordinario para lograr un buen rendimiento. Esto, a menudo, incluye programar con trucos en lenguaje ensamblador. Un problema interesante es escribir un algoritmo de ordenación de números en punto flotante —una tarea necesaria a veces en el código científico. Escoger un algoritmo de ordenación y escribir una versión para DLXV que utilice tantas operaciones vectoriales como sea posible. (Sugerencia: una buena elección es la ordenación rápida donde se pueden utilizar las operaciones de comparar, comprimir y expandir vectores.)

7.13 [25] <7.6> En algunas máquinas vectoriales, los registros vectoriales son direccionables, y los operandos para una operación vectorial pueden ser dos partes diferentes del mismo registro vectorial. Esto permite otra solución para la reducción mostrada en la página 410. La idea clave de sumas parciales es reducir el vector a m sumas donde m es la latencia total a través de la unidad funcional vectorial, incluyendo los tiempos de escritura y lectura de operandos. Suponer que los registros del vector DLXV son direccionables (p. e., se puede iniciar una operación vectorial con el operando V1(16), indicando que el operando de entrada comenzó con el elemento 16). Suponer también que la latencia total para sumas, incluyendo lectura y escritura de operandos, es de ocho ciclos. Escribir una secuencia de código de DLXV que reduzca el contenido de V1 a ocho sumas parciales. Puede hacerse con una operación vectorial.

7.14 [40] <7.2-7.6> Extender el simulador DLX al simulador DLXV incluyendo la posibilidad de contar ciclos de reloj. Escribir algunos programas de benchmarks cortos en el lenguaje ensamblador de DLX y DLXV. Medir la velocidad sobre DLXV, el porcentaje de vectorización y la utilización de unidades funcionales.

7.15 [50] <7.5> Modificar el compilador de DLX para que incluya un comprobador de dependencias. Ejecutar algún código científico y hacer bucles en él y medir qué porcentaje de las sentencias pueden ser vectorizadas.

7.16 [Discusión] Algunos defensores de las máquinas vectoriales pueden argüir que los procesadores vectoriales han proporcionado el mejor camino para aumentar la potencia de cálculo de los computadores al centrar su atención en aumentar el rendimiento vectorial máximo. Otros argüirán que el énfasis en el rendimiento máximo está desplazado porque un porcentaje creciente de los programas están dominados por rendimiento no vectorial. (¿Recuerda la Ley de Amdahl?) Los defensores responderán que los programadores deberían trabajar para hacer sus programas vectorizables. ¿Qué piensa usted sobre este argumento?

7.17 [Discusión] Considerar los puntos surgidos en Observaciones finales (Sección 7.9). Este aspecto —las ventajas relativas de las máquinas escalares segmentadas frente a las máquinas vectoriales FP— es la fuente de muchos debates a principios de los años noventa. ¿Qué ventajas ve usted por cada lado? ¿Qué haría en esta situación?

Idealmente sería deseable una capacidad indefinidamente grande de memoria tal que cualquier particular... palabra estuviese inmediatamente disponible... Estamos... forzados a reconocer la posibilidad de construir una jerarquía de memorias, cada una de las cuales tenga mayor capacidad que la precedente pero que sea menos rápidamente accesible.

A. W. Burks, H. H. Goldstine y J. von Neumann,
Discusión preliminar del diseño lógico de un instrumento de cálculo electrónico (1946)

8.1 Introducción: principio de localidad

8.2 Principios generales de jerarquía de memoria

8.3 Caches

8.4 Memoria principal

8.5 Memoria virtual

8.6 Protección y ejemplos de memoria virtual

8.7 Más optimizaciones basadas en el comportamiento de los programas

8.8 Tópicos avanzados. Mejora del rendimiento de memoria cache

8.9 Juntando todo: la jerarquía de memoria del VAX-11/780

8.10 Falacias y pifias

8.11 Observaciones finales

8.12 Perspectiva histórica y referencias

Ejercicios

8 Diseño de la jerarquía de memoria

8.1 Introducción: principio de localidad

Los pioneros de los computadores predijeron correctamente que los programadores querrían cantidades ilimitadas de memoria rápida. Como predice la regla 90/10 del primer capítulo, la mayoría de los programas, afortunadamente, no acceden a todo el código o a los datos de memoria uniformemente (ver Sección 1.3). La regla 90/10 se puede replantear como el *principio de localidad*. Esta hipótesis, que mantiene que todos los programas favorecen una parte de su espacio de direcciones en cualquier instante de tiempo, tiene dos dimensiones:

- *Localidad temporal* (localidad en el tiempo). Si se referencia un elemento, tenderá a ser referenciado pronto.
- *Localidad espacial* (localidad en el espacio). Si se referencia un elemento, los elementos cercanos a él tenderán a ser referenciados pronto.

Una jerarquía de memoria es una reacción natural a la localidad y tecnología. El principio de localidad y la directriz que el hardware más pequeño es más rápido mantienen el concepto de una jerarquía basada en diferentes velocidades y tamaños. Como la memoria más lenta es más barata, una jerarquía de memoria está organizada en varios niveles —cada uno más pequeño, más rápido y más caro por byte que el nivel siguiente—. Los niveles de la jerarquía están contenidos en el siguiente; todos los datos de un nivel se encuentran también en el nivel siguiente, y todos los datos en ese nivel inferior

se encuentran en el siguiente a él, y así sucesivamente hasta que alcancemos el extremo inferior de la jerarquía.

Este capítulo incluye una media docena de ejemplos que demuestran cómo aprovechando el principio de localidad, se puede mejorar el rendimiento. Todas estas estrategias hacen corresponder direcciones desde una memoria mayor a una memoria más pequeña pero más rápida. Como parte de la correspondencia de direcciones, la jerarquía de memoria habitualmente tiene la responsabilidad de comprobar direcciones; esquemas de protección utilizados para hacer esto se cubren en este capítulo. Más tarde exploraremos aspectos avanzados de la jerarquía de memoria y rastrearemos un acceso a memoria a través de tres niveles de memoria en el VAX-11/780.

8.2 Principios generales de jerarquía de memoria

Antes de proceder con ejemplos de jerarquías de memoria, definamos algunos términos generales aplicables a todas las jerarquías de memoria. Una jerarquía de memoria normalmente consta de muchos niveles, pero en cada momento se gestiona entre dos niveles adyacentes. El nivel *superior* —el más cercano al procesador— es más pequeño y más rápido que el nivel *inferior* (ver Fig. 8.1). La mínima unidad de información que puede estar presente o no presente en la jerarquía de dos niveles se denomina *bloque.* El tamaño de un bloque puede ser fijo o variable. Si es fijo, el tamaño de memoria es un múltiplo de ese tamaño de bloque. La mayor parte de este capítulo se centra en tamaños de bloque fijos, aunque en la Sección 8.6 se explica un diseño de bloques variables.

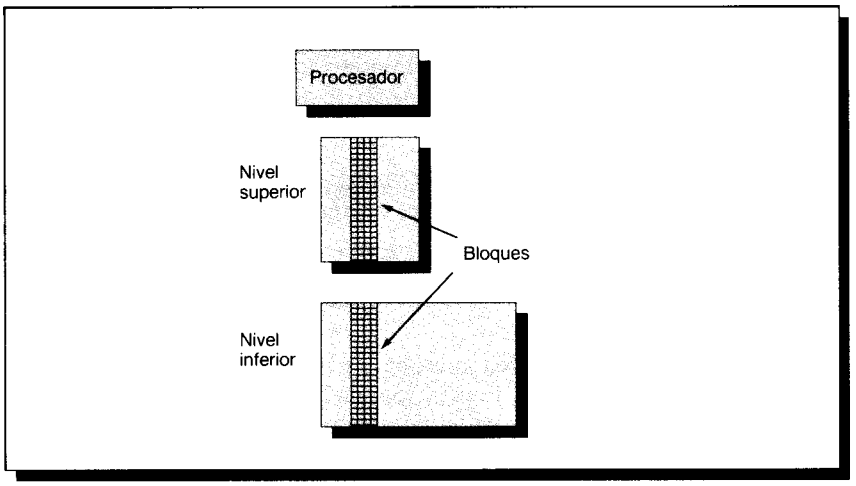


FIGURA 8.1 Cada par de niveles de la jerarquía de memoria se puede considerar como un nivel superior y otro inferior. En cada nivel la unidad de información que está o no presente se denomina *bloque.*

El éxito o fracaso de un acceso al nivel superior se designa como acierto o fallo: un *acierto (hit)* es un acceso a memoria que se encuentra en el nivel superior, mientras que un *fallo (miss)* significa que no se encuentra en ese nivel. La *frecuencia de aciertos* o tasa de aciertos —como un promedio— es la fracción de accesos a memoria encontrados en el nivel superior. Esto a veces se representa como un porcentaje. La *frecuencia de fallos* (1,0 – frecuencia de aciertos) es la fracción de accesos a memoria no encontrados en el nivel superior.

Como el rendimiento es la principal razón para tener una jerarquía de memoria, la velocidad de aciertos y fallos es importante. El *tiempo de acierto* es el tiempo para acceder al nivel superior de la jerarquía de memoria, que incluye el tiempo para determinar si el acceso es un acierto o un fallo. *Penalización de fallo* es el tiempo para sustituir un bloque del nivel superior por el bloque correspondiente del nivel más bajo, más el tiempo en proporcionar este bloque al dispositivo que lo ha pedido (normalmente la CPU). La penalización de fallo se divide además en dos componentes: *tiempo de acceso* —el tiempo para acceder a la primera palabra de un bloque en un fallo— y *tiempo de transferencia* —el tiempo adicional para transferir las restantes palabras de bloque—. El tiempo de acceso está relacionado con la latencia del nivel más bajo de memoria, mientras que el tiempo de transferencia está relacionado con el ancho de banda entre las memorias de nivel superior y nivel inferior. (A veces se utiliza *latencia de acceso* para significar tiempo de acceso.)

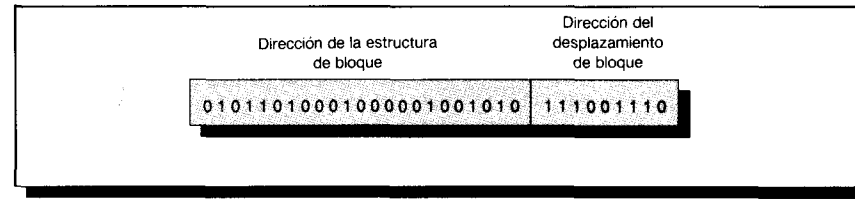


FIGURA 8.2 Ejemplo de las partes de dirección de la estructura y dirección del desplazamiento de una dirección de memoria del nivel inferior de 32 bits. En este caso el tamaño del bloque es 512, haciendo el tamaño de la dirección de desplazamiento de 9 bits y el tamaño de la dirección de la estructura del bloque de 23 bits.

La dirección de memoria está dividida en piezas que acceden a cada parte de la jerarquía. La *dirección de la estructura del bloque* es la parte de orden superior de la dirección, que identifica un bloque en ese nivel de la jerarquía (ver Fig. 8.2). La *dirección del desplazamiento del bloque* es la parte de orden inferior de la dirección e identifica un elemento en un bloque. El tamaño de la dirección del desplazamiento de bloque es $\log_2$ (tamaño de bloque); el tamaño de la dirección de la estructura del bloque es entonces el tamaño de la dirección completa en este nivel menos el tamaño de la dirección de desplazamiento de bloque.

Evaluación del rendimiento de una jerarquía de memoria

Debido a que el número de instrucciones es independiente del hardware, podría pensarse en evaluar el rendimiento de la CPU utilizando ese número. Sin embargo, como vimos en los Capítulos 2 y 4, las medidas indirectas del rendimiento han sido erróneamente utilizadas por muchos diseñadores de computadores. La tentación correspondiente de evaluar el rendimiento de la jerarquía de memoria es concentrarse en la frecuencia de fallos, ya que ésta, también, es independiente de la velocidad del hardware. Como veremos, la frecuencia de fallos puede ser tan engañosa como el número de instrucciones. Una mejor medida del rendimiento de la jerarquía de memoria es el tiempo medio para acceder a memoria:

$$\text{Tiempo medio de acceso a memoria} = \text{Tiempo de acierto} + \\ + \text{Frecuencia de fallos} \cdot \text{Penalización de fallo}$$

Los componentes del tiempo medio de acceso se pueden medir bien en tiempo absoluto —por ejemplo, 10 ns en un acierto— o en el número de ciclos de reloj que la CPU espera la memoria —como, por ejemplo, una penalización de fallo de 12 ciclos de reloj—. Recordar que el tiempo medio de acceso a memoria es todavía una medida indirecta del rendimiento; así, aunque sea una medida mejor que la frecuencia de fallos, no es un sustituto del tiempo de ejecución.

La relación entre tamaño de bloque y penalización de fallos, así como frecuencia de fallos se muestra abstractamente en la Figura 8.3. Estas represen-

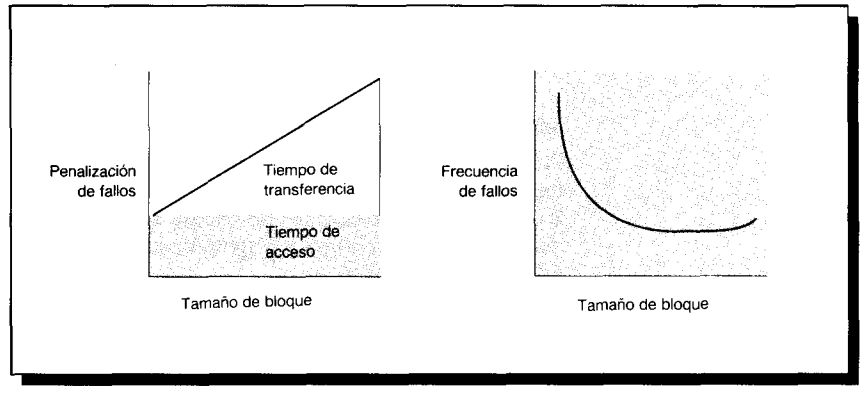


FIGURA 8.3 Tamaño de bloque frente a penalización de fallos y frecuencia de fallos. La parte correspondiente al tiempo de transferencia en la penalización de fallos, obviamente, crece al incrementar el tamaño del bloque. Para una memoria de nivel superior de tamaño fijo, las frecuencias de fallos disminuyen al incrementar el tamaño de bloque hasta que la parte no utilizada del bloque llega a ser tan grande que quita el sitio a información útil del nivel superior; entonces la frecuencia de fallos comienza a aumentar. El punto, en la curva de la derecha, donde la frecuencia de fallos comienza a aumentar con el incremento del tamaño de bloque se denomina, a veces, *punto de polución*.

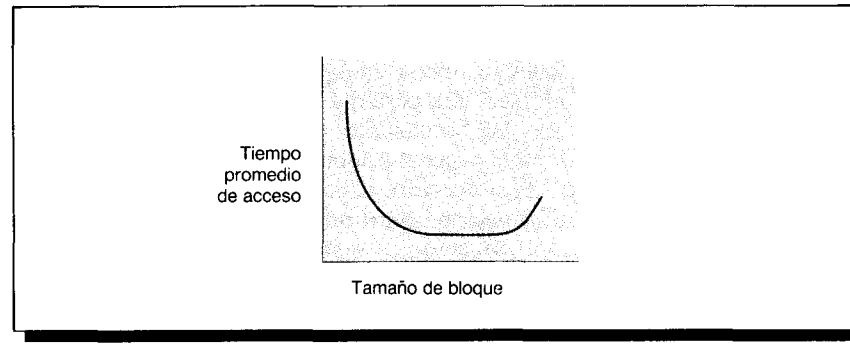


FIGURA 8.4 La relación entre tiempo medio de acceso a memoria y tamaño de bloque.

taciones suponen que no cambia el tamaño de memoria del nivel superior. La parte del tiempo de acceso de la penalización de fallos no está afectada por el tamaño de bloque, pero el tiempo de transferencia aumenta con el tamaño del bloque. Si el tiempo de acceso es grande, la penalización adicional cuando aumenta el tamaño del bloque será inicialmente pequeña. Sin embargo, incrementar el tamaño de bloque significa menos bloques en el nivel superior de memoria. Incrementar el tamaño de bloque hace disminuir la frecuencia de fallos hasta que la reducción de fallos por bloques mayores (localidad especial) es contrarrestada por el aumento de fallos debido a un número menor de bloques.

El objetivo de una jerarquía de memoria es reducir el tiempo de ejecución, no los fallos. Por consiguiente, los diseñadores de computadores prefieren un tamaño de bloque con tiempo de acceso medio menor mejor que una frecuencia de fallos más baja. Esto está relacionado con el producto de la frecuencia de fallos y penalización de fallos, como muestra abstractamente la Figura 8.4. Por supuesto, el rendimiento global de la CPU es el último test de rendimiento, así que debe tenerse cuidado cuando se reduzca el tiempo medio de acceso a memoria para asegurarse que los cambios en la duración del ciclo de reloj y el CPI mejoran el rendimiento global así como el tiempo medio de acceso a memoria.

Implicaciones de una jerarquía de memoria a la CPU

Los procesadores diseñados sin jerarquía de memoria son más simples porque los accesos a memoria siempre emplean la misma cantidad de tiempo. Los fallos en una jerarquía de memoria significan que la CPU debe poder manejar tiempos de acceso a memoria variables. Si la penalización de fallos es del orden de decenas de ciclos de reloj, el procesador normalmente espera a que se complete la transferencia a memoria. Por otro lado, si la penalización de fallos es de miles de ciclos de reloj del procesador, es muy costoso dejar a la CPU inactiva; en este caso, la CPU es interrumpida y utilizada para otro proceso durante el tratamiento del fallo. Por tanto, evitar los gastos de una gran

penalización de la fallo significa que cualquier acceso a memoria puede producir una interrupción de la CPU. Esto también significa que la CPU debe poder recuperar cualquier dirección de memoria que pueda provocar dicha interrupción, para que el sistema pueda saber qué transferir para satisfacer el fallo (ver Sección 5.6). Cuando se completa la transferencia a memoria, se restaura el proceso original y se reintenta la instrucción que falló.

El procesador debe tener también algún mecanismo para determinar si la información está o no en el nivel superior de la jerarquía de memoria. Esta comprobación se realiza en cada acceso a memoria y afecta al tiempo de acierto; mantener un rendimiento aceptable requiere, habitualmente, que la comprobación se implemente en hardware. La implicación final de una jerarquía de memoria es que el computador debe tener un mecanismo para transferir bloques entre la memoria de nivel superior e inferior. Si la transferencia de bloques es de decenas de ciclos de reloj, se controla por hardware; si es de miles de ciclos de reloj, se puede controlar por software.

Cuatro preguntas para clasificar las jerarquías de memoria

Los principios fundamentales que rigen todas las jerarquías de memoria nos permiten utilizar términos que transcienden los niveles de los que estamos hablando. Estos mismos principios nos permiten plantear cuatro preguntas sobre cualquier nivel de la jerarquía:

P1. ¿Dónde puede ubicarse un bloque en el nivel superior? *(Ubicación de bloque)*

P2. ¿Cómo se encuentra un bloque si está en el nivel superior? *(Identificación de bloque)*

P3. ¿Qué bloque debe reemplazarse en caso de fallo? *(Sustitución de bloque)*

P4. ¿Qué ocurre en una escritura? *(Estrategia de escritura)*

Estas preguntas nos ayudarán a comprender mejor los diferentes compromisos demandados por las relaciones entre las memorias a diferentes niveles de una jerarquía.

8.3 Caches

Cache: es un sitio seguro para ocultar o almacenar cosas.

Nuevo Diccionario Mundial de Webster del Lenguaje Americano. Segunda edición de Colegio (1976)

Cache es el nombre inicialmente escogido para representar el nivel de jerarquía de memoria entre la CPU y memoria principal, y este es el uso dominante del término. Aunque el concepto de cache es más joven que la arquitectura IBM 360, hoy día aparecen caches en toda clase de computadores y en

Tamaño de bloque (línea)	4-128 bytes
Tiempo de acierto	1-4 ciclos de reloj (normalmente 1)
Penalización de fallo	8-32 ciclos de reloj
(tiempo de acceso)	(6-10 ciclos de reloj)
(tiempo de transferencia)	(2-22 ciclos de reloj)
Frecuencia de fallo	1 % - 20 %
Tamaño de cache	1 KB - 256 KB

FIGURA 8.5 Valores típicos de parámetros clave de la jerarquía de memoria para caches en estaciones de trabajo y minicomputadores de 1990.

algunos computadores más de una vez. En efecto, la palabra se ha hecho tan popular que ha sustituido a «buffer» en muchos círculos informáticos.

Los términos generales definidos en la sección anterior se pueden utilizar para caches, aunque se utiliza con frecuencia la palabra *línea* en lugar de bloque. La Figura 8.5 muestra el rango típico de parámetros de la jerarquía de memoria para caches.

Ahora examinemos con más detalle las caches respondiendo a las cuatro preguntas sobre jerarquías de memoria.

P1. ¿Dónde puede ubicarse un bloque en una cache?

Las restricciones sobre dónde se coloca un bloque crean tres categorías de organización cache:

- Si cada bloque solamente tiene un lugar donde puede aparecer en la cache, la cache se dice que es *de correspondencia directa (direct mapped).* La correspondencia es habitualmente (dirección de la estructura de bloque) módulo (número de bloques de la cache).
- Si un bloque se puede colocar en cualquier parte de la cache, la cache se dice que es *totalmente asociativa.*
- Si un bloque se puede colocar en un conjunto restringido de lugares de la cache, la cache se dice que es *asociativa por conjuntos.* Un *conjunto* es un grupo de dos o más bloques de la cache. Un bloque se hace corresponder primero a un conjunto y después el bloque se puede colocar en cualquier parte del conjunto. El conjunto se escoge habitualmente por selección de bits; es decir (dirección de la estructura de bloque) módulo (número de **conjuntos** en la cache). Si hay *n* bloques en un conjunto, la ubicación de la cache se dice *asociativa por conjuntos de n vías (asociatividad n).*

El rango de caches desde las de correspondencia directa a las totalmente asociativas es realmente un continuo de niveles de asociatividad por conjuntos: la cache de correspondencia directa es simplemente asociativa por conjuntos de una vía y la totalmente asociativa con *m* bloques podría denominarse aso-

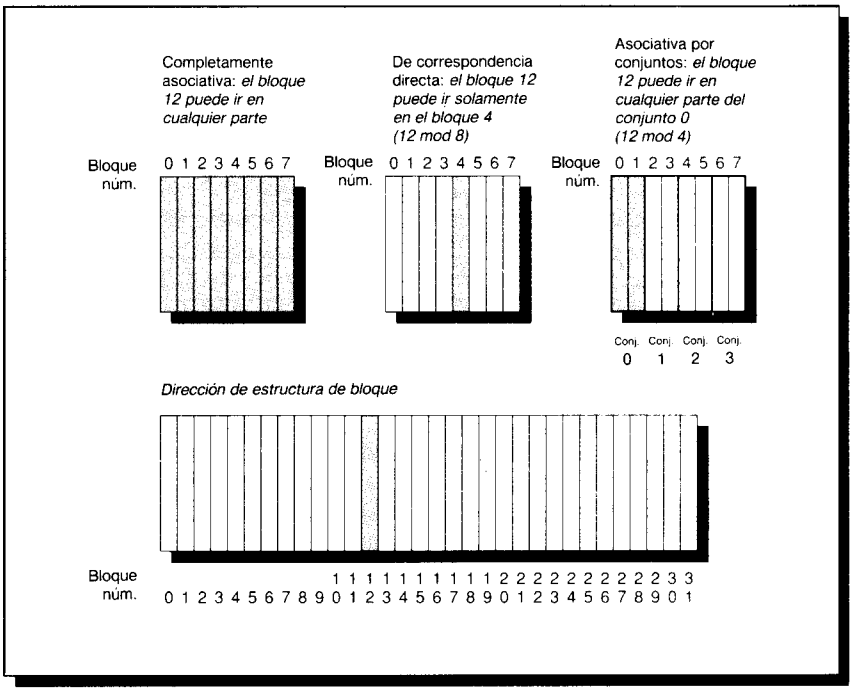


FIGURA 8.6 La cache tiene 8 bloques, mientras que la memoria tiene 32 bloques. La organización asociativa por conjuntos tiene 4 conjuntos de 2 bloques por conjunto, y se denomina asociativa por conjuntos de dos vías. (Las caches reales contienen cientos de bloques y las memorias reales contienen cientos de miles de bloques.) Suponer que no hay nada en la cache y que la dirección de la estructura del bloque en cuestión identifica el bloque 12 del nivel inferior. Las tres opciones para las caches se muestran de izquierda a derecha. En la completamente asociativa, el bloque 12 del nivel inferior puede ir a cualquiera de los 8 bloques de la cache. Con la correspondencia directa, el bloque 12 sólo se puede colocar en el bloque 4 (12 módulo 8). La asociativa por conjuntos, que tiene algunas características de ambas, permite que el bloque se coloque en cualquier parte del conjunto 0 (12 módulo 4). Con dos bloques por conjunto, esto significa que el bloque 12 se puede colocar bien en el bloque 0 o en el bloque 1 de la cache.

ciativa por conjuntos de m vías. La Figura 8.6 muestra dónde se puede colocar el bloque 12 en una cache de acuerdo con la política de ubicación de bloques.

P2. ¿Cómo se encuentra un bloque si está en la cache?

Las caches incluyen una etiqueta de dirección en cada bloque que identifica la dirección de la estructura del bloque. La etiqueta de cada bloque de cache que pueda contener la información deseada se comprueba para ver si coincide con la dirección de la estructura de bloque requerido por la CPU. La Figura 8.7 da un ejemplo. Como la velocidad es esencial, todas las posibles etiquetas se

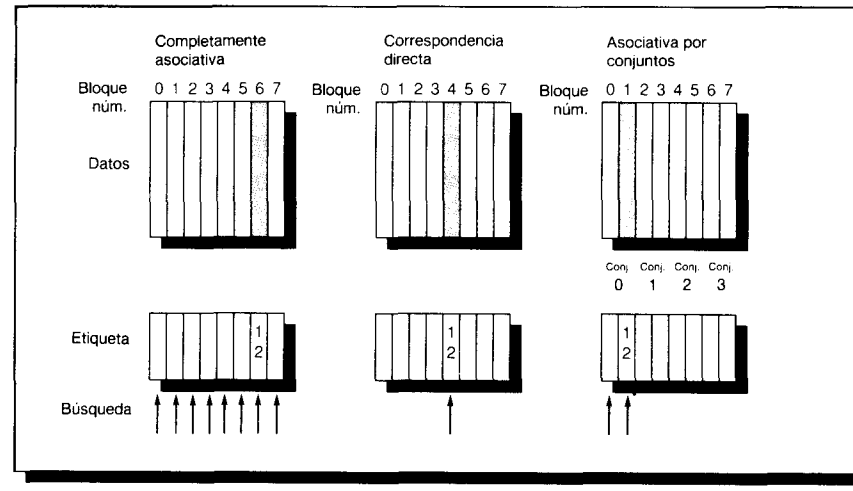


FIGURA 8.7 En la ubicación completamente asociativa, el bloque correspondiente a la dirección 12 de la estructura de bloque puede aparecer en cualquiera de los 8 bloques; por tanto, se deben buscar las 8 etiquetas. El dato deseado se encuentra en el bloque 6 de la cache en este ejemplo. En la ubicación de correspondencia directa sólo hay un bloque de la cache donde se puede encontrar el bloque 12 de memoria. En la ubicación asociativa por conjuntos, con 4 conjuntos, el bloque 12 de memoria debe estar en el conjunto 0 (12 mod 4); por tanto, se comprueban las etiquetas de los bloques de la cache 0 y 1. En este caso, el dato se encuentra en el bloque 1 de la cache. La velocidad de los accesos a la cache requiere que la búsqueda deba realizarse en paralelo para las correspondencias completamente asociativa y asociativa por conjuntos.

buscan en paralelo; la búsqueda serie podría hacer contraproductivo la asociatividad por conjuntos.

Debe haber una forma de saber que un bloque de cache no tiene información válida. El procedimiento más común es añadir un *bit de validez* a la etiqueta para que indique si esta entrada contiene, o no, una dirección válida. Si el bit no está a 1, no puede haber coincidencia en esta dirección.

Una omisión común al calcular el coste de las caches es olvidar el coste de la memoria de etiquetas. Para cada bloque se requiere una etiqueta. Una ventaja de incrementar el tamaño de los bloques es que el gasto por entrada de cache debido a las etiquetas llega a convertirse en una fracción más pequeña del coste total de la cache.

Antes de proceder con la siguiente pregunta, exploremos las relaciones entre una dirección de la CPU con la cache. La Figura 8.8 muestra cómo se divide una dirección en tres campos para encontrar datos en una cache asociativa por conjuntos: el campo de *desplazamiento de bloque* se utiliza para seleccionar el dato deseado del bloque; el campo de *índice* se utiliza para seleccionar el conjunto, y el campo de etiqueta para la comparación. Aunque la comparación podría hacerse sobre una parte mayor de la dirección que la de la etiqueta, no es necesario:

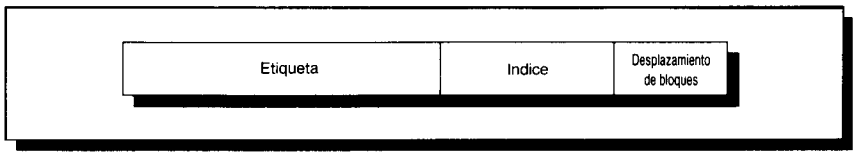


FIGURA 8.8 Las 3 partes de una dirección en una cache asociativa por conjuntos o de correspondencia directa. La etiqueta se utiliza para comprobar todos los bloques del conjunto y el índice se utiliza para seleccionar el conjunto. El desplazamiento de bloque es la dirección del dato deseado en el bloque.

- Comprobar el índice sería redundante, ya que se utilizó para seleccionar el conjunto que se va a comprobar (una dirección almacenada en el conjunto 0, por ejemplo, debe tener 0 en el campo de índice o no estaría almacenada en el conjunto 0).
- El desplazamiento es innecesario en la comparación porque todos los desplazamientos de bloque coinciden y el bloque completo está presente o no.

Si el tamaño total se mantiene igual, incrementando la asociatividad, se aumenta el número de bloques por conjunto, disminuyendo así el tamaño del índice e incrementando el tamaño de la etiqueta. Esto es, el límite etiqueta/índice de la Figura 8.8 se desplaza a la derecha al incrementar la asociatividad.

P3. ¿Qué bloque debe reemplazarse en un fallo de la cache?

Si la elección se hiciese entre un bloque que tiene un dato válido y un bloque que no lo tiene, entonces sería fácil seleccionar qué bloque reemplazar. Desafortunadamente, la frecuencia elevada de aciertos de la cache significa que la decisión hay que tomarla entre bloques que tengan datos válidos.

Un beneficio de la ubicación de correspondencia directa es que las decisiones hardware se simplifican. En efecto, es tan simple que no hay elección: sólo se comprueba un bloque para un acierto, y sólo se puede reemplazar ese bloque. Con la ubicación completamente asociativa o asociativa por conjuntos, hay varios bloques a elegir cuando se produce un fallo. Hay dos estrategias principales empleadas para seleccionar el bloque a reemplazar:

- *Aleatoria.* Para extender la ubicación uniformemente, se seleccionan aleatoriamente los bloques candidatos. Algunos sistemas utilizan un esquema para distribuir datos a través de un conjunto de bloques, de una manera pseudoaleatoria, para obtener un comportamiento reproducible, que es particularmente útil durante la depuración hardware.
- *Menos recientemente usado* (LRU). Para reducir la posibilidad de desechar información que se necesitará pronto, se registran los accesos a los bloques. El bloque sustituido es el que hace más tiempo que no es utilizado. Esto hace uso de un corolario de localidad temporal: si los bloques usados recientemente se usan probablemente de nuevo, entonces el mejor candidato

Direcciones de la estructura de bloque		3	2	1	0	0	2	3	1	3	0
Número de bloque LRU	0	0	0	0	3	3	3	1	0	0	2

FIGURA 8.9 Bloques menos recientemente usados (LRU) para una secuencia de direcciones de estructura del bloque en una jerarquía de memoria completamente asociativa. Esto supone que hay 4 bloques y que el comienzo el bloque LRU es el número 0. El número del bloque LRU se muestra debajo de cada nueva referencia de bloque. Otra política, primero en entrar/primero en salir, *First-in-first-out* (FIFO), sencillamente descarta el bloque que se utilizó N accesos a bloques diferentes, independiente de su patrón de referencia en las N-1 últimas referencias. El reemplazo aleatorio, generalmente, se comporta mejor que el FIFO y es más fácil de implementar.

Asociatividad:	**2 vías**		**4 vías**		**8 vías**	
Tamaño	**LRU**	**Aleatorio**	**LRU**	**Aleatorio**	**LRU**	**Aleatorio**
16 KB	5,18 %	5,69 %	4,67 %	5,29 %	4,39 %	4,96 %
64 KB	1,88 %	2,01 %	1,54 %	1,66 %	1,39 %	1,53 %
256 KB	1,15 %	1,17 %	1,13 %	1,13 %	1,12 %	1,12 %

FIGURA 8.10 Frecuencia de fallos comparando la estrategia LRU (menos recientemente usado) con el reemplazo aleatorio para algunos tamaños y asociatividades. Estos datos fueron coleccionados para un tamaño de bloque de 16 bytes utilizando una de las trazas VAX conteniendo el código del sistema operativo y de usuario (SAVE0). Esta traza se incluye en el suplemento software para uso del curso. Hay poca diferencia entre LRU y reemplazo aleatorio para caches de tamaño mayor en esta traza.

para ser eliminado es el menos usado recientemente. La Figura 8.9 muestra el bloque menos recientemente usado para una secuencia de direcciones de estructura de bloque en una jerarquía de memoria completamente asociativa.

Una virtud de la estrategia aleatoria es que es sencilla de construir en hardware. Cuando el número de bloques a gestionar aumenta, la estrategia LRU se convierte en enormemente cara y, frecuentemente, se utiliza sólo una aproximación de ésta. La Figura 8.10 muestra las diferencias en frecuencias de fallos entre la estrategia de reemplazo LRU y la de reemplazo aleatoria. La política de reemplazo juega un papel más importante en las caches más pequeñas que en las caches mayores donde hay más opciones para sustituir.

P4. ¿Qué ocurre en una escritura?

Las lecturas dominan los accesos a las caches. Todos los accesos de instrucciones son lecturas y muchas instrucciones no escriben en memoria. La Figura 4.34 sugiere una mezcla de un 9 por 100 de almacenamientos y un 17

por 100 de cargas para cuatro programas de DLX, de forma que las escrituras ocasionan menos del 10 por 100 del tráfico de memoria. Hacer el caso común rápido significa optimizar las caches para las lecturas, pero la Ley de Amdahl nos hace pensar que los diseños de alto rendimiento no pueden despreciar la velocidad de las escrituras.

Afortunadamente, el caso común también es el caso fácil para hacerlo rápido. El bloque se puede leer al mismo tiempo que se lee y compara la etiqueta, para que la lectura del bloque comience tan pronto como esté disponible la dirección de la estructura del bloque. Si la lectura es un acierto, el bloque pasa inmediatamente a la CPU. Si es un fallo, no hay beneficio —pero tampoco perjuicio.

Este no es el caso de las escrituras. El procesador especifica el tamaño de la escritura, habitualmente entre 1 y 8 bytes; sólo se puede cambiar esa porción de un bloque. En general esto significa una secuencia de operaciones de leer-modificar-escribir en el bloque: leer el bloque original, modificar una parte y escribir el nuevo valor del bloque. Sin embargo, la modificación de un bloque no puede comenzar hasta que se compruebe la etiqueta para ver si es un acierto. Como la comprobación de la etiqueta no puede realizarse en paralelo, entonces las escrituras, normalmente, duran más que las lecturas.

Por ello, son las políticas de escritura las que distinguen muchos diseños de cache. Hay dos opciones básicas para escribir en la cache:

- *Escritura directa* (o *almacenamiento directo*) *(Write through* o *store through).* La información se escribe en el bloque de la cache **y** en el bloque de la memoria de nivel inferior.
- *Postescritura (Write back* —también llamado *copy back* o *store in—).* La información se escribe sólo en el bloque de la cache. El bloque modificado de la cache se escribe en memoria principal sólo cuando es reemplazado.

Los bloques de cache de postescritura se denominan *limpios* o *modificados (clean* o *dirty),* dependiendo que la información de la cache difiera de la memoria de nivel inferior. Para reducir la frecuencia de postescrituras de bloques en el reemplazo, se utiliza comúnmente una característica denominada *bit de modificación (dirty bit).* Este bit de estado indica si el bloque se modificó o no en la cache. Si no lo fue, el bloque no es escrito, ya que el nivel inferior tiene la misma información que la cache.

Los dos tipos de escritura tienen sus ventajas. Con la postescritura, las escrituras se realizan a la velocidad de la memoria cache, y múltiples escrituras en un bloque requieren sólo una escritura en la memoria de nivel inferior. Como cada escritura no va a memoria, la postescritura utiliza menos ancho de banda de memoria, haciendo atractivo este tipo de escritura en multiprocesadores. Con la escritura directa, los fallos de lectura no ocasionan en las escrituras en el nivel inferior, y este tipo de escritura es más fácil de implementar que el anterior. La escritura directa también tiene la ventaja de que la memoria principal tiene la copia más reciente de los datos. Esto es importante en los multiprocesadores y en las E/S, que examinaremos en la Sección 8.8. Por consiguiente, los multiprocesadores quieren postescritura para reducir el tráfico de memoria por procesador y escritura directa para mantener la cache y memoria consistentes.

En la estrategia de escritura directa, cuando la CPU debe esperar la finalización de cada escritura antes de proceder con la siguiente operación, diremos que la CPU se detiene en las escrituras. Una optimización común para reducir las detenciones de escritura es un *buffer de escritura,* que permite al procesador continuar mientras se actualiza la memoria. Como veremos en la Sección 8.8, las detenciones de escritura pueden presentarse aun con buffers de escritura.

Hay dos opciones en un fallo de escritura:

- *Ubicar en escritura* (también denominado *búsqueda en escritura).* El bloque se carga, seguido de las acciones anteriores de acierto de escritura. Esto es similar a un fallo de lectura.
- *No ubicar en escritura* (también denominado *evitar escritura).* El bloque se modifica en el nivel inferior y no se carga en la cache.

Aunque cualquier política de fallo de escritura se pueda utilizar con la escritura directa o la postescritura, generalmente las caches de postescritura utilizan la ubicación de escritura (esperando que las escrituras posteriores en ese bloque sean capturadas por la cache) y caches de escritura directa, con frecuencia, utilizan no ubicación de escritura (ya que las escrituras posteriores a ese bloque deberán ir, en cualquier caso, a memoria).

Una cache ejemplo: la cache del VAX-11/780

Para dar contenido a estas ideas, la Figura 8.11 muestra la organización de la cache del VAX-11/780. La cache contiene 8 192 bytes de datos en bloques de 8 bytes con ubicación asociativa por conjuntos de dos vías, reemplazo aleatorio, escritura directa con un buffer de escritura de una palabra, y no ubicación de escritura en un fallo de escritura.

Sigamos los pasos de un acierto de cache tal como aparecen en la Figura 8.11. (Los cinco pasos se muestran con números inscritos en círculos.) La dirección que llega a la cache está dividida en dos campos: la dirección de la estructura del bloque de 29 bits y el desplazamiento del bloque de 3 bits. La dirección de la estructura del bloque, además, está dividida en etiqueta de dirección e índice de cache. El paso 1 muestra esta división.

El índice de la cache selecciona el conjunto que se va a examinar para ver si el bloque está en la cache. (Un conjunto es un bloque de cada banco de la Figura 8.11.) El tamaño del índice depende del tamaño de cache, tamaño del bloque y grado de asociatividad. En este caso, el índice resulta ser de 9 bits:

$$\frac{\text{Bloques}}{\text{Banco}} = \frac{\text{Tamaño de cache}}{\text{Tamaño de bloque} \cdot \text{Grado de asociatividad}} =$$

$$\frac{8\,192}{8 \cdot 2} = 512 = 2^9$$

En una cache asociativa por conjuntos de dos vías, el índice se envía a ambos bancos. Este es el paso 2.

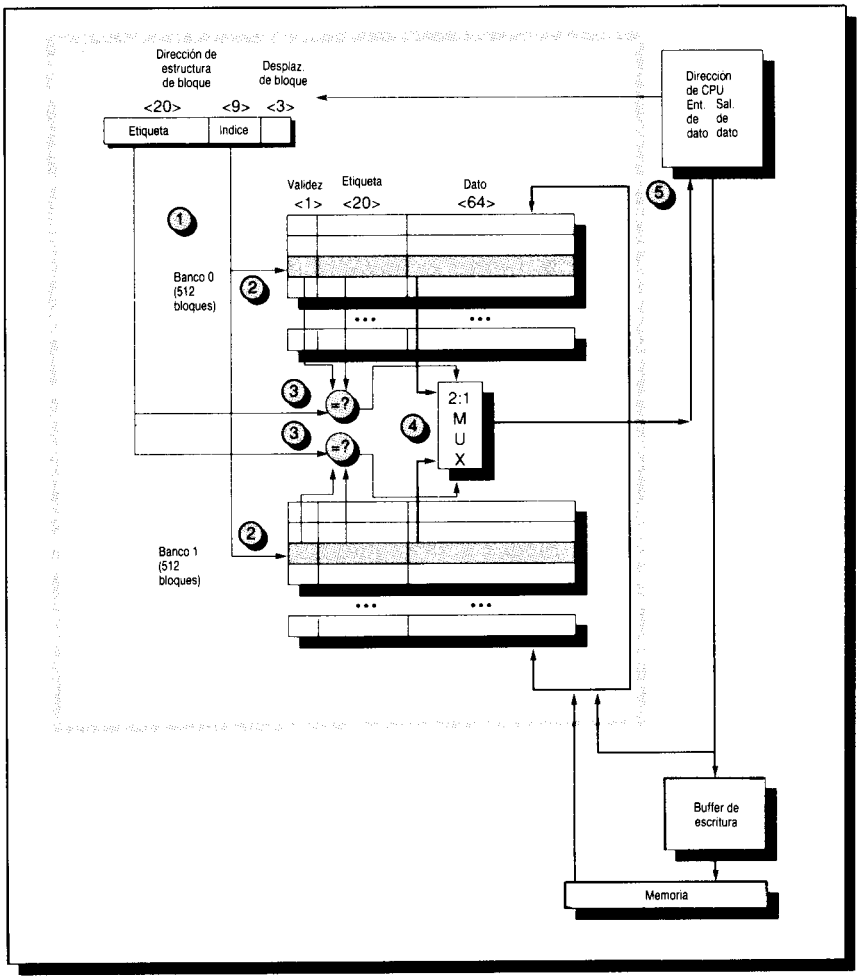


FIGURA 8.11 Organización de la cache del VAX-11/780. La cache de 8 KB es asociativa por conjuntos de dos vías con bloques de 8 bytes. Tiene 512 conjuntos con dos bloques por conjunto; el conjunto lo selecciona el índice de 9 bits. Los cinco pasos de un acierto de lectura, mostrados como números inscritos en un círculo en orden de ocurrencia, etiquetan esta organización. La línea de memoria a la cache se utiliza en un fallo para cargar la cache. La multiplexación, como se encuentra en el paso 4, no es necesaria en una cache de correspondencia directa. Observar que el desplazamiento se conecta al seleccionar el chip *(chip select)* de las SRAM de datos para permitir que las palabras adecuadas se envíen al multiplexor 2:1.

Después de leer una etiqueta de dirección de cada banco, la porción de la etiqueta de dirección de la estructura del bloque se compara con las etiquetas. Este es el paso 3 de la figura. Para estar seguro que la etiqueta contiene información válida, el bit de validez debe ser 1, o los resultados de la comparación se ignoran.

Suponiendo que coincide una de las etiquetas, un multiplexor 2:1 (paso 4) selecciona el bloque del conjunto coincidente. ¿Por qué no pueden coincidir ambas etiquetas? Es responsabilidad del algoritmo de reemplazo asegurar que una dirección aparezca en sólo un bloque. Para reducir el tiempo de acierto, los datos se leen al mismo tiempo que las etiquetas de dirección; por tanto, en el instante que el multiplexor del bloque está listo, los datos también están listos.

Este paso se necesita en caches asociativas por conjunto, pero se puede omitir en caches de correspondencia directa *(direct-mapped)*, ya que no hay que hacer selección. El multiplexor usado en este caso puede estar en el camino crítico de temporización, poniendo en peligro la duración del ciclo de reloj de la CPU. (El ejemplo de la pág. 451 y la falacia de la pág. 519 exploran el compromiso entre frecuencias más bajas de fallos y duración mayor del ciclo de reloj.)

En el paso final, la palabra se envía a la CPU. Los cinco pasos se llevan a cabo en un único ciclo de reloj de la CPU.

¿Qué ocurre en un fallo? La cache envía una señal de parada a la CPU indicándole que espere, y de memoria se leen dos palabras (ocho bytes). Eso necesita 6 ciclos de reloj en el VAX-11/780 (ignorando interferencias del bus). Cuando llegan los datos, la cache debe escoger un bloque para reemplazarlo; el VAX-11/780 selecciona uno de los dos bancos aleatoriamente. Sustituir un bloque significa actualizar los datos, la etiqueta de dirección y el bit de validez. Una vez que se hace esto, la cache lleva a cabo un ciclo regular de acierto y devuelve el dato a la CPU.

Las escrituras son más complicadas en el VAX-11/780, ya que pueden realizarse en cualquier memoria. Si la palabra que se va a escribir está en la cache, los cuatro primeros pasos son los mismos. El siguiente paso es escribir el dato en el bloque, después escribir la parte cambiada en la cache. El VAX-11/780 utiliza no ubicación en escritura. Consecuentemente, en un fallo de escritura la CPU escribe «evitando» la cache en la memoria del nivel inferior y no afecta a la cache.

Como es una cache de escritura directa, el proceso no finaliza todavía. La palabra también se envía a un buffer de escritura de una palabra. Si el buffer está vacío, se escriben en él la palabra y dirección completa y se ha terminado. La CPU continúa trabajando mientras el buffer de escritura escribe la palabra en memoria. Si el buffer está lleno, la cache (y CPU) deben esperar hasta que esté vacío.

Rendimiento de la cache

El tiempo de CPU se puede dividir en los ciclos de reloj que la CPU emplea en la ejecución del programa y en los ciclos de reloj que la CPU emplea esperando al sistema de memoria. Por ello,

$$\begin{aligned}\text{Tiempo de CPU} = (&\text{Ciclos de reloj de ejecución-CPU} + \\ +\ &\text{Ciclos de reloj de detención-memoria}) \cdot \text{Duración ciclo de reloj}\end{aligned}$$

Para simplificar la evaluación de alternativas cache, a veces los diseñadores suponen que todas las detenciones de memoria se deben a la cache. Esto

es cierto para muchas máquinas; en máquinas donde no es cierto, la cache todavía domina las detenciones que no se deben exclusivamente a la cache. Utilizamos aquí esta hipótesis simplificadora, ¡pero es importante tener en cuenta **todas** las detenciones de memoria cuando se calcule el rendimiento final!

La fórmula anterior plantea la pregunta de si los ciclos de reloj para un acceso a cache se deben considerar parte de los ciclos de reloj de ejecución-CPU o parte de los ciclos de reloj de detención de memoria. Aunque cualquier convenio es defendible, el más ampliamente aceptado es incluir los ciclos de reloj de los aciertos como ciclos de reloj de ejecución CPU.

Los ciclos de reloj de detención de memoria se pueden definir entonces en función del número de accesos a memoria por programa, penalización de fallos (en ciclos de reloj), y frecuencia de fallos para lecturas y escrituras:

$$\text{Ciclos de reloj de detención de memoria} =$$

$$\frac{\text{Lecturas}}{\text{Programa}} \cdot \text{Frecuencia fallos lectura} \cdot$$

$$\cdot \text{Penalización de fallos de lectura} + \frac{\text{Escrituras}}{\text{Programa}} \cdot$$

$$\cdot \text{Frecuencia de fallos de escritura} \cdot \text{Penalización de fallos de escritura}$$

Simplificamos la fórmula completa combinando juntas lecturas y escrituras:

$$\text{Ciclos de reloj de detención de memoria} =$$

$$= \frac{\text{Acceso a memoria}}{\text{Programa}} \cdot \text{Frecuencia de fallos} \cdot \text{Penalización de fallos}$$

Factorizando el recuento de instrucciones (IC) del tiempo de ejecución y ciclos de detención de memoria, obtenemos ahora una fórmula de tiempo-CPU que incluye accesos a memoria por instrucción, frecuencia de fallos y penalización de fallos:

$$\text{Tiempo de CPU} = \text{IC} \cdot \left(\text{CPI}_{\text{ejecución}} + \frac{\text{Accesos a memoria}}{\text{Instrucción}} \cdot \right.$$

$$\left. \cdot \text{Frecuencia de fallos} \cdot \text{Penalización de fallos} \right) \cdot \text{Tiempo de ciclo de reloj}$$

Algunos diseñadores prefieren medir la frecuencia de fallos como *fallos por instrucción* en lugar de fallos por referencia a memoria:

$$\frac{\text{Fallos}}{\text{Instrucción}} = \frac{\text{Accesos a memoria}}{\text{Instrucción}} \cdot \text{Frecuencia de fallos}$$

La ventaja de esta medida es que es independiente de la implementación hardware. Por ejemplo, la unidad de instrucciones del VAX-11/780 puede hacer repetidas referencias a un único byte (ver Sección 8.7), que puede reducir artificialmente la frecuencia de fallos si se midiese como fallos por referencias a memoria en lugar de por instrucción ejecutada. El inconveniente es que esta medida es dependiente de la arquitectura, por ello es más popular con arquitectos que trabajan con una sola familia de computadores. Entonces utilizan esta versión de la fórmula del tiempo de CPU:

$$\text{Tiempo de CPU} = \text{IC} \cdot$$

$$\cdot \left(\text{CPI}_{\text{ejecución}} + \frac{\text{Fallos}}{\text{Instrucción}} \cdot \text{Penalización de fallos} \right) \cdot$$

$$\cdot \text{ Tiempo de ciclo de reloj}$$

Ahora podemos explorar las consecuencias de las caches en el rendimiento.

Ejemplo

Usemos el VAX-11/780 como primer ejemplo. La penalización de fallos de cache es de 6 ciclos de reloj, y todas las instrucciones normalmente emplean 8,5 ciclos de reloj (ignorando detenciones de memoria). Suponer que la frecuencia de fallos es del 11 por 100, y que hay una media de 3,0 referencias a memoria por instrucción. ¿Cuál es el impacto en el rendimiento cuando se incluye el comportamiento de la cache?

Respuesta

$$\text{Tiempo de CPU} =$$

$$= \text{IC} \cdot \left(\text{CPI}_{\text{ejecución}} + \frac{\text{Ciclos de reloj de detención de memoria}}{\text{Instrucción}} \right) \cdot$$

$$\cdot \text{ Tiempo de ciclo de reloj}$$

El rendimiento, incluyendo los fallos de la cache, es

$$\text{Tiempo de CPU}_{\text{con cache}} = \text{IC} \cdot (8{,}5 + 3{,}0 \cdot 11\,\% \cdot 6) \cdot$$

$$\text{Tiempo de ciclo de reloj} = \text{Recuento de instrucciones} \cdot 10{,}5 \cdot \text{Tiempo de ciclo de reloj}$$

La duración del ciclo de reloj y el recuento de instrucciones son iguales, con o sin cache, así que el tiempo de CPU incrementa con el CPI de 8,5 a 10,5. Por consiguiente, el impacto de la jerarquía de memoria es alargar el tiempo de CPU un 24 por 100.

Ejemplo

Ahora, calculemos el impacto en el rendimiento cuando el comportamiento de la cache se tiene en cuenta en una máquina con un CPI más bajo. Suponer que la penalización de fallos de la cache es de 10 ciclos de reloj y, en promedio, las instrucciones emplean 1,5 ciclos de reloj; la frecuencia de fallos es del 11 por 100, y hay un promedio de 1,4 referencias de memoria por instrucción.

Respuesta

$$\text{Tiempo de CPU} =$$

$$= \text{IC} \cdot \left(\text{CPI}_{\text{ejecución}} + \frac{\text{Ciclos de reloj de detención de memoria}}{\text{Instrucción}} \right) \cdot$$

$$\cdot \text{ Tiempo de ciclo de reloj}$$

Haciendo las mismas hipótesis que en el ejemplo anterior sobre aciertos de cache, el rendimiento, incluyendo los fallos de caches, es

$$\text{Tiempo de CPU}_{\text{con cache}} = \text{IC} \cdot (1{,}5 + 1{,}4 \cdot 11\,\% \cdot 10) \cdot$$

$$\text{Tiempo de ciclo de reloj} = \text{Recuento de instrucciones} \cdot 3{,}0 \cdot \text{Tiempo de ciclo de reloj}$$

La duración del ciclo de reloj y el recuento de instrucciones son iguales, con o sin cache; así, el tiempo de CPU aumenta con el CPI de 1,5 a 3,0. La inclusión del comportamiento de la cache duplica el tiempo de ejecución.

Como ilustran estos ejemplos, las penalizaciones de comportamiento de la cache varían de significativas a enormes. Además, los fallos de la cache tienen un doble impacto en una CPU con CPI bajo y un reloj rápido:

1. A más bajo CPI, el impacto es más pronunciado.
2. Independientemente de la CPU, las memorias principales tienen tiempos similares de acceso a memoria, ya que se construyen con los mismos chips de memoria. Cuando se calcula el CPI, la penalización de fallos de la cache se mide en ciclos de reloj de **CPU** necesarios para un fallo. Por tanto, una mayor frecuencia de reloj de CPU lleva a una mayor penalización de los fallos, aun cuando las memorias principales sean de la misma velocidad.

La importancia de la cache para una CPU con bajo CPI y altas frecuencias de reloj es entonces grande y, consiguientemente, mayor es el peligro de despreciar el comportamiento de la cache al valorar el rendimiento de dichas máquinas.

Aunque minimizar el tiempo medio de acceso a memoria es un objetivo razonable y lo utilizaremos mucho en este capítulo, tener en cuenta que el objetivo final es reducir el tiempo de ejecución de la CPU.

Ejemplo

¿Cuál es el impacto de dos organizaciones cache diferentes en el rendimiento de una CPU? Suponer que el CPI es normalmente 1,5, con una duración del ciclo de reloj de 20 ns; que hay 1,3 referencias a memoria por instrucción y que el tamaño de ambas caches es de 64 KB. Una cache es de correspondencia directa y la otra es asociativa por conjuntos de dos vías. Como la velocidad de la CPU está ligada directamente a la velocidad de las caches, suponer que la duración del ciclo de reloj de CPU debe alargarse un 8,5 por 100 para acomodar el multiplexor de selección de la cache asociativa por conjuntos (paso 4 de la Fig. 8.11). Para la primera aproximación, la penalización de fallos es de 200 ns para cualquier organización cache. (En la práctica, se debe redondear hacia arriba o hacia abajo un número entero de ciclos de reloj.) Primero, calcular el tiempo medio de acceso a memoria y, después, el rendimiento de la CPU.

Respuesta

La Figura 8.12 muestra que la frecuencia de fallos de una cache de correspondencia directa de 64 KB es del 3,9 por 100 y que la frecuencia de fallos para una cache asociativa por conjuntos de dos vías del mismo tamaño es del 3,0 por 100. El tiempo medio de acceso a memoria es

$$\text{Tiempo medio acceso-memoria} = \text{Tiempo aciertos} + \\ + \text{Frecuencia de fallos} \cdot \text{Penalización fallos}$$

Por tanto, el tiempo para cada organización es

$$\text{Tiempo medio-acceso a memoria}_{\text{1-vía}} = 20 + 0{,}039 \cdot 200 = 27{,}8 \text{ ns}$$

$$\text{Tiempo medio-acceso a memoria}_{\text{2-vía}} = 20 \cdot 1{,}085 + 0{,}030 \cdot 200 = 27{,}7 \text{ ns}$$

El tiempo medio de acceso a memoria es mejor para la cache asociativa por conjuntos de dos vías.

El rendimiento de la CPU es

$$\text{Tiempo de CPU} = \text{IC} \cdot$$

$$\cdot \left(\text{CPI}_{\text{ejecución}} + \frac{\text{Fallos}}{\text{Instrucción}} \cdot \text{Penalización de fallos} \right) \cdot$$

$$\cdot \text{Tiempo de ciclo de reloj} = \text{IC} \cdot \left(\text{CPI}_{\text{ejecución}} \cdot \text{Tiempo de ciclo de reloj} + \right.$$

$$\frac{\text{Accesos a memoria}}{\text{Instrucción}} \cdot \text{Frecuencia de fallos} \cdot \text{Penalización de fallos} \cdot$$

$$\left. \cdot \text{Tiempo de ciclo de reloj} \right)$$

Sustituyendo 200 ns por (penalización de fallos · duración de ciclo reloj), el rendimiento de cada organización cache es

$$\text{Tiempo de CPU}_{1\text{-vía}} = \text{IC} \cdot (1{,}5 \cdot 20 + 1{,}3 \cdot 0{,}039 \cdot 200) = 40{,}1 \cdot \text{IC}$$
$$\text{Tiempo de CPU}_{2\text{-vía}} = \text{IC} \cdot (1{,}5 \cdot 20 \cdot 1{,}805 + 1{,}3 \cdot 0{,}030 \cdot 200) = 40{,}4 \cdot \text{IC}$$

y el rendimiento relativo es

$$\frac{\text{Tiempo de CPU}_{2\text{-vía}}}{\text{Tiempo de CPU}_{1\text{-vía}}} = \frac{40{,}4 \cdot \text{Recuento de instrucciones}}{40{,}1 \cdot \text{Recuento de instrucciones}}$$

En contraste con los resultados de los tiempos medios de acceso, la cache de correspondencia directa conduce a un rendimiento ligeramente mejor. Como el tiempo de CPU es nuestra evaluación de base (y la cache de correspondencia directa es más simple de construir), en este ejemplo la cache preferida es la de correspondencia directa. (Ver la pifia de la pág. 519 para más detalles sobre este tipo de cuestiones.)

Las tres fuentes de fallos de la cache: forzosos, capacidad y conflictos

Un modelo intuitivo de comportamiento de la cache atribuye todos los fallos a una de tres fuentes:

- *Forzosos.* El primer acceso a un bloque no está en la cache; así que el bloque debe ser traído a la cache. Estos también se denominan *fallos de arranque frío* o *fallos de primera referencia.*
- *Capacidad.* Si la cache no puede contener todos los bloques necesarios durante la ejecución de un programa, se presentarán fallos de capacidad debido a los bloques que se descartan y posteriormente se recuperan.
- *Conflicto.* Si la estrategia de ubicación de bloques es asociativa por conjuntos o de correspondencia directa, los fallos de conflicto (además de los forzosos y de capacidad), ocurrirán, ya que se puede descartar un bloque y posteriormente recuperarlo si a un conjunto le corresponden demasiados bloques. Estos fallos también se denominan *fallos de colisión.*

La Figura 8.12 muestra la frecuencia relativa de los fallos de la cache, descompuestas por las «tres C» [*Compulsory* (forzoso), *Capacity* (capacidad), *Conflict* (conflicto)]. Para mostrar el beneficio de la asociatividad, los fallos de conflicto se dividen en fallos causados por cada disminución de la asociatividad. Las categorías están etiquetadas con *n*-vías, significando los fallos causados al ir al nivel más bajo de asociatividad a partir del inmediatamente superior. Aquí están las cuatro categorías:

Tamaño cache	Grado asociatividad	Frecuencia total fallos	Componentes de la frecuencia de fallos (porcentaje relativo) *(suma = 100 % de frecuencia total de fallos)*					
			Forzoso		Capacidad		Conflicto	
1 KB	1 vía	0,191	0,009	5 %	0,141	73 %	0,042	22 %
1 KB	2 vías	0,161	0,009	6 %	0,141	87 %	0,012	7 %
1 KB	4 vías	0,152	0,009	6 %	0,141	92 %	0,003	2 %
1 KB	8 vías	0,149	0,009	6 %	0,141	94 %	0,000	0 %
2 KB	1 vía	0,148	0,009	6 %	0,103	70 %	0,036	24 %
2 KB	2 vías	0,122	0,009	7 %	0,103	84 %	0,010	8 %
2 KB	4 vías	0,115	0,009	8 %	0,103	90 %	0,003	2 %
2 KB	8 vías	0,113	0,009	8 %	0,103	91 %	0,001	1 %
4 KB	1 vía	0,109	0,009	8 %	0,073	67 %	0,027	25 %
4 KB	2 vías	0,095	0,009	9 %	0,073	77 %	0,013	14 %
4 KB	4 vías	0,087	0,009	10 %	0,073	84 %	0,005	6 %
4 KB	8 vías	0,084	0,009	11 %	0,073	87 %	0,002	3 %
8 KB	1 vía	0,087	0,009	10 %	0,052	60 %	0,026	30 %
8 KB	2 vías	0,069	0,009	13 %	0,052	75 %	0,008	12 %
8 KB	4 vías	0,065	0,009	14 %	0,052	80 %	0,004	6 %
8 KB	8 vías	0,063	0,009	14 %	0,052	83 %	0,002	3 %
16 KB	1 vía	0,066	0,009	14 %	0,038	57 %	0,019	29 %
16 KB	2 vías	0,054	0,009	17 %	0,038	70 %	0,007	13 %
16 KB	4 vías	0,049	0,009	18 %	0,038	76 %	0,003	6 %
16 KB	8 vías	0,048	0,009	19 %	0,038	78 %	0,001	3 %
32 KB	1 vía	0,050	0,009	18 %	0,028	55 %	0,013	27 %
32 KB	2 vías	0,041	0,009	22 %	0,028	68 %	0,004	11 %
32 KB	4 vías	0,038	0,009	23 %	0,028	73 %	0,001	4 %
32 KB	8 vías	0,038	0,009	24 %	0,028	74 %	0,001	2 %
64 KB	1 vía	0,039	0,009	23 %	0,019	50 %	0,011	27 %
64 KB	2 vía	0,030	0,009	30 %	0,019	65 %	0,002	5 %
64 KB	4 vía	0,028	0,009	32 %	0,019	68 %	0,000	0 %
64 KB	8 vía	0,028	0,009	32 %	0,019	68 %	0,000	0 %
128 KB	1 vía	0,026	0,009	34 %	0,004	16 %	0,013	50 %
128 KB	2 vías	0,020	0,009	46 %	0,004	21 %	0,006	33 %
128 KB	4 vías	0,016	0,009	55 %	0,004	25 %	0,003	20 %
128 KB	8 vías	0,015	0,009	59 %	0,004	27 %	0,002	14 %

FIGURA 8.12 Frecuencia total de fallos para cada tamaño de cache y porcentaje de cada fallo de acuerdo con las «tres C» [Compulsory *(forzoso)*, Capacity *(capacidad)*, Conflict *(conflicto)*]. Los fallos forzosos son independientes del tamaño de la cache, mientras que los fallos de capacidad decrecen cuando la capacidad aumenta. Hill [1987] midió esta traza utilizando bloques de 32 bytes y reemplazo LRU. La generó en un VAX-11 corriendo Ultrix mezclando trazas de tres sistemas, utilizando una carga de trabajo de multiprogramación y tres trazas de usuarios. La longitud total fue del orden de un millón de direcciones; la parte mayor de datos referenciados durante la traza fue 221 KB. La Figura 8.13 muestra gráficamente la misma información. Observar que la regla empírica de cache 2:1 (en la primera cubierta) está soportada por las estadísticas de esta tabla: una cache de correspondencia directa de tamaño N tiene aproximadamente la misma frecuencia de fallos que una cache asociativa por conjuntos de 2 vías de tamaño N/2.

8-vías: de completamente asociativa (no conflictos) a asociativa de 8-vías

4-vías: de asociativa de 8-vías a asociativa de 4-vías

2-vías: de asociativa de 4-vías a asociativa de 2-vías

1-vía: de asociativa de 2-vías a asociativa de 1-vía (correspondencia directa)

La Figura 8.13 presenta los mismos datos gráficamente. El gráfico superior muestra frecuencias absolutas de fallos: el inferior dibuja el porcentaje de todos los fallos por tamaño de cache.

Habiendo identificado las tres C, ¿qué puede hacer un diseñador de computadores con respecto a ellas? Conceptualmente, los conflictos son los más fáciles: la ubicación completamente asociativa evita todos los fallos de conflicto. Sin embargo, la asociatividad es cara en hardware y puede ralentizar el tiempo de acceso (ver el ejemplo anterior o la segunda falacia de la Sección 8.10), llevando a una disminución del rendimiento global. Hay poco que hacer sobre la capacidad excepto comprar mayores chips de memoria. Si la memoria de nivel superior es mucho más pequeña de la que se necesita para un programa, y se emplea un porcentaje significativo del tiempo en transferir datos entre los dos niveles de la jerarquía, la jerarquía de memoria se dice castigada *(thrash)*. Ya que se requieren muchos reemplazos, castigo *(thrashing)* significa que la máquina corre próxima a la velocidad de la memoria de nivel más bajo, o quizá aún más lenta debido a la sobrecarga de fallos. Hacer bloques mayores reduce el número de fallos forzosos, pero puede incrementar los fallos de conflicto.

Las tres «C» dan una visión de las causas de los fallos, pero este modelo simple tiene sus límites. Por ejemplo, incrementar el tamaño de cache reduce los fallos de conflicto así como los de capacidad, ya que una cache mayor dispersa las referencias. Por tanto, un fallo puede ir de una categoría a otra cuando cambian los parámetros. Las tres C ignoran la política de reemplazo, ya que es difícil de modelar y, en general, es de menor significado. En circunstancias específicas la política de reemplazo realmente puede llevar a comportamientos anómalos, como frecuencias más pobres de fallos para mayor asociatividad, lo que es directamente contradictorio con el modelo de las tres C.

Elecciones para tamaños de bloque en caches

Las Figuras 8.3 y 8.4 mostraban en abstracto la relación del tamaño de bloque con la frecuencia de fallos y tiempos de acceso a memoria. Las Figuras 8.14 y 8.15 muestran los números específicos para un conjunto de programas y tamaños de cache. Tamaños de bloque mayores reducen los fallos forzosos, como sugiere el principio de localidad espacial. Al mismo tiempo, bloques mayores también reducen el número de bloques de la cache, incrementando los fallos de conflicto.

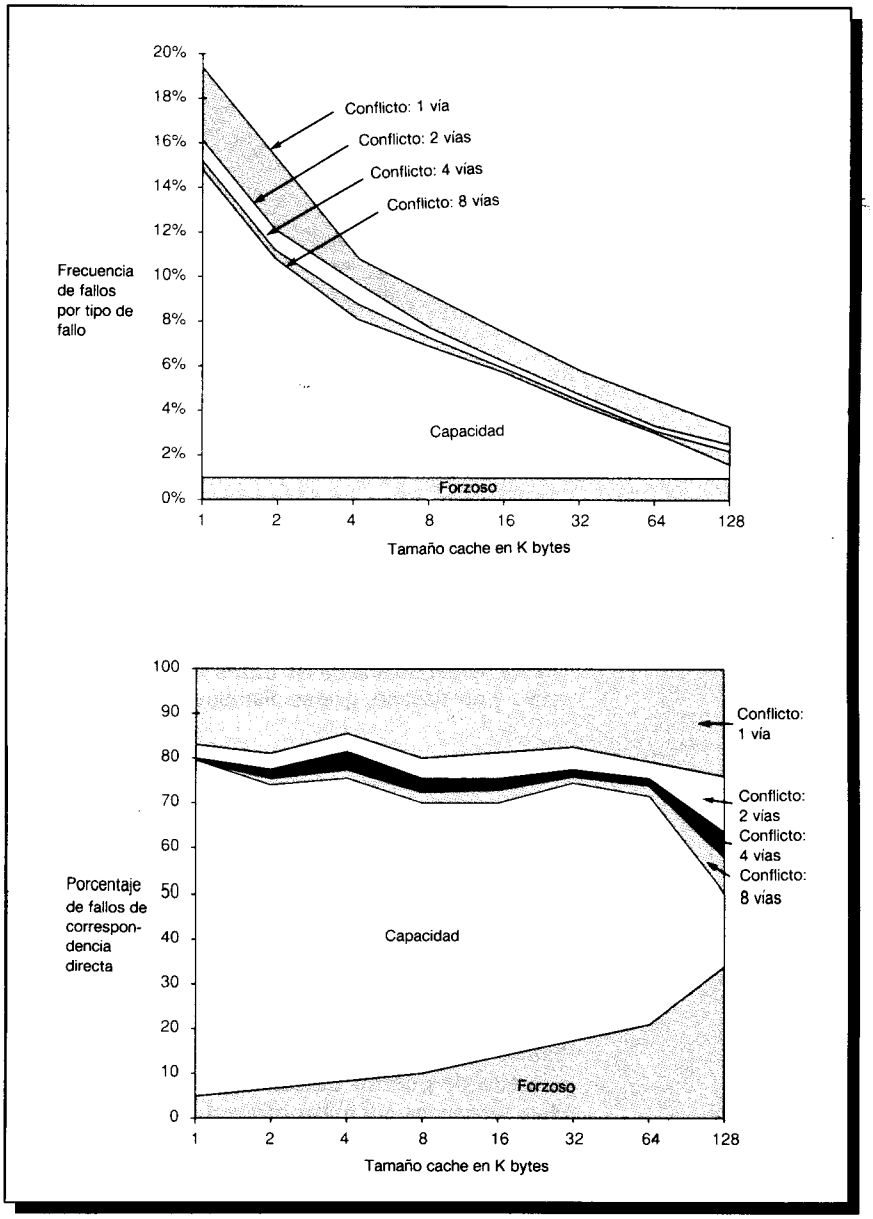


FIGURA 8.13 Frecuencia total de fallos (superior) y distribución de la frecuencia de fallos (inferior) para cada tamaño de cache de acuerdo con las tres C para los datos de la Figura 8.12. El diagrama superior es la frecuencia real de fallos, mientras que el inferior está escalado respecto a la frecuencia de fallos de correspondencia directa.

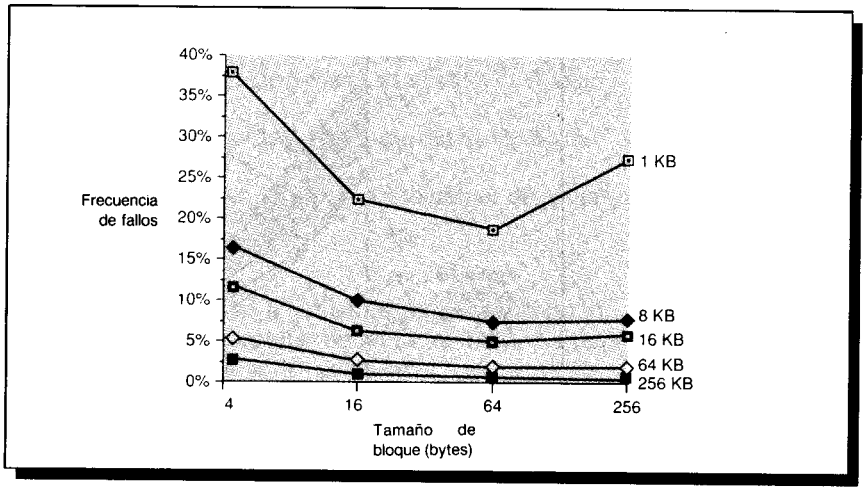


FIGURA 8.14 Frecuencia de fallos frente a tamaño de bloque. Observar que para una cache de 1 KB, los bloques de 256 bytes tiene una frecuencia de fallos mayor que los bloques de 16 o 64 bytes. (El bloque más pequeño es de 4 bytes.) En este ejemplo particular, la cache debería haber sido de 256 KB con el fin de que al incrementar el tamaño de bloque siempre dé como resultado una disminución de fallos. Estos datos se obtuvieron para una cache de correspondencia directa utilizando una de las trazas VAX que contiene los códigos del sistema operativo y del usuario, que se distribuyen con este libro (SAVE0).

Caches de sólo instrucciones o sólo datos frente a caches unificadas

De forma distinta a otros niveles de la jerarquía de memoria, las caches a veces se dividen en caches de instrucciones y caches de datos. Las caches que pueden contener instrucciones o datos son caches *unificadas* o caches *mixtas.* La CPU sabe si está emitiendo la dirección de una instrucción o de un dato; así, puede haber puertos separados para ambos, casi doblando el ancho de banda entre la cache y la CPU. (La Sección 6.4 del Capítulo 6 muestra las ventajas de puertos duales de memoria para ejecución segmentada.) Las caches separadas ademásn ofrecen la oportunidad de optimizar cada cache separadamente: diferentes capacidades, tamaños de bloque y asociatividades pueden conducir a un mejor rendimiento. Dividir entonces afecta al coste y rendimiento más allá de lo que está indicado por el cambio de las frecuencias de fallos. Ahora, limitamos nuestra discusión a este punto simplemente para mostrar cómo la frecuencia de fallos para instrucciones difiere de la frecuencia de fallos para datos.

La Figura 8.16 muestra cómo las caches de instrucciones tienen menor frecuencia de fallos que las caches de datos. La separación de instrucciones y datos elimina fallos debidos a conflictos entre los bloques de instrucciones y los bloques de datos, pero, al dividir, también se fija el espacio de cache dedicado a cada tipo. Una buena comparación de caches separadas de datos e ins-

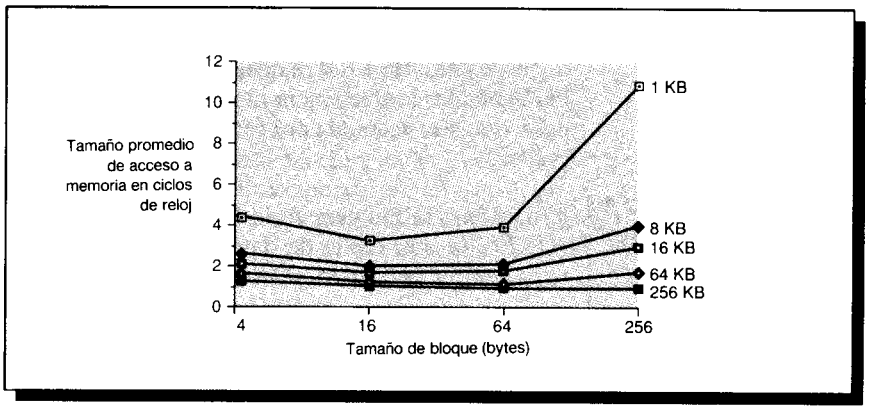


FIGURA 8.15 Tiempo medio de acceso frente a tamaño de bloque utilizando las frecuencias de fallos de la Figura 8.14. Se supone una latencia de 8 ciclos de reloj y que la memoria y bus pueden transferir 4 bytes por ciclo de reloj. En un fallo el bloque completo se carga en la cache antes de que la palabra requerida se envie a la CPU. El tiempo medio de acceso a memoria más bajo es o para bloques de 16 bytes o de 64 bytes, y los bloques de 256 bytes son mejores que los bloques de 4 bytes sólo para la cache mayor.

Tamaño	Instrucción sólo	Sólo datos	Unificada
0,25 KB	22,2 %	26,8 %	28,6 %
0,50 KB	17,9 %	20,9 %	23,9 %
1 KB	14,3 %	16,0 %	19,0 %
2 KB	11,6 %	11,8 %	14,9 %
4 KB	8,6 %	8,7 %	11,2 %
8 KB	5,8 %	6,8 %	8,3 %
16 KB	3,6 %	5,3 %	5,9 %
32 KB	2,2 %	4,0 %	4,3 %
64 KB	1,4 %	2,8 %	2,9 %
128 KB	1,0 %	2,1 %	1,9 %
256 KB	0,9 %	1,9 %	1,6 %

FIGURA 8.16 Frecuencia de fallos para caches de distintos tamaños de sólo datos, sólo instrucciones, y unificadas. Los datos son para una cache asociativa de 2 vías utilizando reemplazo LRU con bloques de 16 bytes para un promedio de trazas de usuario/sistema en la VAX-11 y trazas de sistema en el IBM 370 [Hill 1987]. El porcentaje de referencias a instrucciones en estas trazas es aproximadamente del 53 por 100.

trucciones con las caches unificadas requiere que el tamaño total de la cache sea el mismo. Por tanto, una cache separada de instrucciones de 1 KB y una cache de datos de 1 KB se deberán comparar con una cache unificada de 2 KB. Para calcular la frecuencia media de fallos en caches separadas de instrucciones y datos se necesita conocer el porcentaje de referencias a memoria en cada cache.

Ejemplo

¿Cuál tiene la frecuencia de fallos más baja: una cache de instrucciones de 16 KB con una cache de datos de 16 KB o una cache unificada de 32 KB? Suponer que el 53 por 100 de las referencias son instrucciones.

Respuesta

Como se estableció en la leyenda de la Figura 8.16, el 53 por 100 de los accesos a memoria son referencias a instrucciones. Por tanto, la frecuencia global de fallos para las caches divididas es

$$53\,\% \cdot 3{,}6\,\% + 47\,\% \cdot 5{,}3\,\% = 4{,}4\,\%$$

Una cache unificada de 32 KB tiene una frecuencia de fallos ligeramente más baja, de 4,3 por 100.

8.4 Memoria principal

...el desarrollo único que puso a los computadores en sus cimientos fue la invención de una forma fiable de memoria de núcleos... Su coste era razonable, era fiable y, como era fiable, en el transcurso del tiempo pudo aumentarse su tamaño.

Maurice Wilkes, *Memoirs of a Computer Pioneer (Memorias de un pionero de los computadores)*, (1985, pág. 209)

Suponiendo que sólo haya un nivel de cache, la memoria principal es el siguiente nivel en la jerarquía. La memoria principal satisface las demandas de las caches y unidades vectoriales, y sirve como interfaz de E/S, ya que es el destino de la entrada, así como la fuente para la salida. De forma distinta a las caches, las medidas de rendimiento de la memoria principal hacen énfasis en la latencia y ancho de banda. Generalmente, la latencia de la memoria principal (que afecta a la penalización de fallos de la cache) es la preocupación primordial de la cache, mientras que el ancho de banda de la memoria principal es la preocupación primordial de las E/S y unidades vectoriales. Cuando los bloques de cache crecen desde 4-8 bytes a 64-256 bytes, el ancho de banda de la memoria principal también llega a ser importante para las caches. La relación entre memoria principal y E/S se explica en el Capítulo 9.

La latencia de memoria tradicionalmente se expresa utilizando dos medidas: tiempo de acceso y duración del ciclo. El *tiempo de acceso* es el tiempo desde que se pide una lectura hasta que llega la palabra deseada, mientras que la *duración del ciclo* es el tiempo mínimo entre peticiones a memoria. En los años setenta, cuando las DRAM crecían en capacidad, el coste de un empaquetamiento con todas las líneas de dirección necesarias llegó a ser un pro-

blema. La solución fue multiplexar las líneas de dirección, reduciendo así el número de patillas de dirección a la mitad. La mitad superior de la dirección viene primero, durante un *strobe de acceso a filas* o RAS *(row access strobe).* Esto es seguido por la segunda parte de la dirección durante el *strobe de acceso a columnas* o CAS *(column-access strobe).* Estos nombres provienen de la organización interna del chip, ya que la memoria está organizada como una matriz rectangular direccionada por filas y columnas.

Un requerimiento adicional de las DRAM proviene de la propiedad que indica su primera letra, D, de dinámica. Cada fila de una DRAM debe ser accedida en un cierto intervalo de tiempo, como por ejemplo 2 milisegundos, o la información de la DRAM puede perderse. Este requerimiento significa que el sistema de memoria ocasionalmente no está disponible porque está enviando una señal para refrescar cada chip. El coste de un refresco es normalmente un acceso completo a memoria (RAS y CAS) para cada fila de la DRAM. Como la matriz de memoria de una DRAM probablemente es cuadrada, el número de pasos en un refresco es habitualmente la raíz cuadrada de la capacidad de la DRAM.

En contraste con las DRAM, están las SRAM —la primera letra significa «estática» *(static)*—. La naturaleza dinámica de los circuitos para la DRAM requiere que los datos se vuelvan a escribir después de ser leídos, de aquí la diferencia entre el tiempo de acceso y la duración del ciclo, y también la necesidad de refresco. Las SRAM utilizan más circuitos por bit para prevenir que se distorsione la información cuando se lea. Por ello, de forma distinta a las DRAM, no hay diferencia entre tiempo de acceso y duración de ciclo y no hay necesidad de refrescar las SRAM. En los diseños con DRAM el énfasis está en la capacidad, mientras que los diseños con SRAM están relacionados con la capacidad y velocidad. (Por estos motivos, las líneas de dirección de las SRAM no están multiplexadas.) Para memorias diseñadas con tecnologías comparables, la capacidad de las DRAM es aproximadamente 16 veces la de las SRAM, y el tiempo del ciclo de las SRAM es de 8 a 16 veces más rápido que las DRAM.

La memoria principal de prácticamente todos los computadores vendidos en la última década está compuesta por DRAM semiconductoras (y prácticamente todas las caches utilizan SRAM). Amdahl sugirió la regla empírica que la capacidad de memoria debería crecer linealmente con la velocidad de la CPU para mantener un sistema equilibrado (ver Sección 1.4), y los diseñadores de la CPU contaron con las DRAM para atender esa demanda: esperaban una mejora de cuatro veces en capacidad cada tres años. Desgraciadamente, el rendimiento de las DRAM está creciendo a una velocidad mucho más lenta. La Figura 8.17 muestra una mejora del rendimiento en el tiempo de acceso a filas de aproximadamente el 22 por 100 por generación, o el 7 por 100 por año. Como se observó en el Capítulo 1, el rendimiento de CPU mejoró del 18 al 35 por 100 por año antes de 1985, y desde esa época ha saltado del 50 al 100 por 100 por año. La Figura 8.18 dibuja estas proyecciones optimistas y pesimistas de rendimiento de la CPU frente a la mejora estacionaria de rendimiento del 7 por 100 en la velocidad de las DRAM.

La diferencia de rendimiento de CPU-DRAM es claramente un problema sobre el horizonte —la Ley de Amdahl nos avisa lo que ocurrirá si ignoramos una parte de la computación mientras tratamos de acelerar el resto—. La

Año de introducción	Tamaño del chip	Acceso a filas DRAM más lenta	Acceso a filas DRAM más rápida	Acceso a columna (CAS)	Tiempo de ciclo
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992?	16 Mbit	≈85 ns	≈65 ns	≈15 ns	≈140 ns

FIGURA 8.17 Tiempos de las DRAM rápidas y lentas de cada generación. La mejora por un factor de dos en los accesos a columnas se produjo junto con el cambio de DRAM NMOS a DRAM CMOS. Con tres años por generación, la mejora de rendimiento del tiempo de acceso a filas es aproximadamente el 7 por 100 por año. Los datos de la última fila representan el rendimiento predicho para las DRAM de 16 Mbits, que no están todavía disponibles.

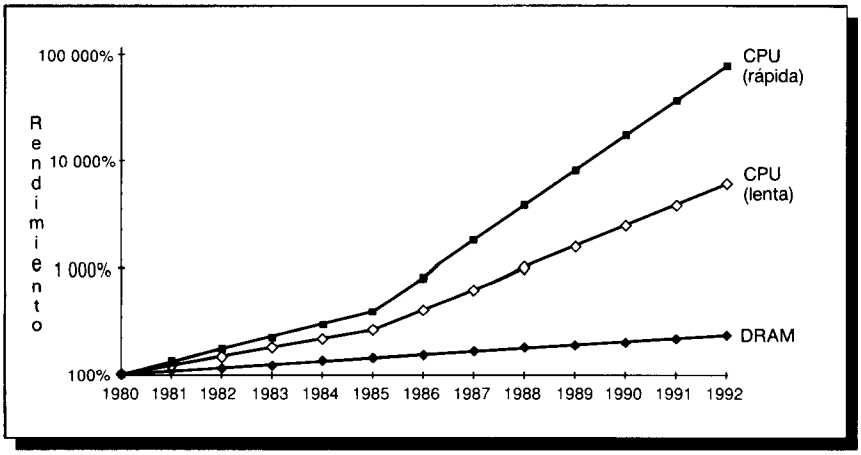


FIGURA 8.18 Comenzando con el rendimiento de 1980 como punto de partida, se dibuja el rendimiento de las DRAM y CPU a lo largo del tiempo. El punto de partida de DRAM de 64 KB en 1980, con tres años para la siguiente generación. La línea de CPU lentas supone una mejora del 19 por 100 por año hasta 1985 y una mejora del 50 por 100 después. La línea de CPU rápidas supone una mejora del rendimiento del 25 por 100 entre 1980 y 1985 y un 100 por 100 por año después. Observar que el eje vertical debe estar en la escala logarítmica para registrar el tamaño del salto de rendimientos CPU-DRAM.

Sección 8.8 describirá lo que puede hacerse con la organización cache para reducir esta diferencia de rendimiento, pero simplemente hacer caches mayores no puede eliminarlo. También se necesitan organizaciones innovadoras de memoria principal. En el resto de esta sección examinaremos técnicas para organizar memorias con el fin de mejorar el rendimiento, incluyendo especialmente técnicas para las DRAM.

Organizaciones para mejorar el rendimiento de la memoria principal

Aunque generalmente es más fácil mejorar el ancho de banda de memoria con nuevas organizaciones que reducir la latencia, una mejora del ancho de banda permite incrementar el tamaño de los bloques de cache sin el correspondiente incremento en la penalización de fallos.

Ilustramos estas organizaciones mostrando cómo se satisface un fallo de cache. Suponer que el rendimiento de la organización básica de memoria es

1 ciclo de reloj para enviar la dirección

6 ciclos de reloj para el tiempo de acceso por palabra

1 ciclo de reloj para enviar una palabra de datos

Dado un bloque de cache de cuatro palabras, la penalización de fallos es de 32 ciclos de reloj, con un ancho de banda de memoria de medio byte por ciclo de reloj.

La Figura 8.19 muestra algunas de las opciones para conseguir sistemas de memoria más rápidos. La aproximación más sencilla para incrementar el ancho de banda de memoria, entonces, es hacer la memoria más ancha.

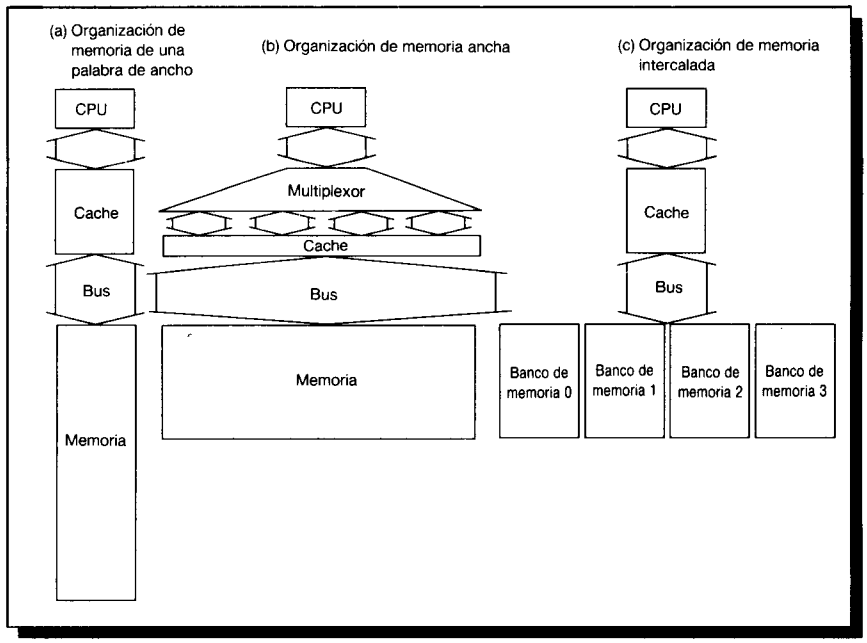


FIGURA 8.19 Tres ejemplos de anchura de bus, ancho de memoria, y entrelazado de memoria para lograr mayor ancho de banda de memoria. a) es el diseño más simple, siendo todo de la anchura de una palabra; b) muestra una memoria, bus y cache más anchos; mientras que c) muestra una cache y bus delgados con una memoria entrelazada.

Memoria principal más ancha

Las caches están, con frecuencia, organizadas con una anchura de una palabra porque la mayoría de los accesos de la CPU son de ese tamaño. A su vez, la memoria principal tiene el ancho de una palabra para que coincida con la anchura de la cache. Duplicar o cuadruplicar el ancho de la memoria, por tanto, duplicará o cuadruplicará el ancho de banda de memoria. Con una anchura de memoria principal de dos palabras, la penalización de fallos en nuestro ejemplo caería desde $4 \cdot 8$ o 32 ciclos de reloj a $2 \cdot 8$ o 16 ciclos de reloj. Con cuatro palabras de ancho la penalización de fallos es exactamente $1 \cdot 8$ ciclos de reloj. El ancho de banda es entonces un byte por ciclo de reloj en un ancho de dos palabras y de dos bytes por ciclo de reloj cuando la memoria tiene un ancho de cuatro palabras.

Hay coste en el bus más ancho. La CPU accederá todavía a la cache una palabra cada vez, por ello, se necesita ahora un multiplexor entre la cache y la CPU —y ese multiplexor puede estar en el camino crítico de temporización—. (Sin embargo, si la cache es más rápida que el bus, el multiplexor se puede colocar entre la cache y el bus.) Otro inconveniente es que como la memoria principal es tradicionalmente expansible por el usuario, el incremento mínimo se duplica o cuadruplica. Finalmente, las memorias con corrección de errores tienen dificultades con las escrituras en una parte del bloque protegido (por ejemplo, una escritura de un byte); el resto de los datos se debe leer para que el nuevo código de corrección de errores se pueda calcular y almacenar cuando se escriba el dato. Si la corrección de errores se hace sobre la anchura completa, la memoria más ancha incrementará la frecuencia de estas secuencias de «leer-modificar-escribir», porque más escrituras se convierten en escrituras parciales de bloques. Muchos diseños de memoria más ancha han separado corrección de errores cada 32 bits, ya que la mayor parte de las escrituras tienen ese tamaño. Un ejemplo de memoria principal más ancha fue un computador cuya cache, bus y memoria eran todos de 512 bytes de ancho.

Memoria entrelazada

Los chips de memoria se pueden organizar en bancos para leer o escribir múltiples palabras a la vez, en lugar de una sola palabra. Los bancos son de una palabra de ancho para que la anchura del bus y de la cache no necesiten cambiar, pero enviando direcciones a varios bancos, les permite a todos leer simultáneamente. Por ejemplo, enviar una dirección a cuatro bancos (con los tiempos de acceso mostrados en la pág. 460) da una penalización de fallos de $1 + 6 + 4 \cdot 1$ o 11 ciclos de reloj, dando un ancho de banda de, aproximadamente 1,5 bytes por ciclo de reloj. Los bancos también son útiles en las escrituras. Aunque las escrituras muy seguidas normalmente tendrán que esperar a que acaben las escrituras anteriores, los bancos permiten un ciclo de reloj para cada escritura, siempre que no estén destinadas al mismo banco.

La correspondencia entre direcciones y bancos afecta el comportamiento del sistema de memoria. El ejemplo anterior supone que las direcciones de cuatro bancos están entrelazadas a nivel de palabra —el banco 0 tiene todas

las palabras cuya dirección módulo 4 es 0, el banco 1 todas las palabras cuya dirección módulo 4 es 1, y así sucesivamente. Esta correspondencia se denomina *factor de entrelazado; memoria entrelazada,* normalmente, significa bancos de memoria que están entrelazadas a nivel de palabra. Esto optimiza los accesos secuenciales a memoria. Un fallo de lectura de cache es un caso ideal para una memoria entrelazada a nivel de palabra, ya que las palabras de un bloque se leen secuencialmente. Las caches de postescritura hacen escrituras así como lecturas secuenciales, obteniendo incluso más eficiencia de la memoria entrelazada.

Ejemplo

Considerar la siguiente descripción de una máquina y su rendimiento de cache:

Tamaño de bloque = 1 palabra

Anchura del bus de memoria = 1 palabra

Frecuencia de fallos = 15 por 100

Accesos a memoria por instrucción = 1,2

Penalización de fallos de cache = 8 ciclos (como antes)

Ciclos medios por instrucción (ignorando los fallos de cache) = 2

Si cambiamos el tamaño de bloque a dos palabras, la frecuencia de fallos cae al 10 por 100, y un bloque de cuatro palabras tiene una frecuencia de fallos del 5 por 100. ¿Cuál es la mejora en el rendimiento al entrelazar dos y cuatro vías frente a duplicar el ancho de memoria y del bus, suponiendo los tiempos de acceso de la página 460?

Respuesta

El CPI para la máquina base utilizando bloques de una palabra es

$$2 + (1{,}2 \cdot 15\,\% \cdot 8) = 3{,}44$$

Como la duración del ciclo de reloj y el recuento de instrucciones no puede cambiar en este ejemplo, podemos calcular la mejora de rendimiento comparando el CPI.

Incrementar el tamaño de bloque a dos palabras da las siguientes opciones:

bus y memoria de 32 bits, sin entrelazado: $= 2 + (1{,}2 \cdot 10\,\% \cdot 2 \cdot 8) = 3{,}92$

bus y memoria de 32 bits, con entrelazado: $= 2 + (1{,}2 \cdot 10\,\% \cdot (1 + 6 + 2)) = 3{,}08$

bus y memoria de 64 bits, sin entrelazado: $= 2 + (1{,}2 \cdot 10\,\% \cdot 1 \cdot 8) = 2{,}96$

Por tanto, doblando el tamaño del bloque se ralentiza la implementación más simple (3,92 frente a 3,44), mientras que con entrelazado o memoria más

ancha el sistema es el 12 o el 16 por 100 más rápido, respectivamente. Si incrementamos el tamaño del bloque a cuatro, se obtiene lo siguiente:

bus y memoria de 32 bits, sin entrelazado $= 2 + (1{,}2 \cdot 5\,\% \cdot 4 \cdot 8) = 3{,}92$

bus y memoria de 32 bits, con entrelazado $= 2 + (1{,}2 \cdot 5\,\% \cdot (1 + 6 + 4)) = 2{,}66$

bus y memoria de 64 bits, sin entrelazado $= 2 + (1{,}2 \cdot 5\,\% \cdot 2 \cdot 8) = 2{,}96$

De nuevo, el bloque mayor penaliza el rendimiento para el caso simple, aunque la memoria entrelazada de 32 bits sea ahora más rápida —29 por 100 frente al 16 por 100 para la memoria y bus más anchos.

La motivación original para los bancos de memoria fue entrelazar accesos secuenciales. Una razón adicional es permitir múltiples accesos independientes. Controladores múltiples de memoria permiten bancos (o conjuntos de bancos de entrelazados por palabras) operar independientemente. Por ejemplo, un dispositivo de entrada puede utilizar un controlador y su memoria, la cache puede utilizar otro, y una unidad vectorial puede utilizar un tercero. Para reducir las oportunidades de conflictos se necesitan muchos bancos; la SX/3 de NEC, por ejemplo, tiene hasta 128 bancos.

Cuando aumenta la capacidad por chip de memoria, hay menos chips en un sistema de memoria del mismo tamaño, haciendo mucho más caros múltiples bancos. Por ejemplo, una memoria principal de 16 MB emplea 512 chips de memoria de 256 K (262 144) × 1 bit, organizada fácilmente en 16 bancos de 32 chips de memoria. Pero sólo emplea 32 chips de memoria de 4 M (4 194 304) × 1 bit para 16 MB, haciendo que el límite sea un banco. Esta es la principal desventaja de los bancos de memoria entrelazados. Aun cuando la regla empírica de Amdahl/Case para sistemas equilibrados de computadores recomiende aumentar la capacidad de memoria con el incremento del rendimiento de la CPU, un crecimiento del 60 por 100 en la capacidad de DRAM era superior en el pasado a la velocidad de incremento del rendimiento de la CPU (pág. 18, Cap. 1). Si la frecuencia de incremento de velocidad de CPU, vista a finales de los años ochenta, puede mantenerse (Fig. 8.18) y estos sistemas siguen la regla empírica de Amdahl/Case, entonces el número de chips puede no reducirse.

Una segunda desventaja del entrelazado es, de nuevo, la dificultad de la expansión de memoria principal. Como el hardware de control de memoria probablemente necesitará bancos de igual tamaño, duplicar la memoria principal probablemente será el incremento mínimo.

Entrelazado de DRAM específico para mejorar el rendimiento de la memoria principal

Los tiempos de acceso a las DRAM se dividen en accesos a filas y accesos a columnas. Las DRAM disponen de un buffer de una fila de bits dentro de la

DRAM para los accesos a las columnas. Esta fila es habitualmente la raíz cuadrada del tamaño de la DRAM —1 024 bits para 1 Mbit, 2 048 para 4 Mbits, y así sucesivamente. Todas las DRAM vienen con señales de temporización opcional que permiten accesos repetidos al buffer sin un tiempo de acceso a filas. Hay tres versiones para esta optimización:

- *Modo nibble.* La DRAM puede proporcionar tres bits extra de posiciones secuenciales para cada acceso a filas.
- *Modo de página.* El buffer actúa como una SRAM; cambiando las direcciones de las columnas, se puede acceder a bits aleatorios en el buffer hasta el siguiente acceso a fila o tiempo de refresco.
- *Columna estática.* Muy similar al modo de página, excepto que no es necesario acertar la línea de strobe de acceso a columnas cada vez que cambie la dirección de la columna; esta opción se ha denominado SCRAM, para la DRAM de columna estática.

A partir de DRAM de 1 Mbit, la mayoría de los dados pueden realizar alguna de las tres opciones, seleccionando la optimización en el instante que el dado se encapsula al escoger los «pads» que se van a usar. Estas operaciones cambian la definición del tiempo del ciclo para las DRAM. La Figura 8.20 muestra la duración tradicional del ciclo más la velocidad más rápida entre accesos en el modo optimizado.

La ventaja de estas optimizaciones es que utilizan la circuitería ya en las DRAM, añadiendo poco costo al sistema mientras logran casi una mejora de cuatro veces en el ancho de banda. Por ejemplo, el modo nibble se diseñó para aprovechar el mismo comportamiento de programa que la memoria entrelazada. El chip lee internamente cada vez cuatro bits, suministrando externamente cuatro bits en el tiempo de cuatro ciclos optimizados. A menos que la duración de la transferencia del bus sea más rápida que la duración del ciclo optimizado, el coste de memoria entrelazada de cuatro vías es sólo el control de temporización más complicado. El modo de página y de columna estática también se podrían utilizar para obtener aún mayor entrelazado con un control ligeramente más complejo. Las DRAM tienden a tener buffers tres-estados débiles, implicando que el entrelazado tradicional con más chips de memoria debe incluir chips de buffers para cada banco de memoria.

Por ello, los autores esperan que la mayoría de los sistemas de memoria principal utilicen en el futuro tales técnicas para reducir la diferencia de rendimiento CPU-DRAM. De forma distinta a las memorias entrelazadas tradicionalmente, no hay desventajas al utilizar estos modos de DRAM cuando éstas aumentan en capacidad, ni existe el problema del mínimo incremento de expansión en memoria principal.

Una posibilidad recientemente aparecida son las DRAM que no multiplexan las líneas de dirección. Al coste de mayor encapsulamiento, un acceso completamente aleatorio tardaría entre el tiempo de acceso a filas y de acceso a columnas en la Figura 8.20. Si las DRAM no codificadas pueden mantenerse próximas al precio por bit del alto volumen de las DRAM codificadas, el arquitecto de computadores tendrá otra opción en su bolsa de trucos para el diseño de memorias.

Tamaño del chip	Acceso a fila DRAM más lenta	Acceso a fila DRAM más rápida	Acceso a columna	Tiempo de ciclo	Tiempo nibble página, columna estática optimizada
64 Kbits	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kbits	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mbits	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mbits	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mbits	≈85 ns	≈65 ns	≈15 ns	≈140 ns	≈30 ns

FIGURA 8.20 Duración de ciclo de DRAM para los accesos optimizados. Estos datos son los mismos que los de la Figura 8.17 con una columna más para mostrar la duración del ciclo optimizado para los tres modos. A partir de la DRAM de 1 Mbit, el tiempo de ciclo optimizado es aproximadamente cuatro veces más rápido que el tiempo de ciclo no optimizado. Es tan rápido que el modo de página fue renombrado *modo rápido de página*. El tiempo de ciclo optimizado es el mismo sin importar cuál de los tres modos optimizados se utilice.

8.5 Memoria virtual

...se ha inventado un sistema para hacer que la combinación de tambores de núcleos magnéticos parezca al programador como un simple nivel de almacenamiento, realizándose automáticamente las transferencias precisas.

Kilburn y cols. [1962]

En cualquier instante de tiempo los computadores están corriendo múltiples procesos, cada uno con su propio espacio de direcciones. (Los procesos se describen en la siguiente sección.) Sería muy caro dedicar una memoria de tamaño igual al espacio total de direcciones a cada proceso, especialmente dado que muchos procesos utilizan sólo una pequeña parte de su espacio de direcciones. De aquí, que deba haber un medio de compartir una cantidad más pequeña de memoria física entre muchos procesos. Una forma de hacer esto, la *memoria virtual,* divide la memoria física en bloques y los asigna a diferentes procesos. Inherente a esta aproximación, debe existir un esquema de *protección* que restrinja un proceso a los bloques que pertenecen exactamente a ese proceso. La mayoría de las formas de memoria virtual reducen también el tiempo para arrancar un programa, ya que no es necesario que todo el código y los datos estén en la memoria física antes de que pueda comenzar un programa.

Aunque la memoria virtual es esencial para los computadores actuales, la compartición no es la razón por la cual se inventó la memoria virtual. En los primeros días si un programa era demasiado grande para la memoria física, incumbía al programador ajustarlo. Los programadores dividían los progra-

mas en partes y entonces identificaban las partes que eran mutuamente exclusivas. Estos *recubrimientos (overlays)* se cargaban y descargaban bajo control del usuario durante la ejecución del programa, asegurando el programador que el programa nunca trataba de acceder a más memoria física principal que la de la máquina. Como se puede imaginar, esta responsabilidad erosionaba la productividad del programador. La memoria virtual, inventada para aligerar a los programadores de este peso, gestionaba automáticamente los dos niveles de la jerarquía de memoria representada por la memoria principal y la secundaria.

Además de compartir el espacio protegido de memoria y gestionar automáticamente la jerarquía de memoria, la memoria virtual también simplifica la carga del programa para su ejecución. Denominado *reubicación,* este procedimiento permite que el mismo programa se ejecute en cualquier posición de la memoria física. (Antes de la popularidad de la memoria virtual, las máquinas incluían un registro de reubicación para este propósito.) Una alternativa a una solución hardware podía ser el software que cambiaba todas las direcciones de un programa cada vez que se ejecutaba.

Algunos términos generales de la jerarquía de memoria de la Sección 8.3 se aplican a la memoria virtual, aunque otros términos sean diferentes. *Página* o *segmento* se utiliza para bloque, y *fallo (fault) de página* o *fallo (fault) de dirección,* se utilizan para fallo *(miss).* Con la memoria virtual, la CPU produce *direcciones virtuales* que son traducidas por una combinación de hardware y software a *direcciones físicas,* que pueden ser utilizadas para acceder a memoria principal. Este proceso se denomina *correspondencia de memoria* o *traducción de direcciones.* Hoy día, los dos niveles de la jerarquía de memoria controlados por la memoria virtual son las DRAM y los discos magnéticos. La Figura 8.21 muestra un rango típico de los parámetros de la jerarquía de memoria para la memoria virtual.

Hay otras diferencias entre las caches y memoria virtual además de las cuantitativas observadas al comparar la Figura 8.21 con la Figura 8.5:

- El reemplazo los fallos de cache está controlado principalmente por hardware, mientras que el reemplazo en memoria virtual está controlado prin-

Tamaño bloque (página)	512-8192 bytes
Tiempo de acierto	1-10 ciclos de reloj
Penalización de fallos	100 000-600 000 ciclos de reloj
(tiempo de acceso)	(100 000-500 000 ciclos de reloj)
(tiempo de transferencia)	(10 000-100 000 ciclos de reloj)
Frecuencia de fallos	0,00001 %-0,001 %
Tamaño de memoria principal	4 MB-2048 MB

FIGURA 8.21 Rangos típicos de parámetros para memoria virtual. Estas cifras, contrastadas con los valores para caches de la Figura 8.5, representan incrementos de 10 a 100 000 veces.

cipalmente por el sistema operativo; una penalización más grande de fallos significa que el sistema operativo puede estar involucrado y emplear más tiempo decidiendo qué sustituir.

- El tamaño de la dirección del procesador determina el tamaño de la memoria virtual, pero el tamaño de la cache es normalmente independiente de la dirección del procesador.
- Además de actuar como memoria de más bajo nivel para la memoria principal en la jerarquía, la memoria secundaria también se utiliza para el sistema de ficheros que normalmente no es parte del espacio de direcciones; la mayor parte de la memoria secundaria está, en efecto, ocupada por el sistema de ficheros.

La memoria virtual abarca varias técnicas relacionadas. Los sistemas de memoria virtual se pueden categorizar en dos clases: los de bloques de tamaño fijo, denominados *páginas,* y los de bloques de tamaño variable, denominados *segmentos.* Las páginas, normalmente, tienen un tamaño fijo entre 512 y 8 192 bytes, mientras que el tamaño del segmento es variable. El segmento mayor soportado en cualquier máquina varía desde 2^{16} bytes hasta 2^{32} bytes; el segmento más pequeño es de un byte.

La decisión de utilizar memoria virtual paginada frente a la segmentada afecta a la CPU. El direccionamiento paginado tiene una única dirección de tamaño fijo dividida en número de página y desplazamiento en una página, análogo al direccionamiento de la cache. Una única dirección no sirve para las direcciones segmentadas; el tamaño variable de los segmentos requiere una palabra para un número de segmento y otra palabra para un desplazamiento dentro del segmento, dando un total de dos palabras. Un espacio de direcciones no segmentadas es más simple para el compilador.

Los pros y contras de estos dos enfoques han sido bien documentados en libros de texto de sistemas operativos; y se resumen en la Figura 8.22. A causa del problema del reemplazo (la tercera línea de la figura), pocas máquinas utilizan hoy día segmentación pura. Algunas máquinas utilizan un enfoque híbrido, denominado *de segmentos paginados,* en el que un segmento es un número entero de páginas. Esto simplifica el reemplazo porque no se necesita que la memoria sea contigua, ni que los segmentos completos estén en memoria principal.

Ahora estamos preparados para responder a las cuatro preguntas de la jerarquía de memoria para la memoria virtual.

P1. ¿Dónde puede ubicarse un bloque en memoria principal?

La penalización de fallos para memoria virtual involucra el acceso a un dispositivo giratorio de memoria magnética y es además bastante alta. Puestos a elegir entre reducir la frecuencia de fallos o un algoritmo de ubicación más sencillo, los diseñadores de los sistemas operativos siempre escogen la frecuencia de fallos más baja, debido al enorme coste de un fallo. Por tanto, los sistemas operativos permiten que los bloques se coloquen en cualquier parte de memoria principal. De acuerdo con la terminología de la Figura 8.6, esta estrategia podría denominarse totalmente asociativa.

	Página	Segmento
Palabras por dirección	Una	Dos (segmento y desplazamiento)
¿Visible al programador?	Invisible a la aplicación del programador	Puede ser visible a la aplicación del programador
Reemplazo de un bloque	Trivial (todos los bloques tienen el mismo tamaño)	Difícil (debe encontrar una parte no utilizada de memoria principal de tamaño variable y contigua)
Uso ineficiente de memoria	Fragmentación interna (porción inutilizada de página)	*Fragmentación externa* (partes no usadas de memoria principal)
Tráfico de disco eficiente	Sí (ajusta tamaño de página para equilibrar tiempo de acceso y tiempo de transferencia)	No siempre (pequeños segmentos pueden transferir sólo unos pocos bytes)

FIGURA 8.22 Paginación frente a segmentación. Ambas pueden desperdiciar memoria, dependiendo del tamaño del bloque y cómo se ajusten los segmentos en memoria principal. Los lenguajes de programación con punteros no restringidos requieren que el segmento y el desplazamiento sean pasados. Una aproximación híbrida, denominada *segmentos paginados,* toma lo mejor de ambos mundos: los segmentos están compuestos de páginas, de forma que reemplazar un bloque sea fácil; sin embargo, un segmento se puede tratar como una unidad lógica.

P2. ¿Cómo se encuentra un bloque si está en memoria principal?

Tanto la paginación como la segmentación cuentan con una estructura de datos que está indexada por el número de página o segmento. Esta estructura de datos contiene la dirección física del bloque. Para la paginación, el desplazamiento es sencillamente concatenado a la dirección física de la página (ver Fig. 8.23). Para la segmentación, el desplazamiento se suma a la dirección física del segmento para obtener la dirección virtual final.

Esta estructura de datos que contiene la dirección física de la página, habitualmente, tiene la forma de una *tabla de páginas.* Indexada por el número de página virtual, el tamaño de la tabla es el número de páginas del espacio de direcciones virtuales. Dada una dirección virtual de 28 bits, páginas de 4 KB y 4 bytes por entrada de tabla de página, el tamaño de la tabla de página será 256 KB. Para reducir el tamaño de esta estructura de datos, algunas máquinas aplican una función de *hashing* a la dirección virtual para que la estructura de datos sólo necesite tener el tamaño del número de páginas **físicas** de la memoria principal; este número será mucho más pequeño que el número de páginas virtuales. Dicha estructura se denomina *tabla de páginas invertida.* Utilizando el ejemplo anterior, una memoria física de 64 MB solamente necesitará 64 KB (4 · 64 MB/4 KB) para una tabla de páginas invertida.

Para reducir el tiempo de conversión de direcciones, los computadores utilizan una cache dedicada a estas conversiones, denominada buffer de traduc-

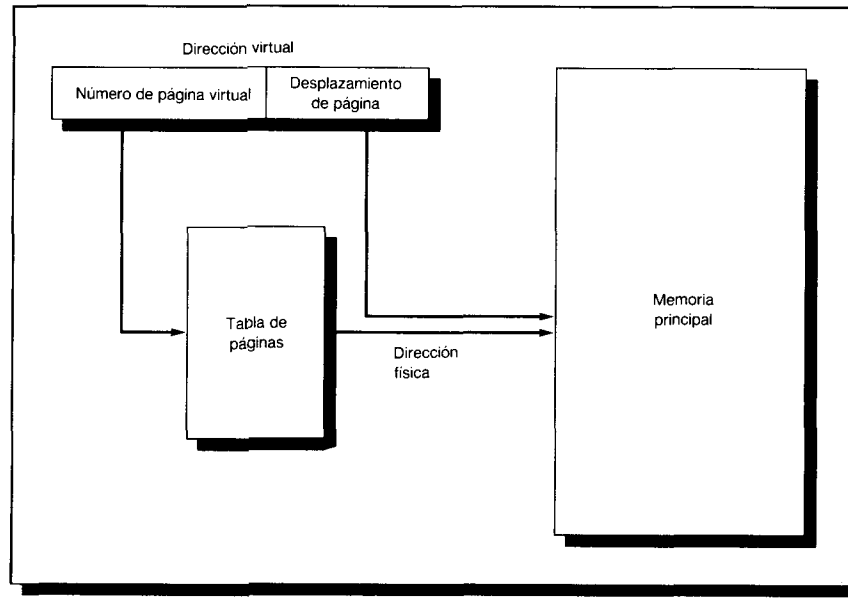


FIGURA 8.23 La correspondencia de una dirección virtual a una dirección física vía una tabla de páginas.

ción anticipada, o simplemente buffer de traducción. En breve se describen con más detalle.

P3. ¿Qué bloque debería sustituirse en un fallo de memoria virtual?

Como mencionamos antes, la directriz predominante de los sistemas operativos es minimizar los fallos de página. Consistente con esto, casi todos los sistemas operativos intentan sustituir el bloque menos recientemente usado (LRU), porque es el que menos, probablemente, se necesitará. Para ayudar al sistema operativo a estimar el LRU, muchas máquinas proporcionan un *bit de uso o bit de referencia,* que se pone a uno siempre que se accede a una página. El sistema operativo borra periódicamente los bits de uso y más tarde los registra para poder determinar qué páginas fueron tocadas durante un período de tiempo particular. Siguiendo la pista de esta forma, el sistema operativo puede seleccionar una página que se encuentre entre las menos recientemente referenciadas.

P4. ¿Qué ocurre en una escritura?

El nivel siguiente a memoria principal contiene discos giratorios magnéticos que tienen accesos de cientos de miles de ciclos de reloj. Debido a la gran discrepancia en el tiempo de acceso, no se ha construido todavía un sistema operativo de memoria virtual que pueda escribir, a través de memoria principal, directamente en el disco en cada almacenamiento de la CPU. (¡Esta observa-

ción no cabe ser interpretada como una oportunidad para llegar a ser famoso al ser el primero en construir uno!) Por ello, la estrategia de escritura es siempre la de postescritura. Como el coste de un acceso innecesario al siguiente nivel más bajo es alto, los sistemas de memoria virtual incluyen un bit de modificación *(dirty)* para que sólo los bloques que han sido alterados desde que se cargaron sean escritos en el disco.

Selección del tamaño de página

El parámetro arquitectónico más obvio es el tamaño de página. Elegir el tamaño de página es una cuestión de equilibrar las fuerzas que favorecen un tamaño de página mayor frente a las que favorecen un tamaño más pequeño. Lo que sigue favorece un tamaño de página mayor:

- El tamaño de la tabla de página es inversamente proporcional al tamaño de la página; puede ahorrarse memoria (u otros recursos utilizados para la correspondencia de memoria) al hacer las páginas mayores.
- Transferir páginas mayores a o desde memoria secundaria, posiblemente sobre una red, es más eficiente que transferir páginas más pequeñas.

(El tamaño mayor de página puede también ayudar a la traducción de las direcciones de cache; ver Sección 8.8.)

La principal motivación para un tamaño de página más pequeño es conservar memoria. Un tamaño de página pequeño dará como consecuencia menos memoria malgastada cuando una región contigua de memoria virtual no sea igual en tamaño a un múltiplo del tamaño de página. El término para esta memoria no utilizada en una página, es *fragmentación interna.* Suponiendo que cada proceso tiene tres segmentos principales (texto, montículo *[heap]* y pila), la memoria media gastada por proceso será 1,5 veces el tamaño de página. Esto es insignificante para máquinas con megabytes de memoria y tamaños de página en el rango de 2 KB a 8 KB. Por supuesto, cuando los tamaños de página son muy grandes (más de 32 KB), se puede malgastar mucha memoria (principal y secundaria), así como ancho de banda de E/S. Un aspecto final es el tiempo de arranque de un proceso; muchos procesos son pequeños, así que mayores tamaños de página alargarán el tiempo de invocar un proceso.

Técnicas para traducción rápida de direcciones

Las tablas de página son habitualmente tan grandes que se almacenan en memoria principal y, con frecuencia, paginadas ellas mismas. Esto significa que cada acceso a memoria es como mínimo el doble de largo, un acceso a memoria para obtener la dirección física y un segundo acceso para obtener el dato. Este coste es excesivo.

Un remedio es recordar la última traducción, para que el proceso de co-

Tamaño de bloque	4-8 bytes (1 entrada de tabla de páginas)
Tiempo de acierto	1 ciclo de reloj
Penalización de fallos	10-30 ciclos de reloj
Frecuencia de fallos	0,1 %-2 %
Tamaño TLB	32-8192 bytes

FIGURA 8.24 Valores típicos de parámetros clave de jerarquía de memoria para TLB. Los TLB son simplemente caches para la traducción de las direcciones, de virtuales a físicas, contenidas en las tablas de página.

rrespondencia se evite si la dirección actual referencia la misma página que la última. Una solución más general es confiar de nuevo en el principio de localidad; si las referencias tienen localidad, entonces la traducción de direcciones para las referencias también debe tener localidad. Manteniendo estas traducciones de direcciones en una cache especial, un acceso a memoria raramente requiere un segundo acceso para traducir el dato. Esta cache especial de traducción de direcciones se conoce como *buffer de traducciones anticipadas (translation-lookaside buffer)* o TLB; también se denomina «buffer de traducciones» o TB. Una entrada del TLB es como una entrada de la cache donde la etiqueta contiene parte de la dirección virtual y la parte del dato contiene un número de estructura de página, campo de protección, bit de uso y bit de modificación. Para cambiar el número físico de estructura de página o protección de una entrada en la tabla de páginas, el sistema operativo debe estar seguro de que la entrada antigua no está en el TLB; en cualquier otro caso, el sistema no se comportará adecuadamente. Observar que este bit de modificación significa que la **página** correspondiente está modificada, no que la traducción de direcciones en la TBL esté modificada ni que un bloque particular de la cache de datos esté modificado. La Figura 8.24 muestra parámetros típicos para los TLB.

Un desafío arquitectónico proviene de la dificultad de combinar caches con memoria virtual. La dirección virtual debe ir primero a través del TLB **antes** que la dirección física pueda acceder la cache, significando que el tiempo de acierto de cache debe alargarse para permitir una traducción de direcciones (o que la segmentación se puede alargar como en el Cap. 6). Una forma para reducir el tiempo de aciertos es acceder a la cache con el desplazamiento de página, la parte de la dirección virtual que no necesita ser traducida. Mientras se están leyendo las etiquetas de dirección de la cache, la parte virtual de la dirección (dirección de la estructura de página) se envía al TLB para que sea traducida. La comparación de direcciones se realiza entre la dirección física del TLB y la etiqueta de la cache. Como el TLB es habitualmente más pequeño y rápido que la memoria de etiquetas de la cache, la lectura del TLB puede hacerse de forma simultánea a la lectura de la memoria de etiquetas sin ralentizar los tiempos de acierto de la cache. El inconveniente de este esquema es que una cache de correspondencia directa no puede ser mayor que una página. Otra opción, las caches direccionadas virtualmente, se explica en la Sección 8.8.

8.6 Protección y ejemplos de memoria virtual

La invención de la multiprogramación condujo a nuevas demandas para protección y compartición entre programas. Estos conceptos están muy ligados con la memoria virtual de los computadores actuales y por ello cubrimos aquí este tema, junto con dos ejemplos de memoria virtual.

La multiprogramación conduce al concepto de *proceso*. Metafóricamente, un proceso es el espacio donde vive y el aire que respira un programa; es decir, un programa en ejecución más cualquier estado necesario para continuar la ejecución del mismo. Tiempo compartido significa compartir la CPU y memoria con varios usuarios al mismo tiempo para dar el aspecto de que cada usuario tiene su propia máquina. Por tanto, en cualquier instante debe ser posible cambiar de un proceso a otro. Esto se denomina *cambio de procesos* o *cambio de contexto*. La Figura 8.25 muestra la frecuencia de estos cambios en el VAX 8700.

Un proceso debe operar correctamente tanto si se ejecuta continuamente, desde que comienza hasta que termina, como si se interrumpe repetidamente y cambia con otros procesos. La responsabilidad para mantener el comportamiento correcto del proceso es compartida por el diseñador de computadores, que debe asegurar que la parte de CPU del estado del proceso se pueda almacenar y restaurar, y el diseñador de sistemas operativos, que debe garantizar que el proceso no interfiera con los cálculos de los demás. La forma más segura para proteger el estado de un proceso de los demás sería copiar la información actual en el disco. Pero un combio de procesos tardaría segundos —demasiado tiempo para un entorno de tiempo compartido—. El problema lo resuelven los sistemas operativos particionando la memoria principal para que procesos diferentes tengan, a la vez, su estado en memoria. Esto significa que el diseñador de sistemas operativos necesita ayuda del diseñador de computadores para lograr que un proceso no pueda modificar a los demás. Además de la protección, los computadores también proporcionan la posibilidad de compartición de código y datos entre procesos, para permitir comunicaciones entre procesos o ahorrar memoria reduciendo el número de copias con información idéntica.

Instrucciones entre cambios de procesos	19,353
Ciclos de reloj entre cambios de procesos	170,113
Tiempo entre cambios de procesos	7,7 ms

FIGURA 8.25 Frecuencia de cambio de procesos en el VAX 8700 para carga de trabajo de tiempo compartido. La mayoría de los cambios se presentan en interrupciones provocadas por eventos de E/S o por el temporizador de intervalos (ver Figura 5.10). Como ni la latencia del dispositivo de E/S, ni el temporizador están afectados por la velocidad del reloj de la CPU, máquinas más rápidas ejecutan generalmente más ciclos de reloj e instrucciones entre cambios de procesos.

Protección de procesos

El mecanismo más simple de protección es un par de registros que comprueban cada dirección para asegurarse que cae entre dos cotas denominadas tradicionalmente *base (base)* y *límite (bound).* Una dirección es válida si

$$\text{Base} \leqslant \text{Dirección} \leqslant \text{Límite}$$

En algunos sistemas la dirección se considera un número sin signo que siempre se suma a la base, así que el test de validez es

$$(\text{Base} + \text{Dirección}) \leqslant \text{Límite}$$

Para que los procesos de usuario estén protegidos entre sí, éstos no pueden modificar los registros base y límite; con todo, el sistema operativo debe poder modificar los registros para que puedan conmutar los procesos. Por consiguiente, el diseñador de computadores tiene tres responsabilidades más para que el diseñador de sistemas operativos proteja los procesos entre sí:

1. Proporcionar como mínimo dos modos, indicando si el proceso en ejecución *es* un proceso de usuario o un proceso del sistema operativo, a veces denominado proceso núcleo *(kernel),* proceso *supervisor* o proceso *ejecutivo.*
2. Proporcionar una parte del estado de la CPU, de forma que un proceso de usuario pueda utilizar pero no escribir. Esto incluye registros base/límite, bit(s) de modo usuario/supervisor y el bit de habilitación/inhabilitación de interrupción. Los usuarios tienen que evitar escribir en este estado porque el sistema operativo no puede controlar procesos de usuario si los usuarios pueden cambiar las comprobaciones del rango de direcciones, inhabilitar interrupciones o darse ellos mismos privilegios de supervisor.
3. Proporcionar mecanismos para que la CPU pueda ir desde el modo de usuario al modo supervisor y viceversa. El cambio a modo de supervisor, normalmente, se logra con una *llamada al sistema,* implementada como una instrucción especial que transfiere el control a una posición dedicada en el espacio de código del supervisor. El PC del punto de la llamada al sistema es salvado, y la CPU se coloca en modo supervisor. El retorno al modo usuario es como un retorno de subrutina que restaura el modo anterior usuario/supervisor.

Base y límite constituyen el sistema mínimo de protección. La memoria virtual proporciona una alternativa a este sencillo modelo. Como hemos visto, la dirección de la CPU debe ir a través de una correspondencia de dirección virtual a física. Esto proporciona la oportunidad para que el hardware compruebe además los errores del programa o proteja los procesos entre sí. La forma más simple de hacer esto es añadir etiquetas de permiso de acceso a cada página o segmento. Por ejemplo, como hoy día pocos programas modifican, intencionadamente, su propio código, un sistema operativo puede de-

tectar escrituras accidentales en el código ofreciendo sólo protección de lectura en las páginas. Esto puede extenderse añadiendo un bit de usuario/núcleo para impedir que un programa de usuario intente acceder a páginas que pertenecen al núcleo. Siempre que la CPU proporcione una señal de lectura/escritura y otra de usuario/núcleo, es fácil para el hardware de traducción de direcciones detectar accesos extraviados a memoria antes que puedan hacer daño. Como vimos en la Sección 5.6 del Capítulo 5, este comportamiento peligroso interrumpe la CPU. Obviamente, a los programas de usuario no se les permite modificar la tabla de páginas.

La protección puede intensificarse, dependiendo de la aprensión del diseñador de computadores o del comprador. Anillos añadidos a la estructura de protección de la CPU expanden la protección de los accesos a memoria de dos niveles (usuario y núcleo) a muchos más. Igual que en un sistema de clasificación militar: muy secreto, secreto, clasificado y no clasificado; *anillos* concéntricos de niveles de seguridad permiten a la persona de máxima confianza acceder a cualquier cosa, al segundo de más confianza acceder a cualquier cosa, excepto al nivel más interno, y así sucesivamente bajando hasta los programas «civiles» que son los de menos confianza y, por consiguiente, tienen el rango más limitado de accesos. También puede haber restricciones en el punto de entrada entre los niveles. La estructura de protección del 80286, que utiliza anillos, se describe más tarde en esta sección. No está claro, hoy día, si los anillos son una mejora al sencillo sistema de los modos usuario y núcleo.

Cuando la aprensión del diseñador desemboca en turbación, estos simples anillos pueden no ser suficientes. El hecho de que un programa en el santuario más interno pueda accecer a cualquier cosa, clama por un nuevo sistema de clasificación. En lugar de un modelo militar, la analogía del siguiente modelo es mediante llaves y cerraduras: un programa no puede acceder a los datos, a menos que tenga la llave. Para que estas llaves, o *capacidades,* sean útiles, el hardware y el sistema operativo deben poder explícitamente pasarlas de un programa a otro sin permitir que el mismo programa las falsifique. Estas comprobaciones requieren gran cantidad de soporte hardware.

Ejemplo de memoria virtual paginada: gestión de la memoria del VAX-11 y el TLB del VAX-11/780

La arquitectura VAX utiliza una combinación de segmentación y paginación. Esta combinación proporciona protección a la vez que minimiza el tamaño de la tabla de páginas. El espacio de direcciones se divide primero en dos segmentos: proceso (bit 31 = 0) y sistema (bit 31 = 1). Cada proceso tiene su propio espacio privado y comparte el espacio del sistema con cada uno de los demás procesos. El espacio de direcciones del proceso se subdivide, además, en dos regiones denominadas P0 y P1, que se distinguen utilizando el bit 30. El área P0 (bit 30 = 0) avanza desde la dirección 0 hacia adelante, mientras que P1 (bit 30 = 1) retrocede hacia la dirección 0. La Figura 8.26 muestra la organización de P0 y P1. Los dos segmentos pueden crecer hasta que uno alcance un tamaño que exceda su espacio de direcciones de 2^{30} bytes y se agote su memoria virtual. Muchos sistemas utilizan, hoy día, ambas combinaciones de segmentos predivididos y paginación. Este enfoque proporciona muchas

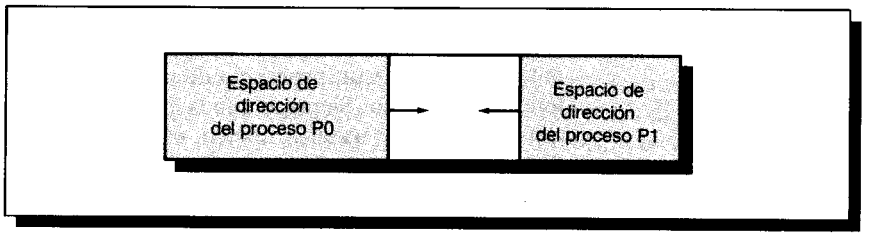


FIGURA 8.26 La organización de P0 y P1 en el VAX. Esto se corresponde con la mitad del espacio de direcciones dedicada a los procesos, seleccionado con un 0 en el bit 31 de una dirección virtual. El bit 30 de la dirección divide P0 y P1. Los sistemas operativos ponen las áreas de texto y monticulo en P0 y una pila de crecimiento descendente en P1.

ventajas: la segmentación divide el espacio de direcciones del sistema y del proceso y conserva el espacio de la tabla de páginas, mientras que la paginación proporciona memoria virtual, reubicación y protección.

Para conservar el espacio de la tabla de páginas, cada una de las tres regiones —proceso P0, proceso P1 y sistema— tiene un par de registros base-límite que indican el comienzo y final de la tabla de páginas para cada región. La alternativa sería tener una tabla de páginas simple que cubriera el espacio completo de direcciones, independientemente del tamaño real del programa. El tamaño pequeño de las páginas del VAX —512 bytes, resultando en grandes tablas de páginas— hace especialmente importante dicha conservación.

La Figura 8.27 muestra la correspondencia de una dirección de VAX. Los dos bits más significativos de una dirección seleccionan el segmento o par de registros base-límite que se van a utilizar para seleccionar una tabla de páginas y comprobar la referencia. Un uno en el primer bit selecciona la tabla de páginas del sistema, cuya base y longitud se encuentran en los registros de base y de longitud del sistema, respectivamente. Un cero en el primer bit de una dirección (como en la figura) selecciona la tabla de páginas de P0 o P1, determinada por los registros base de P0 o P1 y comprobada por los registros límite de P0 o P1. Las tablas de páginas de P0 y P1 están en la memoria virtual del espacio del sistema, mientras que la tabla de páginas del sistema está en la memoria física.

Esto ofrece una forma interesante para conservar la memoria física. Como las tablas de páginas de P0 y P1 están también en la memoria virtual, significa que las tablas de páginas se pueden paginar. Como parte del código y de los datos pueden permanecer en el disco durante la ejecución del programa, las entradas de traducción de la tabla de páginas para ese código y esos datos pueden permanecer en el disco hasta que se utilicen. Esto es especialmente importante para programas cuyo tamaño de memoria varíe dinámicamente durante la ejecución, las tablas de páginas pueden aumentar cuando crece el espacio de P0 o P1, entonces, en el peor caso, un fallo de página del proceso puede producir un segundo fallo de página en la parte que falta, de la tabla de páginas del proceso, necesaria para completar la traducción de las direcciones. ¿Qué evita que todas las tablas de páginas emigren a la memoria secundaria? Algunas tablas de páginas del sistema se cargan en la memoria física cuando arranca el sistema operativo y se impide que emigren al disco. De esa manera,

eventualmente, una serie de fallos debe terminar en una dirección almacenada en la tabla de páginas del sistema que está «congelada» en memoria principal.

Aunque esto explica la traducción de direcciones legales, ¿qué previene al usuario de crear traducciones de direcciones ilegales y de cometer errores? Las tablas de páginas están autoprotegidas de que sean escritas por los programas de usuario. Por ello, el usuario puede intentar cualquier dirección virtual, pero controlando las entradas de la tabla de páginas el sistema operativo controla a qué parte de la memoria física se accede. La compartición de memoria entre procesos se logra haciendo que una entrada de la tabla de páginas en cada espacio de direcciones apunte a la misma página de la memoria física.

Una *entrada de la tabla de páginas* (PTE) en la VAX es sencilla. Además del número de estructura de página física éstos son los únicos campos definidos por la arquitectura:

M: *bit de modificación (modify bit)* que indica que la página está modificada

V: *bit de validez* que indica que esta PTE tiene una dirección válida

PROT: cuatro bits de protección

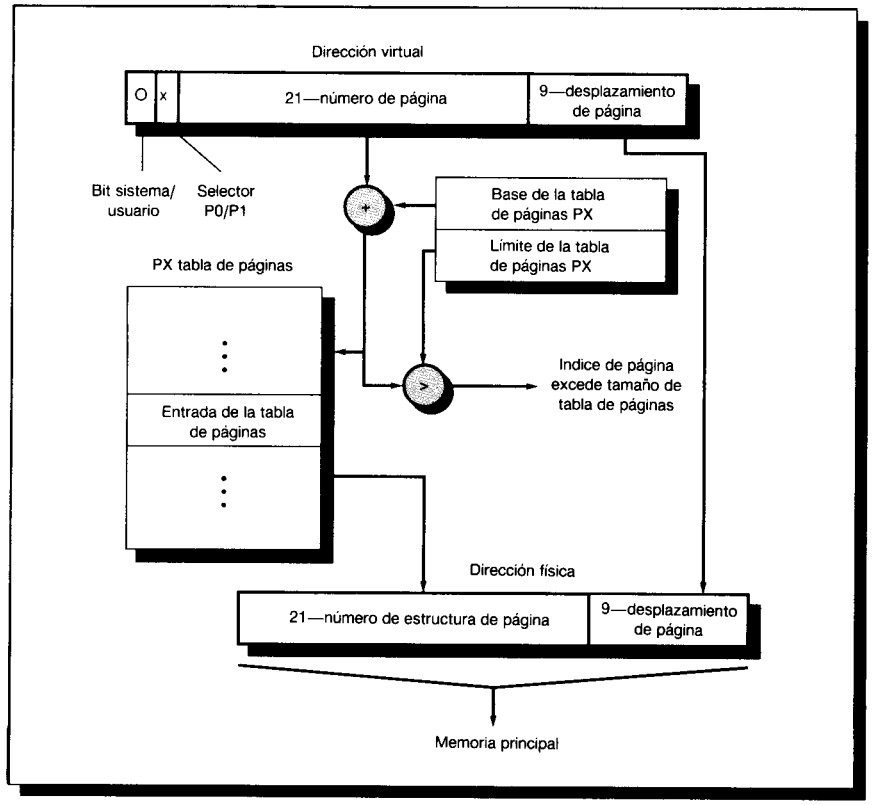


FIGURA 8.27 La correspondencia de una dirección virtual VAX. PX referencia o a P0 o a P1.

Tamaño de bloque	1 PTE (4 bytes)
Tiempo de acierto	1 ciclo de reloj
Penalización de fallos (promedio)	22 ciclos de reloj
Frecuencia de fallos	1 %-2 %
Tamaño de cache	128 PTEs (512 bytes)
Selección de bloques	Aleatoria
Estrategia de escritura	(No aplicable)
Ubicación de bloques	Asociativa por conjuntos de 2 vías

FIGURA 8.28 Parámetros de la jerarquía de memoria del TLB del VAX-11/780.

Observar que no hay bit de referencia o uso. Por consiguiente, un algoritmo de reemplazo de páginas, tal como el LRU, debe basarse en el bit de modificación o alguna técnica software para medir utilización. En lugar de una estructura de protección núcleo/usario, el VAX utiliza una estructura de cuatro niveles que consta de núcleo, ejecutivo, supervisor y usuario. Los cuatro bits de protección en la PTE contienen 16 codificaciones seleccionadas de las combinaciones de: no acceso, accesos de sólo lectura y accesos de lectura-escritura, con los cuatro niveles de seguridad. Por ejemplo, 1001 significa acceso de lectura-escritura para procesos de nivel ejecutivo y núcleo, accesos de lectura para procesos de nivel supervisor y no accesos para procesos de nivel usuario. Además, para aislar estos cuatro niveles, cada uno tiene su propia pila y su propia copia del puntero de pila (R14).

La primera implementación de esta arquitectura fue el VAX-11/780, que emplea un TLB para reducir el tiempo de traducción de direcciones. La Figura 8.28 muestra los parámetros clave de este TLB.

La Figura 8.29 muestra la organización del TLB del VAX-11/780, con cada paso de traducción etiquetado. El TLB utiliza ubicación asociativa por conjuntos de dos vías, por tanto, la traducción comienza (pasos 1 y 2) enviando una parte de la dirección virtual («índice») a ambos bancos para seleccionar las dos etiquetas que se van a comparar. Por supuesto, la etiqueta debe estar marcada «válida» para permitir una comparación. Al mismo tiempo, se comprueba el tipo de acceso a memoria para detectar posibles violaciones (también en el paso 2) utilizando a la información de protección del TLB.

Por razones similares a las del caso cache, no hay necesidad de incluir los 9 bits de desplazamiento de página VAX en el TLB; ni hay razón para incluir los 6 bits de dirección para indexar el TLB. Los bits restantes se utilizan en las comparaciones (paso 3). La etiqueta de la dirección coincidente envía la dirección física correspondiente a través del multiplexor (paso 4). El desplazamiento de página se combina entonces con la estructura de la página física para formar una dirección física completa (paso 5).

Hay una característica inusual del TLB del VAX-11/780: el TLB está además subdividido para asegurar que la parte de direcciones de proceso de la dirección no ocupa más del 50 por 100 de las entradas del TLB. Las 32 entradas

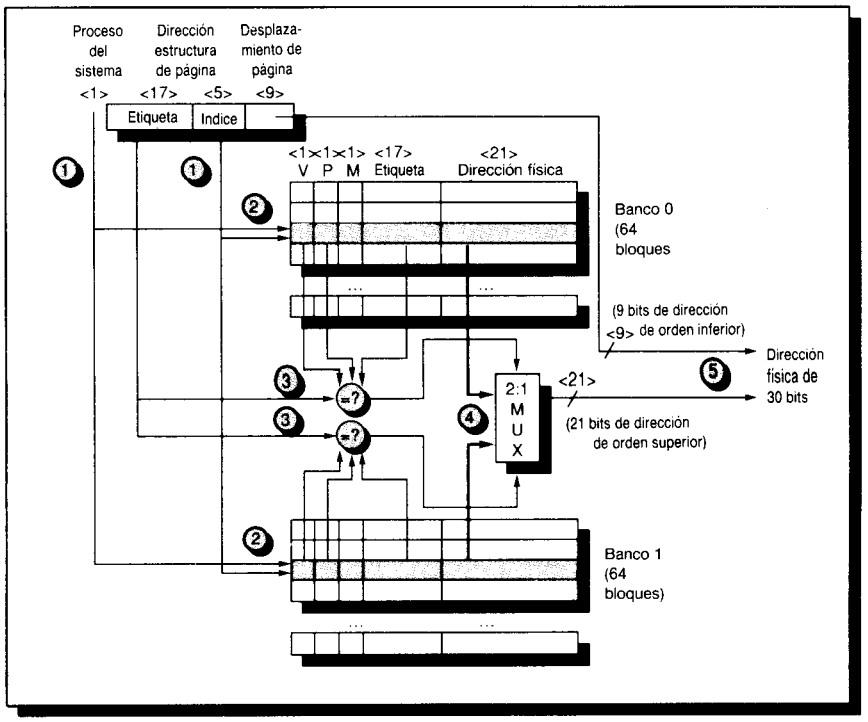


FIGURA 8.29 Operación del TLB de la VAX-11/780 durante la traducción de direcciones. Los cinco pasos de un acierto de TLB se muestran como números inscritos en círculos.

superiores de cada banco están reservadas para el espacio del sistema y las 32 inferiores para el espacio del proceso. El bit más significativo de la dirección se utiliza para seleccionar la mitad apropiada del TLB (paso 1). Como la parte del sistema del espacio de direcciones es la misma para todos los procesos, un cambio de procesos sólo invalida las 32 entradas inferiores de cada banco para el TLB del VAX-11/780. Esta restricción tenía dos objetivos. El primero era reducir el tiempo de cambio de proceso reduciendo el número de entradas del TLB que tenían que invalidarse; el segundo era mejorar el rendimiento evitando que el proceso del sistema o del usuario desechara las traducciones del otro, cuando los cambios de proceso fueran frecuentes. Dividir el TLB habitualmente conducirá a un aumento global de la frecuencia de fallos del TLB, pero puede reducir la frecuencia de fallos máxima del TLB en entornos con intensivos de cambios de proceso.

Un ejemplo de memoria virtual segmentada: protección en el Intel 80286/80386

El segundo sistema es el sistema más peligroso que un hombre jamás haya diseñado... La tendencia general es sobrediseñar el segundo sistema

utilizando todas las ideas y florituras que juiciosamente fueron obtenidas primero.

F. P. Brooks, Jr., *The Mythical Man-Month* [1975]

El 8086 original utilizaba segmentos para el direccionamiento, y no tenía ni memoria virtual ni protección de ningún tipo. Los segmentos tenían registros base pero no registros límite ni comprobaciones de accesos; y antes que un registro segmento se pudiera cargar, el segmento correspondiente debía estar en la memoria física. La dedicación de Intel a la protección y memoria virtual es evidente en modelos posteriores, con algunos campos extendidos para soportar mayores direcciones.

Como el VAX, el 80286 tiene cuatro niveles de protección. El nivel más interno (0) corresponde al modo del VAX, y el nivel más externo (3) corresponde al modo usuario del VAX. El 80286, al igual que el VAX, tiene pilas separadas para cada nivel, con el fin de evitar violaciones de seguridad entre los niveles. También hay estructuras de datos análogas a las tablas de páginas del VAX, que contienen las direcciones físicas para los segmentos, así como una lista de comprobaciones a realizar en las direcciones traducidas.

Los diseñadores de Intel no se detuvieron aquí. El 80286 divide el espacio de direcciones, permitiendo que el sistema operativo y el usuario accedan al espacio completo. El usuario del 80286 puede llamar a una rutina del sistema operativo, en este espacio, e incluso pasar parámetros conservando la protección completa. Esto no es una acción trivial, ya que la pila para el sistema operativo es diferente de la pila del usuario. Sin embargo, el 80286 permite al sistema operativo que mantenga el nivel de protección de la rutina llamada para los parámetros que se le pasan. Este fallo potencial de protección se evita al no permitir que el usuario pida al sistema operativo que acceda indirectamente a alguna cosa que no haya podido acceder por sí mismo. Estos fallos de seguridad se denominan *caballos de Troya.*

Los diseñadores del 80286 estaban guiados por el principio de confiar en el sistema operativo lo menos posible, además de soportar protección y compartición. Como ejemplo del uso de este comportamiento protegido, suponer un programa de nóminas que rellena cheques y también actualiza información hasta la fecha sobre el salario total y pagas de beneficios. Por ello, queremos dar al programa la posibilidad de leer el salario e información hasta la fecha y modificar la información hasta la fecha, pero no el salario. Dentro de poco veremos el mecanismo que soporta estas características. En el resto de esta sección echaremos un vistazo a la protección del 80286 y examinaremos su motivación. Los lectores interesados en los detalles pueden encontrarlo en un libro comprensivo de Crawford y Gelsinger [1987].

Añadir comprobación de límites y correspondencia de memoria

El primer paso para mejorar el 80286 fue conseguir el direccionamiento segmentado para comprobar límites además de suministrar una base. En lugar de una dirección base, como en el 8086, los registros de segmento del 80286

contienen un índice para una estructura de datos de la memoria virtual denominada *tabla de descriptores.* Las tablas de descriptores juegan el papel de las tablas de páginas en el VAX. En el 80286 el equivalente de una entrada de la tabla de páginas es un *descriptor de segmento.* Contiene campos también encontrados en la PTE:

Un *bit de presente,* equivalente al bit de validez de la PTE, utilizado para indicar una traducción válida

Un *campo base,* equivalente a una dirección de estructura de página, que contiene la dirección física del primer byte del segmento

Un *bit de acceso,* como el bit de referencia o de uso de algunas arquitecturas, que es útil para algoritmos de reemplazo

Un *campo de atributo,* como el campo de protección de la PTE del VAX, que especifica las operaciones válidas y niveles de protección para las operaciones que utilizan este segmento

Hay también un *campo límite,* que no se encuentra en los sistemas con paginación, que establece el límite superior de desplazamientos válidos para este segmento. La Figura 8.30 muestra ejemplos de descriptores de segmento del 80286.

Añadir compartición y protección

El siguiente paso de los diseñadores de Intel fue proporcionar compartición protegida. Como en el VAX, la mitad del espacio de direcciones es compartido por todos los procesos y la otra mitad es único para cada proceso; estas mitades se denominan *espacio global de direcciones* y *espacio local de direcciones,* respectivamente. Cada mitad se asocia a una tabla de descriptores con el nombre apropiado. Un descriptor que apunte a un segmento compartido se coloca en la tabla de descriptores globales, mientras que un descriptor para un segmento privado se coloca en la tabla de descriptores locales.

Un programa carga un registro de segmento del 80286 con un índice para la tabla y un bit que indica la tabla que desea. La operación se comprueba de acuerdo con los atributos del descriptor, la dirección física se forma sumando el desplazamiento de la CPU a la base del descriptor, con tal de que el desplazamiento sea menor que el campo límite. De forma distinta a la codificación de operaciones y niveles del PTE de la VAX, cada descriptor de segmentos tiene un campo separado, de dos bits, para dar el nivel de acceso legal de este segmento. Se presenta una violación sólo si el programa intenta utilizar un segmento con un nivel de protección más bajo en el descriptor de segmento.

Ahora podemos mostrar cómo invocar el programa de nóminas para actualizar la información hasta la fecha sin actualizar los salarios. El programa asociaría un descriptor a la información que tiene a cero el campo «escribible», significando que puede leer pero no escribir el dato. Entonces se puede suministrar un programa de confianza que sólo escriba la información hasta la fecha y se da un descriptor con el campo escribible a 1 (ver Fig. 8.30). El

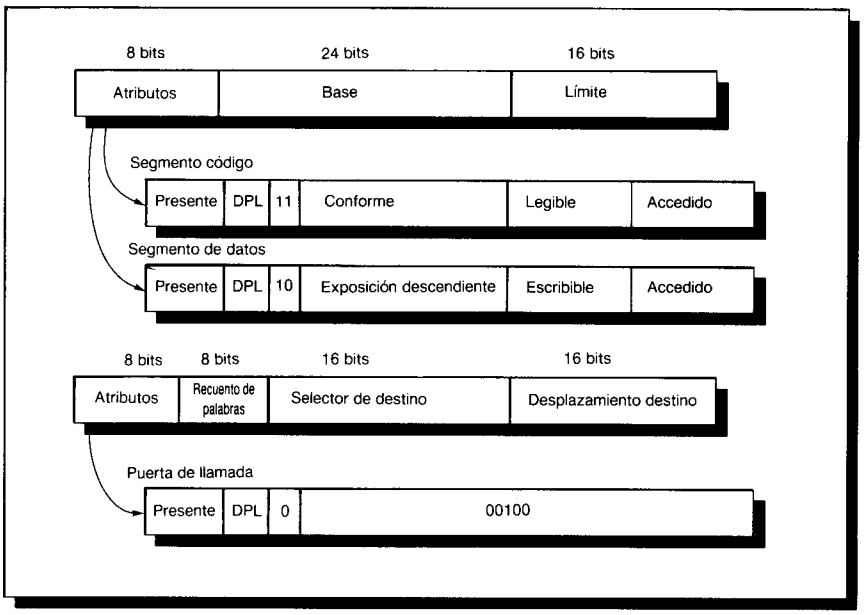


FIGURA 8.30 Los descriptores de segmento del 80286 son todos de 48 bits y se distinguen por los bits del campo de atributos. *Base, límite, presente, legible y escribible* son autoexplicativos. DPL significa *nivel de privilegio del descriptor* —éste se comprueba con el nivel de privilegio del código para ver si se permitirá el acceso. *Conforme* indica que el código toma el nivel de privilegio del código que se está llamando en lugar del nivel de privilegio del llamador; se usa para rutinas de biblioteca. El campo *de expansión descendente* cambia la comprobación para permitir que el campo base sea la marca superior y el campo límite sea la marca inferior. Como se puede esperar, éste se utiliza para segmentos de pilas de crecimiento descendente. *Cuenta de palabras* controla el número de palabras copiadas desde la pila actual a la nueva pila en una puerta de llamada. Los otros dos campos del descriptor de puertas de llamada, *selector de destino y desplazamiento de destino,* seleccionan el descriptor del destino de la llamada y el desplazamiento. Además de estos tres descriptores de segmento hay muchos más en el 80286. El cambio principal en el 80386 fue alargar la base en ocho bits y el límite en cuatro bits.

programa de nóminas invoca el código de confianza, utilizando un descriptor de segmento de código con el campo «conforme» a 1. (Fig. 8.30). Esto significa que el programa llamado toma el nivel de privilegio del código que se está llamando en lugar del nivel de privilegio del que llama. Por consiguiente, el programa de nómina puede leer los salarios y llamar a un programa de confianza para actualizar los totales hasta la fecha, aunque el programa de nóminas no pueda modificar los salarios. Si existe un caballo de Troya en este sistema, para que sea efectivo debe estar localizado en el código de confianza cuya única tarea es actualizar la información hasta la fecha. El argumento para este estilo de protección es que, limitando el alcance de la vulnerabilidad, se mejora la seguridad.

Añadir llamadas seguras del usuario a las puertas del OS y heredar nivel de protección para parámetros

Permitir que el usuario bifurque al sistema operativo es un paso audaz. Entonces, ¿cómo puede un diseñador de hardware aumentar la seguridad de un sistema sin confiar en el sistema operativo o en cualquier otra parte del código? El enfoque del 80286 es restringir en qué puntos del código el usuario puede entrar, colocar parámetros sin peligro en la pila adecuada y asegurar que los parámetros del usuario no adquieren el nivel de protección del código llamado.

Para restringir entradas en códigos de otros, el 80286 proporciona un descriptor de segmentos especial, o *puerta de llamada (call gate),* identificado por un bit en el campo de atributos. De forma distinta a otros descriptores, las puertas de llamada son direcciones físicas completas de un objeto de memoria; el desplazamiento proporcionado por la CPU se ignora. Como indicábamos antes, su propósito es impedir al usuario bifurcar aleatoriamente a cualquier parte de un segmento de código protegido o más privilegiado. En nuestro ejemplo de programación, esto significa que el único sitio donde el programa de nóminas puede invocar el código de confianza está en el propio límite. Esto es necesario para hacer que los segmentos con el campo «conforme» funcionen como se pensó.

¿Qué ocurre si llamador y llamado son «mutuamente sospechosos», ya que no confían entre sí? La solución se encuentra en el campo de número de palabras del descriptor inferior de la Figura 8.30. Cuando una instrucción de llamada invoca un descriptor de puerta de llamada, el descriptor copiará el número de palabras especificadas en el descriptor desde la pila local a la pila que corresponde al nivel de este segmento. Esto permite que el usuario pase parámetros introduciéndolos primero en la pila local. El hardware los transfiere entonces sin peligro a la pila correcta. Un retorno de una puerta de llamada sacará los parámetros de ambas pilas y copiará los valores de retorno en la pila adecuada.

Esto deja todavía abierto el resquicio potencial de que el sistema operativo tenga que utilizar la dirección del usuario, pasada como parámetro, con el nivel de seguridad del sistema operativo, en lugar de con el nivel del usuario. El 80286 resuelve este problema dedicando dos bits en cada registro segmento de la CPU para el *nivel de protección requerido (requested protection level).* Cuando se invoca una rutina del sistema operativo, ésta puede ejecutar una instrucción que inicialice este campo de dos bits, en todos los parámetros de dirección, con el nivel de protección del usuario que llamó a la rutina. Por tanto, cuando estos parámetros de dirección se carguen en los registros segmento, inicializarán el nivel de protección requerido al valor adecuado. El hardware del 80286 utiliza entonces el nivel de protección requerido para evitar cualquier imprudencia: desde las rutinas del sistema no se puede acceder a ningún segmento utilizando esos parámetros si tiene un nivel de protección de más privilegio que el requerido.

Resumen: protección del VAX frente a la del 80286

Si el modelo de protección del 80286 parece más difícil de construir que el del VAX, es porque es así. Este esfuerzo debe ser especialmente frustrante para los ingenieros del 80286, ya que muchos clientes utilizan el 80286 como un 8086 rápido y no explotan el elaborado mecanismo de protección. También, el hecho de que el modelo de protección no sea apropiado para la sencilla protección de paginación de UNIX significa que será utilizado sólo por alguien que escriba un sistema operativo especialmente para este computador. OS/2 de Microsoft es el mejor candidato, pero sólo el tiempo dirá si el coste del rendimiento de esta protección está justificado para un sistema operativo de computadores personales. Quedan dos cuestiones: ¿Será bien utilizado el considerable esfuerzo que debe realizar la ingeniería de protección para cada generación de la familia 80x86? y ¿se demostrará, en la práctica, qué es más seguro que un sistema de paginación?

8.7 Más optimizaciones basadas en el comportamiento de los programas

Hacer el caso frecuente rápido es la inspiración de casi todos los inventos dedicados a mejorar el rendimiento. En esta sección se dan dos ejemplos más de hardware optimizado para el comportamiento del programa. El primero busca instrucciones antes que se necesiten y el segundo evita guardar registros en memoria en las llamadas a procedimientos.

Buffers de prebúsqueda de instrucciones

Muchas máquinas utilizan un *buffer de prebúsqueda de instrucciones* para aprovechar la ejecución secuencial normal de las instrucciones. Normalmente, un buffer de instrucciones contiene de dos a ocho instrucciones secuenciales; cuando la CPU consume una instrucción, se prebusca una palabra de la siguiente instrucción. Prebuscar solamente tiene sentido si el sistema de memoria puede suministrar instrucciones con más rapidez que la CPU las pueda consumir; en cualquier otro caso, el buffer no puede adelantarse a la CPU. Esto se puede conseguir teniendo un camino más ancho que cada vez busque más de una instrucción, o, simplemente, teniendo un sistema de memoria más rápido que la CPU. El inconveniente de los buffers de instrucciones es que incrementa el tráfico de memoria al requerir palabras de instrucciones que pueden no ser necesarias para la CPU, como ocurre cuando se realiza una bifurcación. Los buffers de búsqueda de instrucciones también son útiles para alinear instrucciones de tamaño variable.

El buffer de prebúsqueda de instrucciones de 8 bytes (IB) del VAX-11/780, mostrado en la Figura 8.31, servirá como ejemplo. El código de operación de la instrucción actual está en el byte de orden superior del IB; cuando se con-

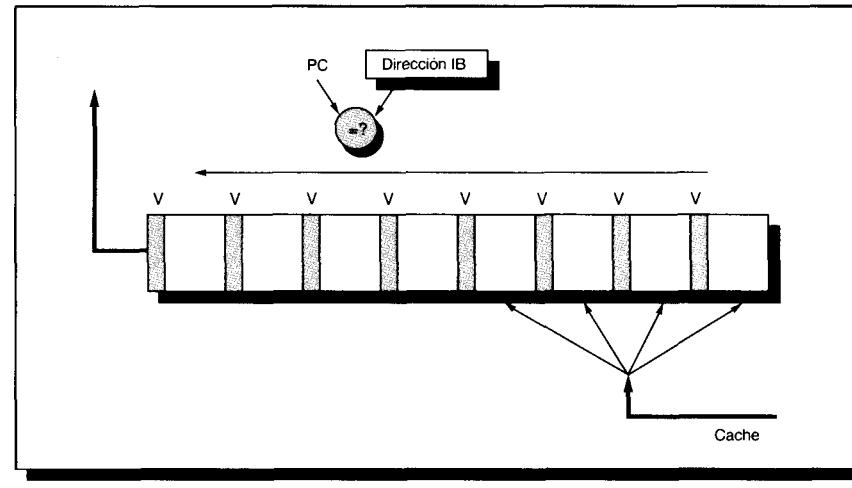


FIGURA 8.31 El buffer de prebúsqueda de instrucciones del VAX-11/780. Cada byte tiene un bit de validez para determinar el número de bytes consecutivos que tienen instrucciones válidas. El decodificador de instrucciones puede leer los cuatro bytes superiores del buffer en un solo ciclo de reloj.

sume parte de la instrucción, el buffer completo es desplazado a la izquierda en la cantidad adecuada. El byte de orden superior puede corresponder a cualquier dirección de byte, aunque los restantes bytes del IB deben ser secuenciales. Las V en la figura representan un bit de validez por byte del buffer de instrucción e indican los bytes secuenciales que contienen instrucciones válidas.

El IB trata de ir adelantado respecto al PC. Siempre que esté libre, como mínimo, un byte en el IB, se realiza una lectura de una palabra alineada de 32 bits que contenga ese byte; sólo se prebuscan de la memoria palabras de 32 bits. Cuando llega la palabra prebuscada de 32 bits, se almacena en el IB una parte de ella dependiendo del espacio disponible. Una palabra de instrucción de 32 bits necesita, por tanto, entre una y cuatro búsquedas de memoria, dependiendo de la suerte.

Cuando el PC cambia debido a un salto o interrupción, el IB puede haber prebuscado una o dos instrucciones innecesarias. Los cambios del PC hacen que todos los bits de validez se pongan a cero y se recargue el IB. La Sección 8.9 examina el impacto del IB sobre el rendimiento.

Registros y ventanas de registros

Las Figuras 3.28 y 3.29 del Capítulo 3 muestran que, guardar registros en las llamadas a los procedimientos y restaurarlos en los retornos, puede contabilizar del 5 al 40 por 100 de las referencias de datos a memoria. Como alternativa, pueden utilizarse varios bancos de registros, asignando uno nuevo en cada llamada. Aunque de esta forma se podría limitar la profundidad de las llamadas a los procedimientos, la limitación se evita operando con los bancos

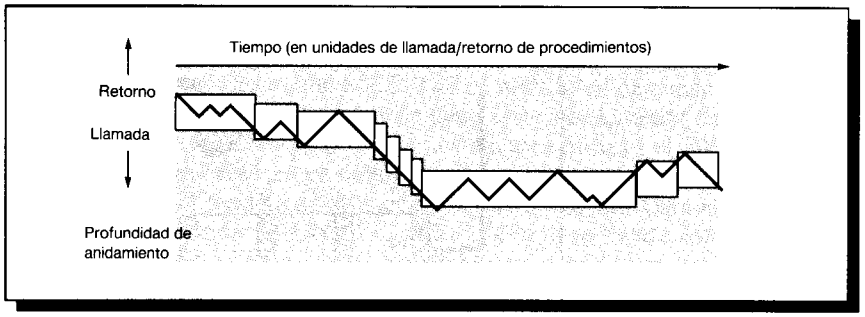


FIGURA 8.32 Cambio con el tiempo de la profundidad de anidamiento de procedimientos. Las cajas muestran llamadas y retornos de procedimientos dentro del buffer antes de un «desbordamiento» o «desbordamiento a cero» de ventanas. El programa comienza con tres llamadas, un retorno, una llamada, un retorno, tres llamadas, y, después, un desbordamiento de las ventanas.

como con un buffer circular, proporcionando una profundidad ilimitada. Esta técnica se ha denominado *ventanas de registros.*

La Figura 8.32 muestra la esencia de la idea. El eje x es el tiempo, medido en llamadas o retornos de procedimiento; el eje y es la profundidad o anidamiento de las llamadas de procedimiento. Cada llamada desplaza el eje y hacia abajo, y cada retorno hacia arriba. Las cajas ponen de manifiesto cuándo debe accederse para guardar alguno de los buffers, bien cuando están completos y a continuación se ejecuta una llamada [desbordamiento *(overflow)* de las ventanas] o cuando están vacíos y se ejecuta un retorno [desbordamiento a cero *(underflow)* de ventana]. La figura muestra ocho desbordamientos de las ventanas y dos desbordamientos a cero durante esta sección de la ejecución de un programa. Durante la vida del programa se igualará el número de desbordamientos y desbordamientos a cero.

También nos podemos preguntar la relación que existe entre el tamaño de los buffers y los dos tipos de desbordamiento. La Figura 8.33 muestra la forma de la curva para diversos programas escritos en diferentes lenguajes de programación. El codo de la curva parece que es de seis a ocho bancos. Aunque esto se cumple para muchos programas, el tamaño óptimo depende de los patrones de llamadas y retornos específicos de cada programa, que pueden ser bastante diferentes en otros programas. El peor caso para las ventanas de registros sería el de cientos de llamadas seguidas por cientos de retornos. Esto haría que la Figura 8.32 pareciese la gráfica de un sismógrafo durante un terremoto ¡y el impacto del rendimiento sería igual de devastador!

La dificultad de pasar parámetros en los registros presenta un inconveniente: si cada procedimiento tiene su propio conjunto único de registros, entonces no hay nada común. Esto puede superarse solapando los bancos o ventanas de registros, de forma que haya una área común en la que pasar parámetros. La Figura 8.34 muestra uno de esos diseños. Seis registros de cada ventana se solapan; los registros R15-R10 del llamador se convierten en R31-R26 después de la llamada. En las ventanas no están incluidos 10 registros,

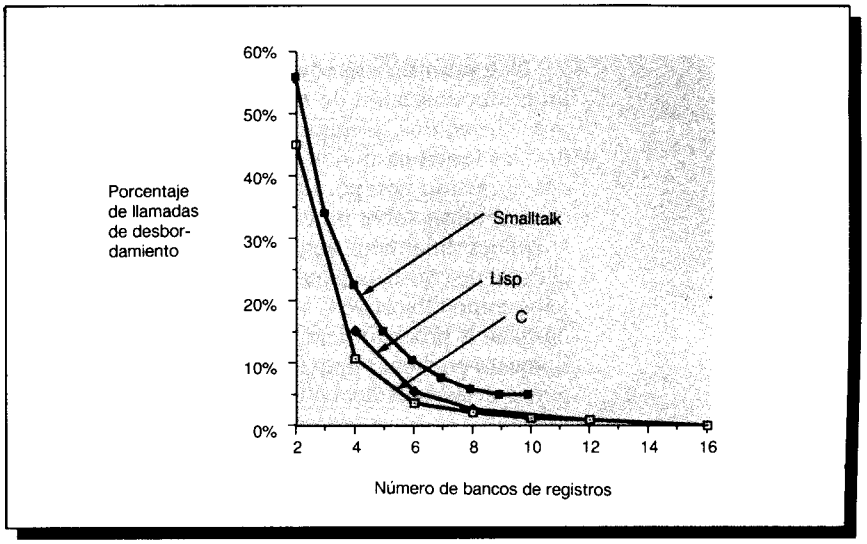


FIGURA 8.33 Número de bancos o ventanas de los registros frente a la frecuencia de desbordamiento para varios programas en C, LISP, y Smalltalk. Los programas medidos para C incluyen un compilador C, un intérprete de Pascal, un programa de ordenación y algunas utilidades UNIX [Halbert y Kessler 1980]. Las medidas LISP incluyen un simulador de circuitos, un demostrador de teoremas, y algunos pequeños benchmarks de LISP [Taylor y cols. 1986]. Los programas de Smalltalk provienen de los benchmarks de macros de Smalltalk [McCall 1983] que incluyen un compilador, «ojeador» *(browser)* y decompilador [Blakken 1983 y Ungar 1987].

por tanto, hay 16 (32 – 10 – 6) registros únicos por ventana aun cuando cada procedimiento vea 32 registros a la vez.

A partir de la Figura 8.33 podemos estimar el porcentaje de llamadas que provocan desbordamiento en las ventanas o de retornos que provocan desbordamientos a cero en ellas, pero para comprender el impacto sobre el rendimiento debemos conocer el coste de ambos desbordamientos. Con un diseño de solapamiento de registros, como el del SPARC, el coste es guardar 16 registros en un desbordamiento (o restaurar 16 registros en un desbordamiento a cero) más el coste de la interrupción. Hoy día en la Sun 4 se emplean alrededor de 60 ciclos de reloj para cualquier tipo de desbordamiento o desbordamiento a cero.

Los pros y contras de las ventanas de registros

Dependiendo de la aplicación, lenguaje de programación y prácticas del usuario, el compilador puede reducir la diferencia entre máquinas con o sin ventanas de registros. Por ejemplo, muchas máquinas tienen registros separados de punto flotante, que significa que los programas intensivos de punto flotante no estarán afectados por las ventanas de registros. Además, muchas re-

ferencias de datos son a objetos que no pueden ser ubicados en registros, como arrays o estructuras (ver Figs. 3.28 y 3.29 del Cap. 3).

Una optimización denominada *ubicación interprocedural de registros* permite una ubicación de registros más inteligente más allá de los límites de los procedimientos. Desgraciadamente, la ubicación de los registros interprocedurales funciona mejor cuando los procedimientos son compilados o enlazados al mismo tiempo. La gran duración de enlace y compilación no coincide con el énfasis sobre un ciclo rápido de depuración-edición-compilación en los lenguajes dinámicos actuales como LISP y Smalltalk. La ubicación interprocedural de registros, generalmente, no es aplicable a lenguajes orientados a objetos como Objective C y Smalltalk porque, en el equivalente dinámico de una llamada a procedimiento, el compilador no conoce el procedimiento que se invocará en dichas llamadas. Las ventanas de registros también simplifican algunas decisiones del compilador, ya que no hay coste extra en utilizar un registro que no se guardará o restaurará separadamente.

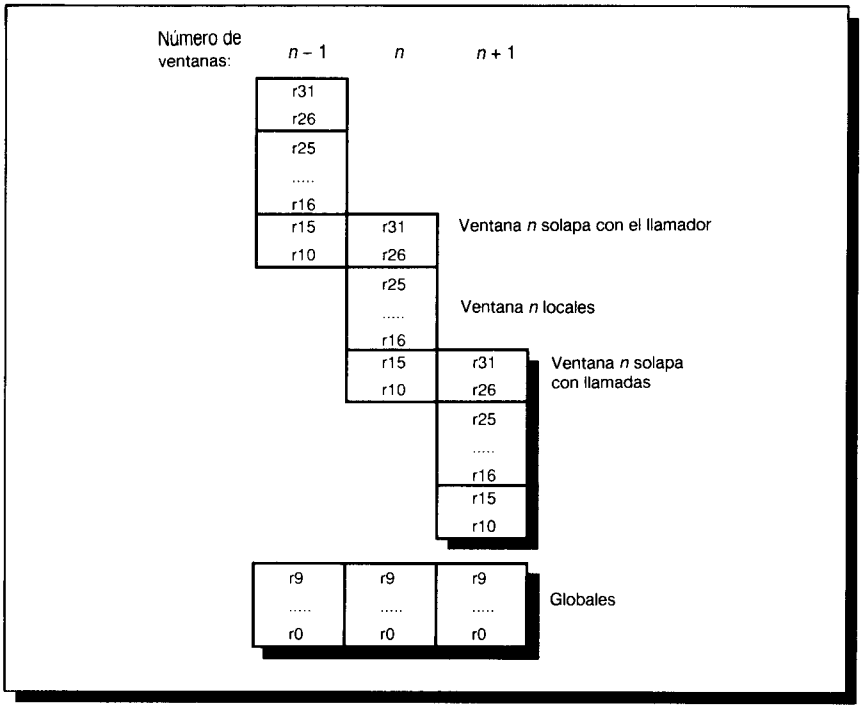


FIGURA 8.34 Los parámetros pueden ser pasados en registros si hay registros comunes entre dos bancos o ventanas. Este esquema divide los registros en globales, que no cambian en una llamada a procedimiento, y locales, que cambian. Teniendo un solapamiento entre los locales para llamadas a procedimientos adyacentes y renumerando los registros en una llamada, los parámetros salientes del llamador se convierten en parámetros entrantes del llamado. Por ejemplo, un valor colocado en el registro 15 antes de una llamada esta en el registro 31 después de la llamada.

	GCC	TeX
Porcentaje de instrucciones de llamada o retorno en DLX	1,8 %	3,6 %
Registros almacenados por llamadas	2,3	3,2
Cargas DLX	3 928 710	2 811 545
Cargas SPARC	3 313 317	2 736 979
Relación de cargas DLX/SPARC	1,20	1,03
Almacenamientos DLX	2 037 226	1 974 078
Almacenamientos SPARC	1 246 538	1 401 186
Relación de almacenamientos DLX/SPARC	1,60	1,41

FIGURA 8.35 Beneficios de las ventanas de registros en cargas y almacenamientos para programas sin punto flotante. La primera fila muestra el porcentaje de instrucciones de DLX ejecutadas que son llamadas o retornos. La segunda fila muestra el número medio de registros salvados y restaurados por llamada en la arquitectura DLX con nivel de optimización O2. Las filas siguientes muestran el número total de cargas y almacenamientos para cada optimización y para la arquitectura SPARC, que tiene ventanas de registros. Estos datos siguientes incluyen las cargas y almacenamientos debidos a desbordamiento y desbordamiento a cero de las ventanas. GCC ejecuta aproximadamente un 20 por 100 más de cargas y un 60 por 100 más de almacenamientos en DLX que en una máquina con ventanas de registros, mientras que TeX ejecuta aproximadamente un 3 por 100 más de cargas y un 41 por 100 más de almacenamientos. Estos ahorros corresponden a aproximadamente el 7 por 100 del recuento de instrucciones para GCC y el 5 por 100 para TeX. La forma en que esto afecta al rendimiento del sistema de memoria, depende de los detalles del resto de la jerarquía de memoria. La ubicación de registros interprocedurales reduce estas diferencias. Por ejemplo, utilizando la optimización O3 en TeX se reduce el número de cargas de DLX en un 5 por 100 a 2 671 631 y el número de almacenamientos en un 10 por 100 a 1 791 831. Observar que las entradas de estos programas no fueron las mismas que las utilizadas en los Capítulos 2 o 4. (Spice no se incluyó porque las ventanas de registros no ofrecían ningún beneficio para programas de punto flotante.)

El peligro de las ventanas de registros es que el mayor número de registros podría ralentizar la frecuencia de reloj. Hasta ahora, éste no ha sido el caso de las máquinas comerciales. La arquitectura SPARC (con ventanas de registro) y la arquitectura MIPS R2000 (sin) son máquinas contemporáneas construidas en varias tecnologías. La frecuencia de reloj del SPARC no ha sido más lenta que la del MIPS para implementaciones en tecnologías similares, probablemente porque los tiempos de acceso a la cache dominan a los tiempos de acceso a los registros en las implementaciones de una u otra arquitectura. Un segundo aspecto es el impacto de las ventanas de registros en el tiempo de cambio de procesos. Sun Microsystems ha encontrado que las peculiaridades del sistema operativo UNIX dominan el tiempo de cambio de procesos, y menos del 20 por 100 del tiempo de cambio de procesos se emplea en guardar o restaurar registros. La Figura 8.35 compara algunas medidas sobre los beneficios de las ventanas de registros en nuestros programas de benchmark.

8.8 Tópicos avanzados. Mejora del rendimiento de memoria cache

Esta sección cubre tópicos avanzados de las memorias cache, exponiendo nuevas ideas a un ritmo mucho más rápido que en las secciones anteriores. Los puntos centrales de este capítulo no se pierden si se salta esta sección; en efecto, la sección de «Juntando todo», que sigue, es independiente de este material.

La creciente separación entre las velocidades de la CPU y memoria principal ha atraído la atención de muchos arquitectos. Después de tomar algunas decisiones fáciles al principio, el arquitecto se enfrenta a un dilema triple cuando intenta reducir el tiempo medio de acceso:

- Incrementar el tamaño de los bloques no mejora el tiempo medio de acceso; la menor frecuencia de fallos no compensa la mayor penalización de los fallos.
- Hacer la cache mayor la haría más lenta, poniendo en peligro la velocidad de reloj de la CPU.
- Hacer la cache más asociativa también la haría más lenta, poniendo en peligro de nuevo la velocidad de reloj de la CPU.

Sin embargo, la frecuencia de fallos calculada a partir de los programas de usuario pinta un cuadro demasiado rosa. La Figura 8.36 muestra la frecuencia de fallos reales de la cache para un programa en ejecución, incluyendo el código del sistema operativo invocado por los programas. Esto revela que el tiempo medio de acceso es peor que el esperado.

Esta sección cubre un amplio conjunto de técnicas para mejorar el rendimiento de la cache: ubicación de subbloques, buffers de escritura, búsqueda fuera de orden, caches direccionadas virtualmente, caches de dos niveles y aspectos relativos a la coherencia del cache. Las secciones de coherencia cache incluyen un ejemplo del problema de datos obsoletos, una visión general de alternativas de coherencia, un ejemplo de protocolo cache, un algoritmo de sincronización utilizado en multiprocesadores con caches coherentes, un diagrama de tiempo mostrando la sincronización de multiprocesadores, y comentarios sobre el impacto de la consistencia de memoria en los procesadores paralelos.

Reducción del tiempo de acierto. Hacer escrituras más rápidas

Como mencionamos antes, las escrituras emplean, habitualmente, más de un ciclo de reloj debido a que la etiqueta se debe comprobar antes de escribir el dato. Hay dos formas para hacer las escrituras más rápidas.

La primera, utilizada en el VAX 8800, segmenta las escrituras para una cache de escritura-directa. Las etiquetas y datos se separan para que se puedan

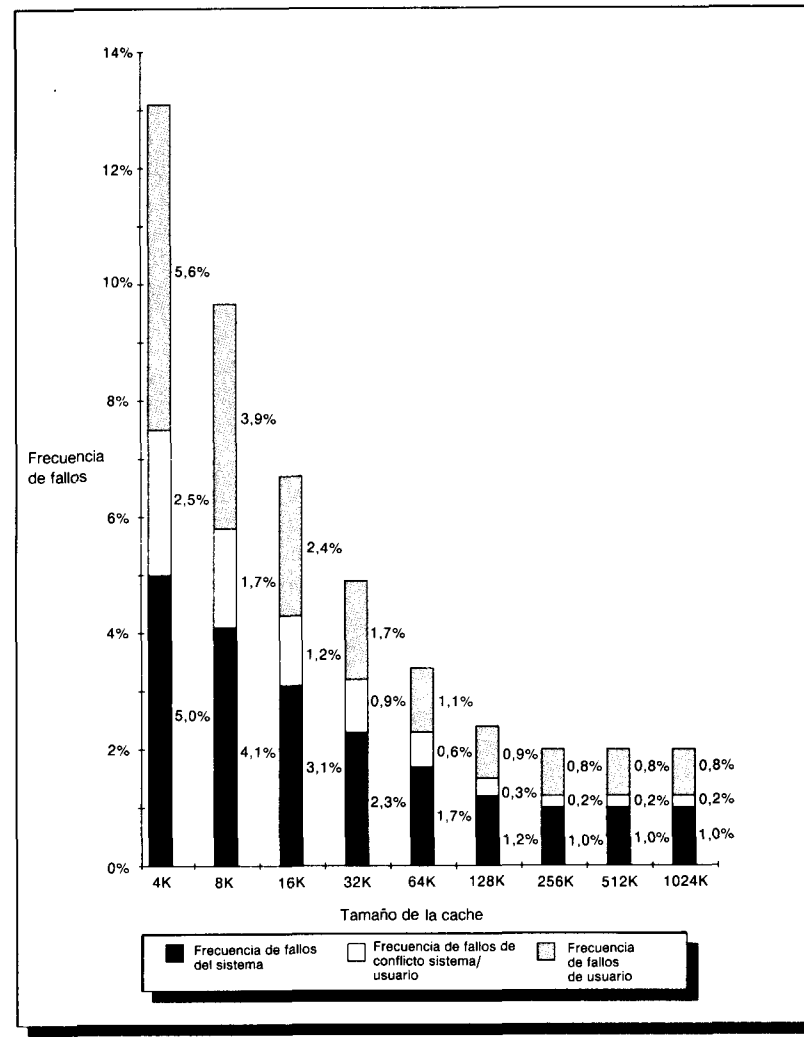


FIGURA 8.36 Frecuencia de fallos de un programa, incluyendo el código del sistema operativo que invoca, frente al tamaño de cache. La categoría superior es lo que se mediría desde una traza de usuario; la categoría inferior es la frecuencia de fallos para el código del sistema operativo; y la categoría intermedia es la frecuencia de fallos debida a conflictos entre el código de usuario y el código del sistema. Agarwal [1987] coleccionó estas estadísticas para el sistema operativo Ultrix corriendo en un VAX, suponiendo caches de correspondencia directa con un tamaño de bloque de 16 bytes.

direccionar independientemente. Como es habitual, la cache compara la etiqueta con la dirección actual de escritura. La diferencia es que el acceso a memoria durante esta comparación utiliza la dirección y los datos de la escritura **previa.** Por tanto, las escrituras se pueden realizar consecutivamente, una por ciclo de reloj, porque la CPU no tiene que esperar que la escritura de la cache

finalice si la primera etapa es un acierto. La segmentación del 8800 no afecta a los aciertos de lectura —la segunda etapa de escritura se realiza durante la primera etapa de la siguiente escritura o durante un fallo de la cache.

Otra forma de reducir las escrituras a un ciclo de reloj involucra caches que deben ser de correspondencia directa, utilizando una técnica conocida como *ubicación de subbloques*. Igual que en el buffer de instrucciones del VAX-11/780, hay un bit de validez en unidades menores que el bloque completo, denominadas *subbloques*. Los bits de validez especifican unas partes del bloque como válidas y otras como inválidas. Una coincidencia de etiqueta no significa que la palabra esté necesariamente en la cache, además los bits de validez para esa palabra deben estar a uno. La Figura 8.37 da un ejemplo. Observar que para caches con ubicación de subbloques, un bloque no se puede seguir definiendo como la mínima unidad transferida entre cache y memoria. Para tales caches, un bloque se define como la unidad de información asociada a una etiqueta de dirección.

La ubicación de subbloques se inventó para reducir la gran penalización de fallos de los bloques grandes (ya que sólo necesita ser leída una parte del bloque grande) y para reducir la memoria de identificadores para pequeñas caches. También puede ayudar en los aciertos de escritura escribiendo **siempre** la palabra (sin importar lo que ocurra con la coincidencia de etiquetas), poniendo el bit de validez a uno, y enviando después la palabra a memoria. Examinemos los casos para ver por qué funciona este truco:

- *Coincidencia de etiquetas y de bit de validez previamente a uno.* Escribir el bloque era la acción adecuada, y no se perdió nada al poner, de nuevo, el bit de validez a uno.

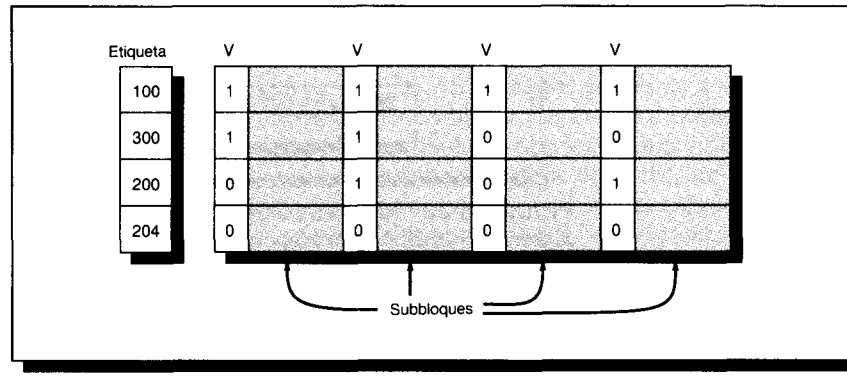


FIGURA 8.37 En este ejemplo hay cuatro subbloques por bloque. En el primer bloque (superior) todos los bits de validez están a 1, lo que equivale a que el bit de validez esté a 1 para un bloque en una cache normal. En el último bloque (inferior), ocurre lo contrario; ningún bit de validez está a 1. En el segundo bloque, las posiciones 300 y 301 son válidas y serán aciertos, mientras que las posiciones 302 y 303 serán fallos. Para el tercer bloque, las posiciones 201 y 203 son aciertos. Si, en lugar de esta organización, hubiera 16 bloques, se necesitarían 16 etiquetas en lugar de 4.

- *Coincidencia de etiquetas y bit de validez previamente no a uno.* La coincidencia de la etiqueta significa que éste es el bloque adecuado; escribir el dato en el bloque hace que sea apropiado poner el bit de validez a uno.
- *No coincidencia de etiquetas.* Esto es un fallo y se modificará la parte de datos del bloque. Sin embargo, como ésta es una cache de escritura directa, no se hace ningún daño; la memoria todavía tiene una copia actualizada del valor anterior. Sólo se necesita cambiar la etiqueta con la dirección de escritura porque el bit de validez ya se ha puesto a uno. Si el tamaño del bloque es de una palabra y la instrucción de almacenamiento está escribiendo una palabra, entonces no hay que hacer nada más. Cuando el bloque es mayor de una palabra o si la instrucción es almacenar un byte o media palabra, entonces o los restantes bits de validez se ponen a cero (ubicando los subbloques sin buscar el resto del bloque) o se pide a memoria que envíe la parte que falta del bloque (ubicar escritura).

Este truco no es posible con caches de postescritura porque la única copia válida del dato puede estar en el bloque, y se sobreescribiría antes de comprobar la etiqueta.

Reducción de la penalización de fallos. Haciendo más rápidos los fallos de escritura

Ahora que hemos visto cómo hacer los aciertos de escritura más rápidos, examinemos los fallos de escritura. Con una cache de escritura directa, la mejora más importante es un buffer de escritura (pág. 447) del tamaño adecuado (ver la falacia de la pág. 520 en la Sección 8.10). Los buffers de escritura, sin embargo, complican las cosas ya que pueden tener el valor actualizado de una posición necesaria en un fallo de lectura.

Ejemplo

Examinar la secuencia de código:

```
SW  512(R0),R3   ; M[512]←R3   (índice de cache 0)
LW  R1,1024(R0)  ; R1←M[1024]  (índice de cache 0)
LW  R2,512(R0)   ; R2←M[512]   (índice de cache 0)
```

Suponer una cache de correspondencia directa que hace corresponder las direcciones 512 y 1024 al mismo bloque, y un buffer de escritura de cuatro palabras. ¿Será R3 siempre igual a R2?

Respuesta

Sigamos la cache para ver el peligro. El dato de R3 se coloca en el buffer de escritura después del almacenamiento. La carga siguiente utiliza el mismo índice de cache y, por tanto, es un fallo. A continuación. tratamos de cargar el dato de la posición 512 en el registro R2; esto también produce un fallo. Si el buffer de escritura no ha terminado de escribir en la posición 512 de memoria, la lectura de la posición 512 pondrá el valor erróneo, antiguo, en el bloque de la cache, y después en R2. Sin las precauciones adecuadas ¡R3 no será igual a R2!

La forma más sencilla de salir de este dilema es que los fallos de lectura esperen hasta que esté vacío el buffer de escritura. Sin embargo, un buffer de escritura de unas pocas palabras en una cache de escritura directa casi siempre tendrá datos en el buffer en un fallo, incrementando así la penalización de fallos de lectura. Los diseñadores del MIPS M/1000 estimaron que, esperando que se vacíe un buffer de cuatro palabras, aumentaría la penalización media de fallos de lectura en un 50 por 100. La alternativa es comprobar el contenido del buffer de escritura en un fallo de lectura, y si no hay conflicto y el sistema de memoria está disponible, permitir que continúe el fallo de lectura.

El coste de las escrituras en una cache de postescritura directa también se puede reducir. Sin más que añadir un buffer de un bloque entero para almacenar un bloque modificado, se puede realizar primero la lectura. Después de que los nuevos datos se carguen en el bloque, la CPU continúa la ejecución. El buffer se escribe entonces en paralelo con la CPU. Análogo a la situación anterior, si se presenta un fallo de lectura, la CPU puede detenerse hasta que el buffer esté vacío.

Reducción de la penalización de fallos. Hacer más rápidos los fallos de lectura

Hacer más rápidas las escrituras es útil, pero son las lecturas las que dominan los accesos a la cache. La estrategia para hacer más rápidos los fallos de lectura es ser impaciente: no esperar que se cargue el bloque completo antes de enviar la palabra requerida a la CPU. Aquí hay dos estrategias específicas:

- Rearranque anticipado. Tan pronto como llegue la palabra requerida del bloque, enviarla a la CPU y permitir que continúe la ejecución.
- *Búsqueda fuera de orden.* Requerir de memoria en primer lugar la palabra que falla y enviarla a la CPU tan pronto como llegue. Permitir que la CPU continúe la ejecución mientras se llena el resto de palabras del bloque. La búsqueda fuera de orden también se denomina búsqueda circular *(wrapped fetch).*

Desafortunadamente, estos trucos de lectura no son tan importantes como parecen. La localidad espacial —la razón para grandes bloques en primer lugar— dicta que la siguiente petición de cache, probablemente, sea para el mismo bloque. Además, manipular otra petición mientras se intenta rellenar el resto del bloque es complicado.

Una razón más sutil por la que la búsqueda fuera de orden no es tan provechosa, como se puede pensar, es que no todas las palabras de un bloque tienen igual probabilidad de que sean accedidas primero. Por ejemplo, con un bloque de 16 palabras en una cache de instrucciones, el punto de entrada medio del bloque es 2,8 palabras a partir del byte de orden superior. Si las entradas estuviesen uniformemente distribuidas, la media sería 8 palabras. La palabra de orden superior es la más probable, debido a los accesos secuenciales desde bloques anteriores en la búsqueda de instrucciones y al recorrido secuencial de los arrays para las caches de datos.

Para las máquinas segmentadas que permiten la terminación fuera de orden utilizando un control de marcador o tipo Tomasulo (Sección 6.7 del Cap. 6), la CPU no necesita detenerse en una fallo de cache, ofreciendo otra forma de reducir las detenciones de memoria. La localidad espacial sugiere que esta optimización (denominada *cache libre de bloqueo)* puede estar limitada en la práctica, ya que de nuevo la referencia siguiente, probablemente, será al mismo bloque.

Hacer más rápidos los aciertos de la cache. Caches direccionadas virtualmente

La penalización de fallos es una parte importante del tiempo medio de acceso, pero el tiempo de aciertos afecta al tiempo medio de acceso y a la velocidad de reloj de la CPU. Ayudando al tiempo de aciertos se puede ayudar, por tanto, a ambas cosas. Una solución, antes mencionada, es utilizar la parte física de la dirección para indexar la cache mientras se envía la dirección virtual a través del TLB. La limitación es que una cache de correspondencia directa puede no ser mayor que el tamaño de página. Para permitir tamaños de cache grandes con páginas de 4 KB en el System/370, IBM utiliza asociatividad alta para que todavía se pueda acceder a la cache con un índice físico. El IBM 3033, por ejemplo, es asociativa por conjuntos de 16 vías, aun cuando hay estudios que muestran que hay poco beneficio en las frecuencias de fallos para asociatividad por conjuntos por encima de 4 vías.

Un esquema para aciertos más rápidos de cache sin esta restricción de tamaño es disponer de accesos a memoria más intensamente segmentados, donde el TLB sea exactamente un paso de la segmentación. El TLB es una unidad distinta más pequeña que la cache y, por tanto, fácilmente segmentada. Este esquema no cambia la latencia de memoria, pero se basa en la eficiencia de la segmentación de la CPU para conseguir un mayor ancho de banda de memoria.

Otra alternativa es utilizar directamente las direcciones virtuales. Estas caches se denominan *caches virtuales.* Esto elimina el tiempo de traducción del TLB en un acierto de la cache. ¿Por qué no construye todo el mundo caches direccionadas virtualmente? Una razón es que, cada vez que cambia de proceso, la dirección virtual referencia diferentes direcciones físicas, requiriendo que se limpie la cache. La Figura 8.38 muestra el impacto en las frecuencias de los fallos de esta limpieza. Una solución es incrementar el ancho de la etiqueta de direcciones de la cache con una *etiqueta identificadora de proceso* (PID). Si el sistema operativo asigna estas etiquetas a procesos, sólo es preciso limpiar la cache cuando se recicle un PID (el PID proporciona protección). La Figura 8.38 muestra esta mejora.

Otra razón por la que las caches virtuales no se adoptan más universalmente, está relacionada con los sistemas operativos y programas de usuario que utilizan dos direcciones virtuales diferentes para la misma dirección física. Estas direcciones duplicadas, denominadas *sinónimas* o *alias,* pueden producir dos copias del mismo dato en una cache virtual; si se modifica una, la otra tendrá un valor erróneo. Con una cache física esto no puede ocurrir, ya que los accesos serían traducidos primero al mismo bloque de la cache fí-

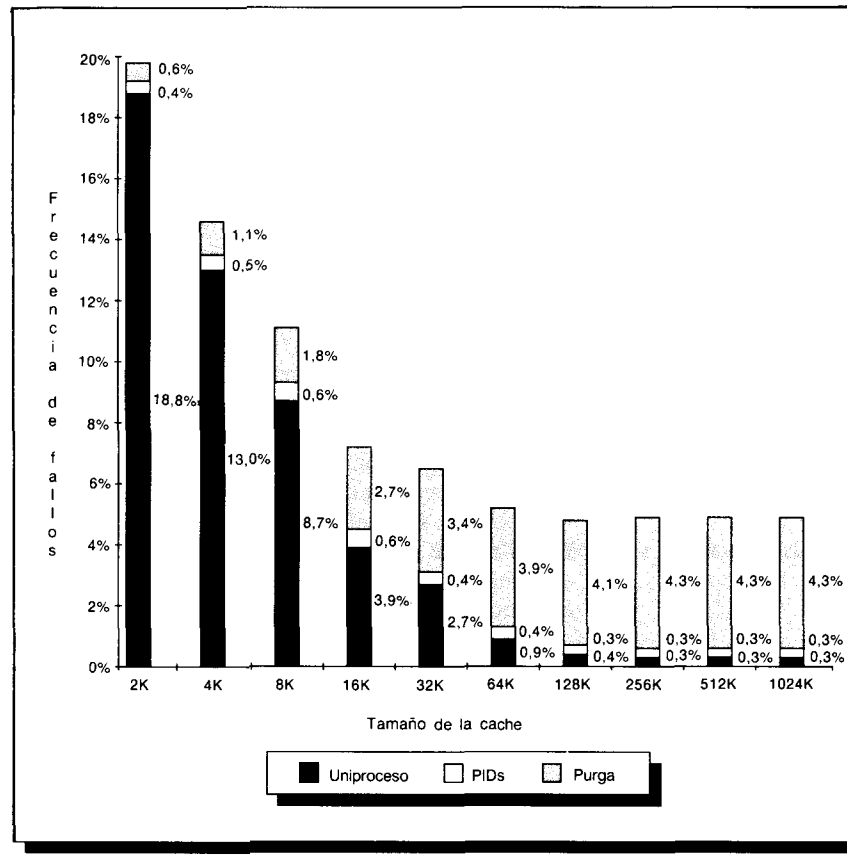


FIGURA 8.38 Frecuencia de fallos frente al tamaño de cache de un programa medido de tres formas: sin cambios de proceso (uniproceso), con cambios de proceso utilizando una etiqueta de identificador de proceso (PID), y con cambios de proceso pero sin PID (purga). Los PID incrementan la frecuencia absoluta de fallos uniproceso de 0,3 a 0,6 y ahorran de 0,6 a 4,3 sobre la purga. Agarwal [1987] obtuvo estas estadísticas para el sistema operativo Ultrix corriendo en un VAX, suponiendo caches de correspondencia directa con un tamaño de bloque de 16 bytes.

sica. Hay esquemas hardware, llamados *antialias,* que pueden garantizar en cada bloque de cache una única dirección física, pero el software puede hacer esto mucho más fácil forzando que los alias compartan algunos bits de dirección. La versión de UNIX de Sun Microsystems, por ejemplo, requiere que todos los alias tengan idénticos los últimos 18 bits de sus direcciones. Por tanto, una cache de correspondencia directa que sea de 2^{18} (256 K) bytes o menor nunca puede tener direcciones físicas duplicadas para bloques. Este requerimiento también simplifica el hardware anti-alias para caches mayores o para caches asociativas por conjuntos. (¡Por supuesto, la mejor solución software desde la perspectiva de los diseñadores de hardware es suprimir los alias!)

El área final de interés sobre las direcciones virtuales son las E/S. Las E/S normalmente utilizan direcciones físicas y, por tanto, requerirán una correspondencia con las direcciones virtuales para interactuar con una cache virtual. (El impacto de la E/S en las caches se discute más adelante.)

Reducción de la penalización de fallos. Caches de dos niveles

Volvamos nuestra atención a la penalización de fallos. Las CPU se están volviendo más rápidas y las memorias principales mayores, pero más lentas con respecto a las CPU más rápidas. La pregunta que se hace el arquitecto es: ¿debería hacerse la cache más rápida para mantenerse a la altura de la velocidad de las CPU, o hacer la cache mayor para superar el creciente desnivel entre la CPU y memoria principal? Una respuesta es: ambas cosas. Al añadir otro nivel de cache entre la cache original y memoria, la cache de primer nivel puede ser bastante más pequeña para que coincida con la duración del ciclo de reloj de la CPU, mientras que la cache de segundo nivel puede ser bastante grande para capturar muchos accesos que irían a la memoria principal.

Las definiciones para un segundo nivel de cache no son siempre sencillas. Comencemos con la definición de *tiempo medio de acceso a memoria* para una cache de dos niveles. Utilizando los subíndices L1 y L2 para referirnos respectivamente, a las caches del primer y segundo nivel, la fórmula original es

$$\text{Tiempo medio de acceso a memoria} = \text{Tiempo acierto}_{L1} + \\ + \text{Frecuencia de fallos}_{L1} \cdot \text{Penalización de fallos}_{L1}$$

y

$$\text{Penalización fallos}_{L1} = \text{Tiempo acierto}_{L2} + \\ + \text{Frecuencia fallos}_{L2} \cdot \text{Penalización fallos}_{L2}$$

por tanto

$$\text{Tiempo medio de acceso a memoria} = \text{Tiempo acierto}_{L1} + \\ + \text{Frecuencia fallos}_{L1} \cdot (\text{Tiempo acierto}_{L2} + \\ \text{Frecuencia fallos}_{L2} \cdot \text{Penalización fallos}_{L2})$$

En esta fórmula, el éxito de la frecuencia de fallos del segundo nivel se mide con las sobras de la cache del primer nivel. Para evitar ambigüedad, adoptaremos estos términos para un sistema cache de dos niveles:

- *Frecuencia local de fallos.* El número de fallos de la cache dividido por el número total de accesos a esta cache; esto es la frecuencia de fallos_{L2} anterior.
- *Frecuencia global de fallos.* El número de fallos de la cache dividido por el número total de accesos a memoria generados por la CPU; utilizando los términos anteriores, esto es frecuencia de fallos_{L1} · frecuencia de fallos_{L2}.

Ejemplo Suponer que en 1 000 referencias a memoria hay 40 fallos en la cache de primer nivel y 20 fallos en la de segundo nivel. ¿Cuáles son las distintas frecuencias de fallos?

Respuesta La frecuencia de fallos para la cache de primer nivel es 40 por 1 000 o 4 por 100. La frecuencia local de fallos para la cache de segundo nivel es 20/40 o 50 por 100. La frecuencia global de fallos de la cache de segundo nivel es 20 por 1 000 o 2 por 100.

La Figura 8.39 y la Figura 8.40 muestran cómo las frecuencias de fallos y el tiempo de ejecución relativo cambian con el tamaño de la cache de segundo nivel. La Figura 8.41 muestra parámetros típicos de las caches de segundo nivel.

Con estas definiciones en mente, podemos considerar los parámetros de las caches de segundo nivel. La diferencia principal entre los dos niveles es que la velocidad de la cache del primer nivel afecta a la frecuencia de reloj de la CPU, mientras que la velocidad de la cache del segundo nivel sólo afecta a la penalización de fallos de la cache del primer nivel. Por tanto, podemos considerar muchas alternativas en la cache del segundo nivel que podrían no ser adecuadas para la cache del primer nivel. Pero hay una consideración para el diseño de la cache del segundo nivel: ¿disminuirá la parte del CPI correspondiente al tiempo medio de acceso a memoria?

La elección inicial para las caches del segundo nivel es el tamaño. Como todo lo de la cache del primer nivel está probablemente en la cache del segundo nivel, ésta debería ser mayor. Si las caches de segundo nivel son sólo un poco mayores, la frecuencia local de fallos será alta. Esta observación inspira diseños de caches enormes del segundo nivel —¡del tamaño de la memoria principal de los recientes computadores!—. Si la cache del segundo nivel es mucho mayor que la del primero, entonces la frecuencia global de fallos es aproximadamente la misma que en una cache mononivel del mismo tamaño (ver Fig. 8.39). Gran tamaño significa que la cache del segundo nivel puede no tener prácticamente fallos de capacidad, dejando para nuestra atención los fallos forzosos y algunos fallos de conflicto. Una pregunta es si la asociatividad por conjuntos tiene más sentido para las caches del segundo nivel.

Ejemplo Dados los datos siguientes, ¿cuál es el impacto de la asociatividad de la cache del segundo nivel en la penalización de fallos?

- Asociatividad por conjuntos de dos vías incrementa el tiempo de aciertos en un 10 por 100 de un ciclo de reloj de la CPU
- Tiempo de aciertos$_{L2}$ para correspondencia directa = 4 ciclos de reloj
- Frecuencia local de fallos$_{L2}$ para correspondencia directa = 25 por 100
- Frecuencia local de fallos$_{L2}$ para asociativa por conjuntos de dos vías = 20 por 100
- Penalización de fallos$_{L2}$ = 30 ciclos de reloj

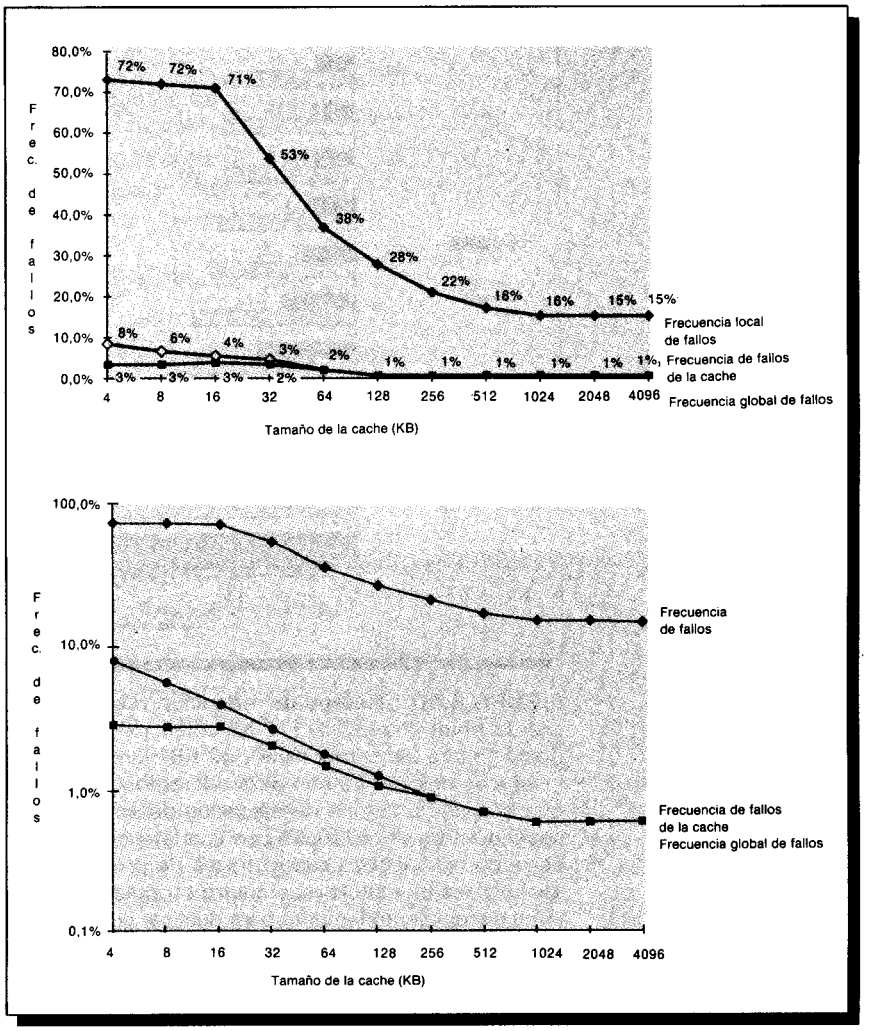


FIGURA 8.39 Frecuencia de fallos frente a tamaño de cache. La gráfica superior muestra los resultados dibujados en una escala lineal como hemos hecho con figuras anteriores, mientras que la gráfica inferior muestra los resultados dibujados en una escala logarítmica. Cuando la frecuencia de fallos disminuye, la escala log hace las diferencias más visibles. Las gráficas muestran la frecuencia de fallos de una cache de un solo nivel junto con la frecuencia local y frecuencia global de fallos de una cache de segundo nivel utilizando una cache de primer nivel de 32 KB. Las caches de segundo nivel **menores** de los 32 KB del primer nivel tienen alta frecuencia de fallos (como mínimo para tamaños similares de bloque), como ilustra esta figura. A partir de 256 KB la cache única y las frecuencias globales de fallos son virtualmente idénticas. Przybylski [1990] coleccionó estos datos utilizando trazas disponibles con este libro: cuatro trazas de programas de sistema y usuario del VAX y cuatro programas de usuario para el MIPS R2000 que fueron aleatoriamente intercaladas para duplicar el efecto de los cambios de procesos.

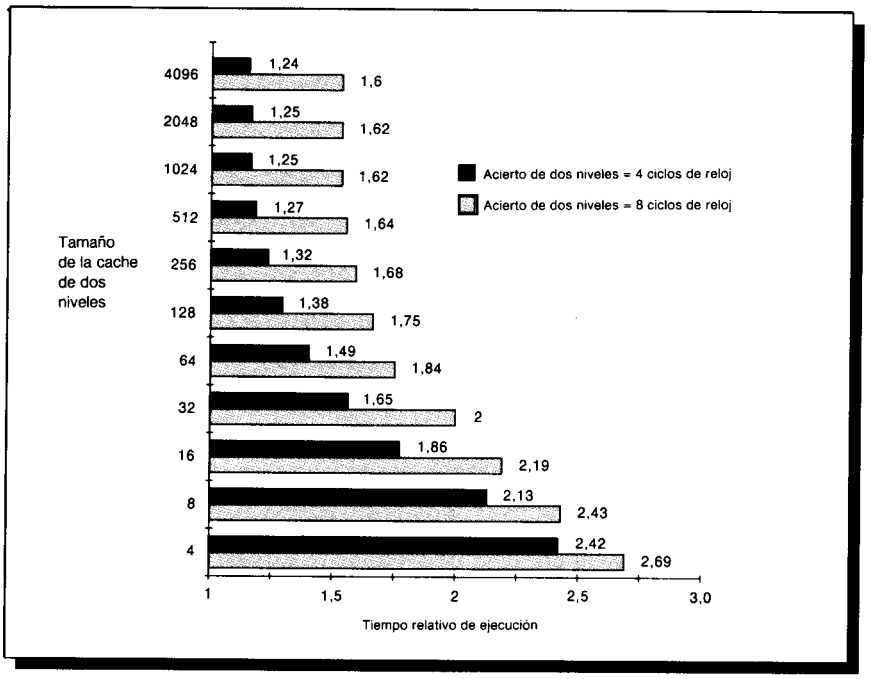


FIGURA 8.40 Tiempo de ejecución relativa para el tamaño de cache de segundo nivel. Przybylski [1990] coleccionó estos datos utilizando una cache de postescritura, de primer nivel y 32 KB, variando el tamaño de la cache de segundo nivel. Las dos barras son para diferentes tiempos de acierto en la cache de segundo nivel. El tiempo de ejecución de referencia de 1,00 es para una cache de segundo nivel de 4096 KB con una latencia de un ciclo de reloj en un acierto de segundo nivel. Utilizó cuatro trazas de programas de usuario y sistema del VAX (disponibles en este libro) y cuatro programas de usuario del MIPS R2000 que se intercalaron aleatoriamente para duplicar el efecto de los cambios de proceso.

Tamaño de bloque (línea)	32-256 bytes
Tiempo de acierto	4-10 ciclos de reloj
Penalización de fallo	30-80 ciclos de reloj
(tiempo de acceso)	(14-18 ciclos de reloj)
(tiempo de transferencia)	(16-64 ciclos de reloj)
Frecuencia local de fallo	15 %-30 %
Tamaño de cache	256 KB - 4 MB

FIGURA 8.41 Valores típicos de los parámetros clave de jerarquía de memoria para caches de segundo nivel.

Respuesta Para una cache de segundo nivel de correspondencia directa, la penalización de fallos de la cache del primer nivel es

$$\text{Penalización de fallos}_{L1} = 4 + 25\,\% \cdot 30 = 11{,}5 \text{ ciclos de reloj}$$

Añadiendo el coste de la asociatividad, se incrementa el coste de acierto sólo en 0,1 ciclos de reloj, haciendo la nueva penalización de fallos de la cache del primer nivel

$$\text{Penalización de fallos}_{L1} = 4{,}1 + 20\,\% \cdot 30 = 10{,}1 \text{ ciclos de reloj}$$

En realidad, las caches del segundo nivel están casi siempre sincronizadas con la del primer nivel y la CPU. De acuerdo con esto, el tiempo de aciertos del segundo nivel debe ser un número integral de ciclos de reloj. Si tenemos suerte, podemos aproximar el tiempo de aciertos del segundo nivel a cuatro ciclos; si no, podemos redondearlo hasta cinco ciclos. Cualquier elección es una mejora sobre la cache del segundo nivel de correspondencia directa:

$$\text{Penalización de fallos}_{L1} = 4 + 20\,\% \cdot 30 = 10{,}0 \text{ ciclos de reloj}$$

$$\text{Penalización de fallos}_{L1} = 5 + 20\,\% \cdot 30 = 11{,}0 \text{ ciclos de reloj}$$

La mayor asociatividad es digna de consideración porque tiene pequeño impacto en el tiempo de aciertos del segundo nivel y porque la mayor parte del tiempo medio de acceso se debe a fallos. Sin embargo, para caches muy grandes los beneficios de la asociatividad disminuyen porque el mayor tamaño ha eliminado muchos fallos de conflictos.

Mientras la localidad espacial mantiene que puede haber un beneficio al incrementar el tamaño de bloque, este incremento puede aumentar los fallos de conflictos en caches pequeñas, ya que puede no haber suficiente sitio para poner datos, incrementando además la frecuencia de fallos. Como esto no es un problema en las caches grandes del segundo nivel, y porque el tiempo de acceso a memoria es relativamente mayor, son populares tamaños mayores de bloque. La Figura 8.42 muestra la variación del tiempo de ejecución cuando cambia el tamaño de bloque del segundo nivel.

Una consideración final atañe a si todos los datos de la cache del primer nivel están siempre en la del segundo nivel. Si es así, se dice que la cache del segundo nivel tiene la *propiedad de inclusión multinivel.* La inclusión es deseable porque la consistencia entre E/S y las caches (o entre caches en un multiprocesador) se puede determinar comprobando la cache del segundo nivel.

El inconveniente a esta inclusión natural es que, el tiempo medio de acceso a memoria más bajo de la cache de primer nivel, puede sugerir bloques más pequeños para esta cache que es más pequeña y bloques mayores para la cache mayor del segundo nivel. La inclusión se puede mantener todavía en este caso con poco trabajo extra en un fallo del segundo nivel: la cache del segundo nivel debe invalidar todos los bloques de primer nivel que correspondan al bloque del segundo nivel que se va a reemplazar, provocando una frecuencia de fallos del primer nivel ligeramente mayor.

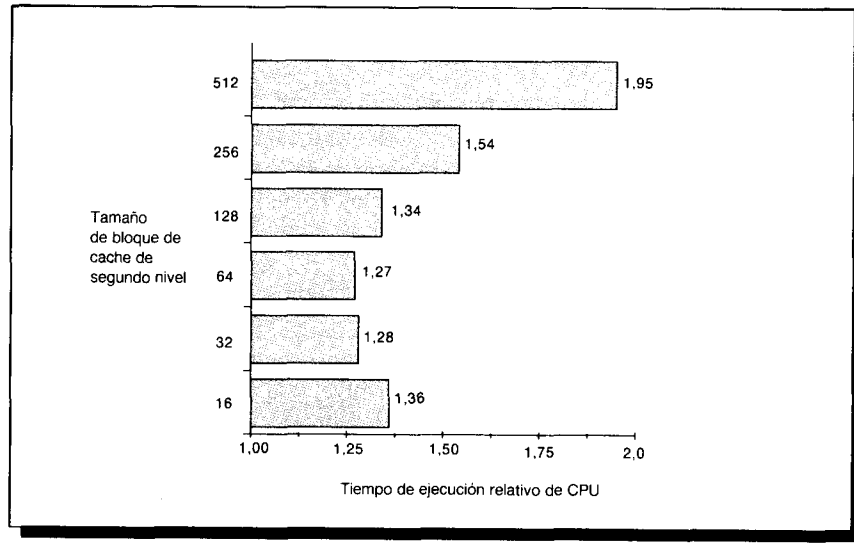


FIGURA 8.42 Tiempo de ejecución relativo por tamaño de bloque para una cache de dos niveles. Przybylski [1990] obtuvo estos datos utilizando una cache de segundo nivel de 512 KB. Utilizó cuatro trazas de programas de usuario y sistema del VAX (disponibles en este libro) y cuatro programas de usuario del MIPS R2000 que se intercalaron aleatoriamente para duplicar el efecto de los cambios de proceso.

Reducción de la frecuencia de fallos reduciendo las limpiezas de cache. E/S

Aunque hay poco más que pueda mejorar el tiempo de ejecución de la CPU, hay cuestiones en el diseño cache que mejoran el rendimiento del sistema, particularmente para entrada/salida. A causa de las caches, los datos se pueden encontrar en memoria o en la cache. Mientras que la CPU es el único dispositivo que cambia o lee los datos y la cache está entre la CPU y memoria, hay poco peligro de que la CPU vea la copia antigua u obsoleta *(stale).* Las E/S significan que existe la oportunidad para que otros dispositivos puedan hacer que las copias sean inconsistentes o para que otros dispositivos lean las copias obsoletas. La Figura 8.43 ilustra el problema. Este se referencia generalmente como problema de *coherencia cache.*

La pregunta es ésta: ¿dónde se llevan a cabo las E/S en el computador —entre el dispositivo de E/S y la cache o entre el dispositivo de E/S y memoria principal? Si la entrada pone datos en la cache y la salida lee datos de la cache, las E/S y la CPU ven los mismos datos, y el problema está resuelto. La dificultad de este enfoque es que interfiere con la CPU. La competencia de las E/S con la CPU para accesos a la cache provocará que la CPU se detenga durante las E/S. La entrada también interferirá con la cache al sustituir alguna información por nuevos datos, que tienen poca probabilidad de ser accedidos por la CPU en un futuro inmediato. Por ejemplo, en un fallo de página, la CPU puede necesitar acceder a algunas palabras de la página, pero no es pro-

bable que un programa acceda a cada palabra de la página si estuviese cargada en la cache.

El objetivo para el sistema de E/S de un computador con cache es evitar el problema de los datos obsoletos, a la vez que interferir con la CPU lo menos posible. Muchos sistemas prefieren, por tanto, que las E/S vayan directamente a memoria principal, que actúa como un buffer de E/S. Si se utiliza una cache de escritura directa, entonces la memoria tiene una copia actualizada de la información y no hay posibilidad de utilizar datos obsoletos en la salida. (Esta es la razón por la que muchas máquinas utilizan escritura directa). Las entradas requieren algún trabajo extra. La solución software es garantizar que ningún bloque del buffer de E/S designado para entrada esté en la cache. En un

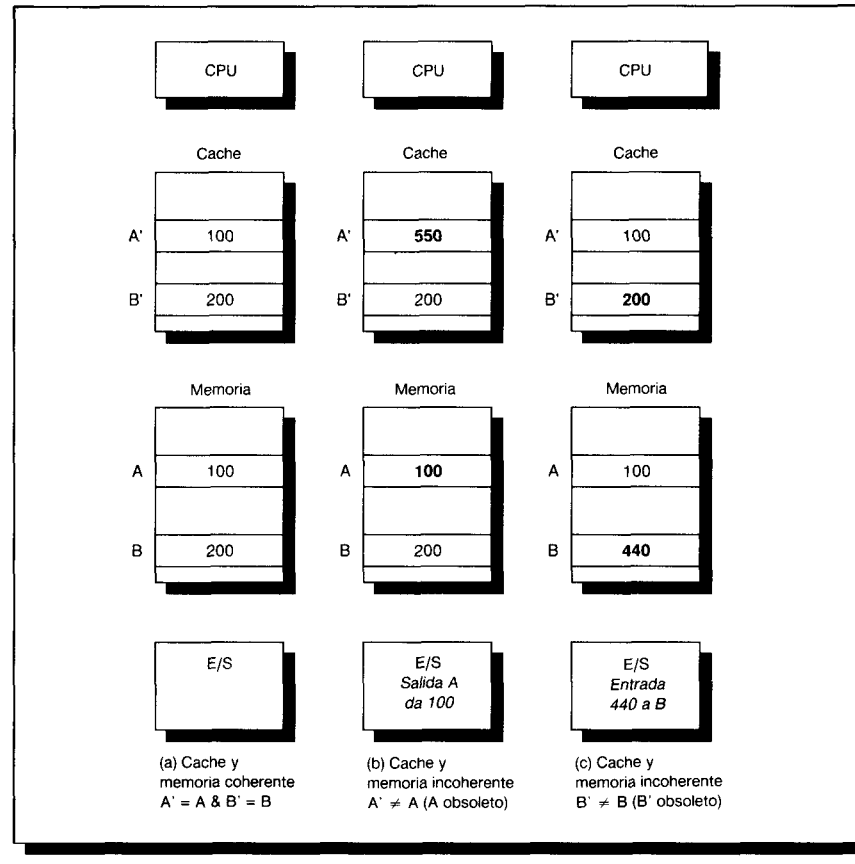


FIGURA 8.43 El problema de coherencia cache. A' y B' referencian las copias de cache de A y B en memoria. a) muestra la memoria cache y principal en un estado coherente. En b) suponemos una cache de postescritura cuando la CPU escribe 550 en A. Ahora A' tiene el valor último, pero el valor de memoria tiene el valor obsoleto de 100. Si una salida usase el valor de A desde memoria, obtendría el dato obsoleto. En c) el sistema de E/S introduce 440 en la copia de memoria de B, así que ahora B' en la cache tiene el dato obsoleto.

primer enfoque, algunas páginas del buffer se marcan como no «cacheables» y el sistema operativo siempre realiza las entradas sobre tales páginas. En otro enfoque, el sistema operativo elimina de la cache las direcciones correspondientes al buffer una vez que se realizó la entrada. Una solución hardware es comprobar las direcciones de E/S en la entrada para ver si están en la cache. Si es así, las entradas de la cache se invalidan para evitar datos obsoletos. Todas estas aproximaciones también se pueden utilizar para la salida con caches de postescritura. Más información, con respecto a esto, se encuentra en el siguiente capítulo.

Reducción del tráfico del bus. Coherencia cache de multiprocesadores

El problema de la coherencia cache se aplica a los multiprocesadores, así como a las E/S. De forma distinta a las E/S, donde copias múltiples de datos es un raro evento —a evitar siempre que sea posible— un programa ejecutándose en múltiples procesadores querrá tener copias del mismo dato en varias caches. El rendimiento de un programa en multiprocesador depende de la eficiencia del sistema para compartir los datos. Los protocolos para mantener la coherencia de múltiples procesadores se denominan *protocolos de coherencia-cache.* Hay dos clases de protocolos para mantener la coherencia cache:

- *Basados en directorio.* La información sobre un bloque de memoria física se mantiene en una única posición.
- *Espionaje (snooping).* Cada cache que tiene una copia de datos de un bloque de memoria física también tiene una copia de la información sobre él. Estas caches se utilizan habitualmente en sistemas de memoria compartida con un bus común; todos los controladores cache vigilan o espían *(snoop)* en el bus para determinar si tienen o no una copia del bloque compartido.

En los protocolos basados en directorio hay lógicamente un único directorio que contiene el estado de cada bloque en memoria principal. La información del directorio puede incluir las caches que tengan copias del bloque, si está modificado *(dirty),* etc. Las entradas al directorio pueden estar distribuidas para que diferentes peticiones puedan ir a diferentes memorias, reduciendo así la contención. Sin embargo, conservan la característica de que el estado compartido de un bloque está siempre en una única sola posición conocida.

Los protocolos de espionaje se hicieron populares con los multiprocesadores que utilizaban microprocesadores y caches, en sistemas de memoria compartida, porque pueden utilizar una conexión física preexistente: el bus a memoria. El espionaje tiene una ventaja sobre los protocolos de directorio, y ésta es que la información de coherencia es proporcional al número de bloques de una cache en lugar de al número de bloques de memoria principal. Los directorios, por otra parte, no requieren un bus que vaya a todas las caches y, por consiguiente, puede resultar en un sistema más escalable (más procesadores).

El problema de coherencia se resume en que un procesador tenga acceso

exclusivo al escribir un objeto y que tenga la copia más reciente al leer un objeto. Por tanto, ambos protocolos, basados en directorios y de espionaje, deben localizar todas las caches que comparten el objeto que se va a escribir. La consecuencia de una escritura en un dato compartido es invalidar las demás copias o difundir la escritura a las copias compartidas. Debido a las caches de postescritura, los protocolos de coherencia también deben ayudar a determinar, en los fallos de lectura, quién tiene el valor más actualizado.

Durante el resto de esta sección nos concentraremos en caches de espionaje; las mismas ideas se aplican a las caches basadas en directorio, excepto que la información sobre el estado de las caches se gestiona de forma diferente, y se involucran sólo si el directorio indica que tienen una copia de un bloque cuyo estado debe cambiar.

A los bits de estado ya existentes en un bloque se añade información sobre compartición para protocolos de espionaje, y esta información se utiliza para vigilar las actividades del bus. En un fallo de lectura, todas las caches comprueban si tienen una copia del bloque requerido y realizan la acción apropiada, como suministrar el dato a la cache donde falló. Análogamente, en una escritura todas las caches comprueban si tienen una copia y actúan entonces, quizá invalidando su copia o cambiándola por el nuevo valor.

Como cada transacción del bus comprueba las etiquetas de dirección de la cache, se puede suponer que interfiere con la CPU. Sería así, si no se duplicase la parte de etiquetas de dirección de la cache (no la cache completa) para obtener un puerto extra de lectura para espiar *(snooping)*. De esta forma, el espionaje interfiere con el acceso de la CPU a la cache sólo cuando hay un problema de coherencia (aunque en un fallo con espionaje la CPU debe arbitrar con el bus el cambio de las etiquetas de espionaje además de las normales). Cuando ocurre una operación de coherencia en la cache se detendrá, probablemente, la CPU, ya que la cache está indisponible. En caches multinivel, si la comprobación de coherencia se puede limitar a la cache inferior a causa de la inclusión multinivel, probablemente no será necesario duplicar las etiquetas de dirección.

Los protocolos de espionaje son de dos tipos, dependiendo de lo que ocurra en una escritura:

- *Invalidación en escritura.* El procesador que escribe hace que se invaliden todas las copias en las otras caches antes de cambiar su copia local. Entonces, es libre de actualizar el dato hasta que otro procesador lo pida. El procesador que escribe, distribuye una señal de invalidación sobre el bus, y todas las caches comprueban si tienen una copia. Si es así, deben invalidar el bloque que contenga la palabra. Por tanto, este esquema permite múltiples lectores pero sólo un escritor.
- *Difusión en escritura.* En lugar de invalidar cada copia del bloque compartido, el procesador que escribe, difunde el nuevo dato sobre el bus; entonces se actualizan todas las copias con el nuevo valor. Este esquema difunde continuamente escrituras para datos compartidos, mientras que el de invalidación en escritura suprime las demás copias para que sólo haya una copia local para escrituras posteriores. Los protocolos de difusión en escritura permiten, habitualmente, que los bloques se identifiquen como com-

partidos (difundidos) o privados (locales). Este protocolo actúa de forma similar a una cache de escritura directa para datos compartidos (difundiendo a otras caches) y como una cache de postescritura para datos privados (los datos modificados salen fuera de la cache sólo cuando se produce un fallo).

La mayoría de los multiprocesadores basados en cache utilizan caches de postescritura porque reducen el tráfico del bus y así se permiten más procesadores sobre un solo bus. Las caches de postescritura utilizan o invalidación o difusión, y existen numerosas variaciones para ambas alternativas (ver sección siguiente). Hasta ahora, no hay consenso sobre cuál es el esquema mejor. Algunos programas tienen menos gastos de coherencia con invalidación de escritura y otros con difusión de escritura. Una sección posterior muestra cómo se puede implementar la sincronización en multiprocesadores basados en coherencia; los accesos para la sincronización parecen favorecer la difusión en escritura.

Una primera intuición ha sido que el tamaño de bloque juega un papel importante en la coherencia cache. Tomar, por ejemplo, el caso de espiar una cache de segundo nivel con un tamaño de bloque de ocho palabras, y que una palabra es escrita y leída, alternativamente, por dos procesadores. Tanto si se utiliza invalidación como difusión en escritura, el protocolo que sólo difunda o envíe una palabra tiene ventaja sobre un esquema que transfiera el bloque completo. Otro aspecto relacionado con grandes bloques se denomina *compartición falsa:* dos variables diferentes compartidas están localizadas en el mismo bloque de cache, haciendo que se transfiera el bloque entre procesadores aun cuando estos estén accediendo a variables diferentes. La investigación sobre compiladores está trabajando para reducir las frecuencias de fallos de cache ubicando datos con alta localidad para determinados procesadores en los mismos bloques. El éxito de este campo puede incrementar el atractivo de grandes bloques para multiprocesadores.

Medidas actuales indican que los datos compartidos tienen menor localidad espacial y temporal que la observada para otros tipos de datos, independientemente de la política de coherencia.

Un protocolo ejemplo

Para ilustrar las complejidades de un protocolo de coherencia cache, la Figura 8.44 muestra un diagrama de transición de estados finitos para un protocolo de invalidación en escritura basado en una política de postescritura. Los tres estados de protocolo están duplicados para que representen transiciones basadas en acciones de la CPU, en contraposición a las transiciones basadas en las operaciones del bus. Esto se hace sólo para esta figura; sólo hay una máquina de estados finitos por cache, con estímulos provenientes o de la CPU conectada o del bus.

Las transiciones se presentan en los fallos de lectura, fallos de escritura o aciertos de escritura; los aciertos de lectura no hacen cambiar el estado de la cache. Cuando la CPU tiene un fallo de lectura, cambia el estado de ese bloque a sólo lectura y postescribe el bloque antiguo si estaba en el estado Lectura/Escritura (modificado). Todas las caches comprueban, a raíz del fallo de

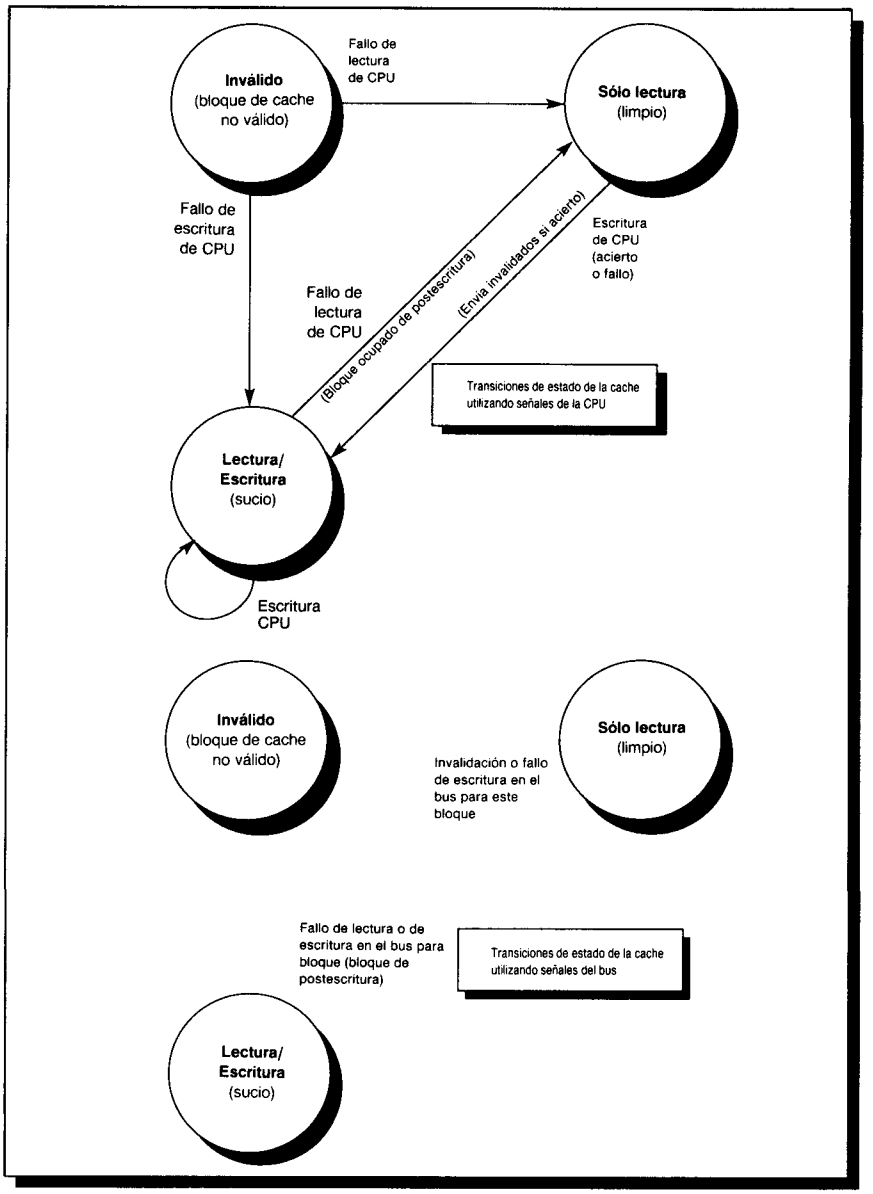


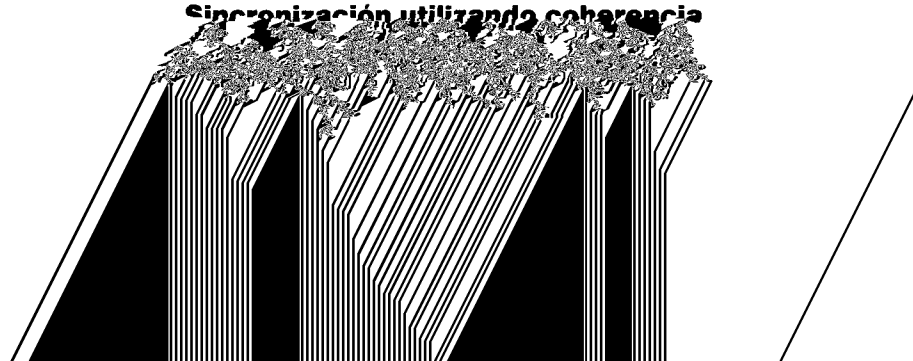
FIGURA 8.44 Un protocolo de coherencia cache de invalidación en escritura. La parte superior del diagrama muestra las transiciones, entre estados, basadas en acciones de la CPU asociadas con esta cache; la parte inferior muestra las transiciones basadas en operaciones sobre el bus. Sólo hay una máquina de estados en una cache, aunque se representen dos para clarificar cuándo ocurre una transición. Las flechas y estados negros estarán en una cache normal, las fechas grises se añaden para obtener la coherencia cache. En contraste a lo que aquí se muestra, algunos protocolos llaman a las escrituras a datos limpios «fallos de escritura», para que no haya señal separada para invalidación.

Nombre	Categoría	Política de escritura de memoria	Característica única
Write Once	Invalidación en escritura	Postescritura después de primera escritura	
Synapse N+1	Invalidación en escritura	Postescritura	Posesión de memoria explícita
Berkeley	Invalidación en escritura	Postescritura	Estado compartido propio
Illinois	Invalidación en escritura	Postescritura	Estado privado limpio; puede suministrar datos de cualquier cache con una copia limpia
Firefly	Difusión en escritura	Postescritura para privada, Escritura directa para compartida	Memoria actualizada en difusión
Dragon	Difusión en escritura	Postescritura para privada, Escritura directa para compartida	Memoria no actualizada en difusión

FIGURA 8.45 Seis protocolos de espionaje resumidos. Archibald y Baer [1986] utilizan estos nombres para describir los seis protocolos, y Eggers [1989] resume las analogías y diferencias como se muestran en la figura. La Figura 8.44 es más simple que cualquiera de estos protocolos.

lectura, si este bloque está en su cache. Si se tiene una copia y está en el estado de Lectura/Escritura, entonces, el bloque se escribe en memoria y se cambia al estado de inválido. (Una optimización no mostrada en la figura, realizaría el cambio de estado de ese bloque a sólo lectura.) Cuando una CPU escribe en un bloque, ese bloque va al estado de Lectura/Escritura. Si la escritura fuese un acierto, una señal de invalidación recorrería el bus. Como las caches vigilan el bus, todas comprueban si tienen una copia de ese bloque; si la tienen, lo invalidan. Si la escritura fuese un fallo, todas las caches con copias irían al estado inválido.

Como se puede imaginar, hay muchas variaciones en la coherencia cache que son mucho más complicadas que este sencillo modelo. Las variaciones incluyen que otras caches intenten o no suministrar el bloque si tienen una copia, si el bloque debe o no ser invalidado en un fallo de lectura, así como invalidar la escritura frente a difundirla como se explicó antes. La Figura 8.45 resume algunos protocolos de espionaje de coherencia cache.



mecanismo para decidir qué procesador supera el bloqueo y proporcionar la operación que bloquea sobre una variable. La arbitración es fácil para los multiprocesadores de bus compartido, ya que el bus es el único camino a memoria: el procesador que consigne el bus bloquea a los demás procesadores el acceso a memoria. Si la CPU y el bus proporcionan una operación atómica de intercambio, los programadores pueden crear bloqueos con la semántica apropiada. El adjetivo *atómica* es clave; significa que un procesador puede leer una posición e inicializarla con el valor bloqueado en la misma operación del bus, evitando que cualquier otro procesador lea o escriba en memoria.

La Figura 8.46 muestra un procedimiento típico para bloquear una varia-

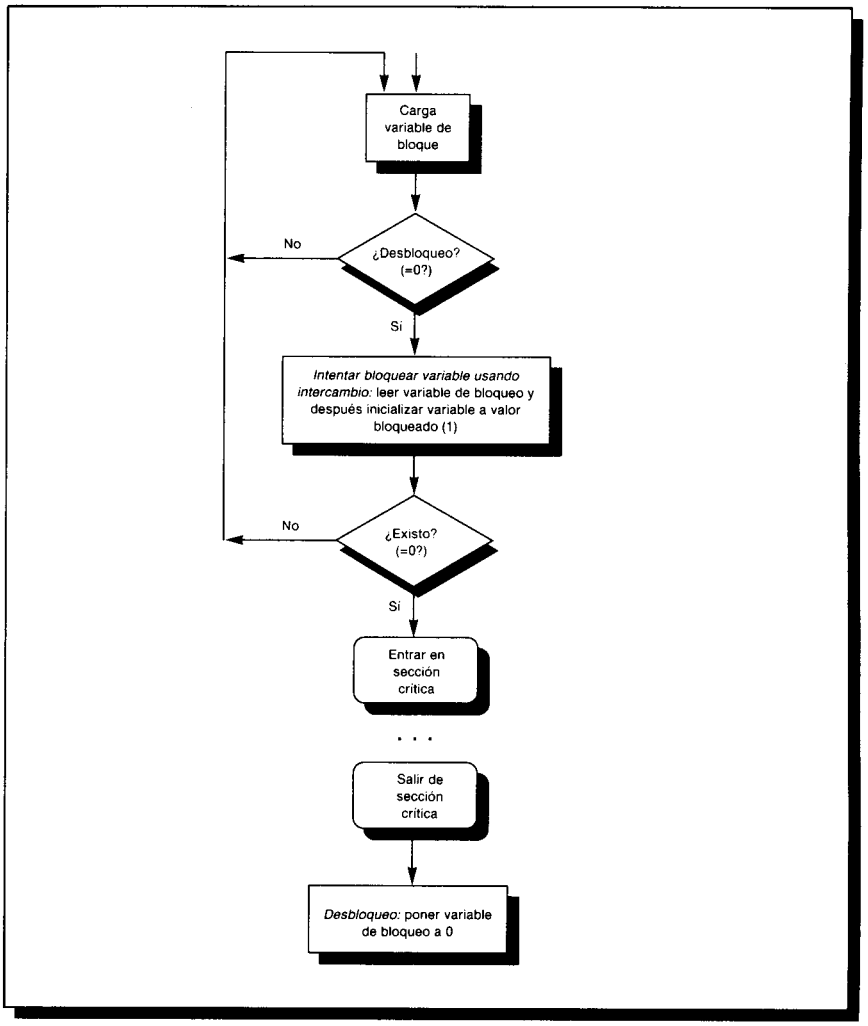


FIGURA 8.46 Pasos para superar un bloqueo para sincronizar procesos y después dejar el acceso desbloqueado a la salida de la sección clave del código.

para múltiples procesos intentando bloquear una variable.) Una vez que el procesador que superó el bloqueo almacena un 0 en la variable de bloqueo, todas las demás caches ven ese almacenamiento e invalidan su copia de la variable de bloqueo. Entonces obtienen el nuevo valor, que es 0. (Con coherencia cache de difusión en escritura como en la pág. 505, las caches actualizarán su copia en lugar de invalidarla primero y cargarla después de memoria.) Este nuevo valor da la salida a la carrera para ver quién puede inicializar primero la variable de bloqueo. El ganador consigue el bus y almacena un 1 en la variable de bloqueo; las otras caches sustituyen su copia de la variable de bloqueo, que contiene 0, por un 1. Leen que la variable está ya bloqueada y deben volver a examinar e iterar. Este esquema tiene dificultades para escalar a muchos procesadores a causa del tráfico de comunicaciones generado cuando desaparece el bloqueo.

Modelos de consistencia de memoria

Cuando introducimos coherencia cache para mantener la consistencia de múltiples copias de un objeto, surge una nueva pregunta: ¿Qué consistencia deben tener los valores vistos por dos procesadores? El problema se comprende mejor con un ejemplo: a continuación se muestran los segmentos de código de los dos procesos P1 y P2:

```
P1:   A = 0;              P2:   B = 0;
      .....                     .....
      A = 1;                    B = 1;
L1:   if (B == 0) ...     L2:   if (A == 0) ...
```

Suponer que los procesos están siendo ejecutados en procesadores diferentes, y que las posiciones `A` y `B` están originalmente «cacheadas» por ambos procesadores con el valor inicial de 0. Si la memoria siempre es consistente, será imposible para **ambas** sentencias «if» (etiquetadas `L1` y `L2`) evaluar sus condiciones como verdaderas (bien `A` = 1 o `B` = 1). Pero suponer que las invalidaciones de escritura tengan un retardo, y que al procesador se le permita continuar durante este retardo, entonces es posible que P1 y P2 no hayan visto las invalidaciones para `B` y `A` (respectivamente) **antes** que intenten leer los valores. La pregunta que surge con este ejemplo es: ¿cuán consistente debe ser la visión que tengan de memoria diferentes procesadores?

Una propuesta, llamada consistencia secuencial, requiere que el resultado de cualquier ejecución sea el mismo que si los accesos de cada procesador se mantuviesen en orden y los accesos entre diferentes procesadores se intercalasen arbitrariamente. En este caso, la aparente anomalía del ejemplo anterior no se puede presentar. Implementar consistencia secuencial requiere, habitualmente, que un procesador retarde cualquier acceso a memoria hasta que se completen todas las invalidaciones provocadas por las escrituras anteriores. Aunque este modelo presenta un sencillo paradigma de programación, reduce el rendimiento potencial, especialmente en una máquina con gran número de procesadores o grandes retardos de interconexión.

Modelos alternativos proporcionan un modelo de consistencia de memoria más débil. Por ejemplo, se puede requerir al programador para que utilice instrucciones de sincronización para ordenar accesos a memoria a la misma variable. Ahora, en lugar de retardar todos los accesos hasta que se completen las invalidaciones, sólo necesitan ser retrasados los accesos de sincronización.

Si los programadores desean consistencia secuencial o alguna forma más débil de consistencia, era todavía una cuestión abierta en 1990. El ejemplo anterior funcionará «correctamente» con consistencia secuencial, pero no con un modelo más débil. Para que una consistencia débil produzca los mismos resultados que la consistencia secuencial, el programa tendría que modificarse para que incluyese operaciones de sincronización que ordenasen los accesos a las variables A y B. Es natural que haya sincronización si se quiere que los procesos vean el último dato independiente de las velocidades de ejecución. Algunas máquinas eligen implementar la consistencia secuencial como modelo de programación, mientras que otras optan por una consistencia más débil. En el futuro, tal como se hacen tentativas para construir mayores multiprocesadores, el problema de la consistencia de memoria será cada vez más crítico para el rendimiento.

8.9 Juntando todo: la jerarquía de memoria del VAX-11/780

El desafío para el diseñador de jerarquías de memoria es elegir parámetros que conjuntamente funcionen bien, no inventar nuevas técnicas ni simular una cache en una configuración bien comprendida. En esta sección, para ilustrar las interacciones, se presenta con detalle un ejemplo completo utilizando la jerarquía de memoria del VAX-11/780. Aunque el VAX-11/780 no es una máquina muy reciente, se dispone de documentación de diseño y medidas en todos los aspectos de su jerarquía de memoria. La Figura 8.48 da una visión global.

Comencemos con una búsqueda de una instrucción inmediatamente después de un salto, cuando está vacío el buffer de búsqueda de instrucciones. La dirección virtual del PC se envía primero al TLB. El bit más significativo y los cinco bits inferiores de la dirección de la estructura de página indexan una entrada de cada banco del TLB. La inclusión del bit más significativo, que se utiliza para distinguir el espacio del sistema del espacio del proceso, garantiza que la mitad de cada banco contiene traducciones del sistema y la otra mitad de procesos. Las etiquetas se comparan para ver si la entrada coincide con la dirección de página requerida por el TLB. Si el bit de validez de la entrada no está a 1, entonces no hay coincidencia, independientemente de lo que indique la comparación de las etiquetas, y se indica un fallo.

Si hay coincidencia, la dirección física se forma concatenando la dirección física de la estructura de página de la entrada de la tabla de páginas del TLB con la parte de desplazamiento de página de la dirección. Para ahorrar tiempo, la parte del TLB que contiene la PTE (entrada de la tabla de páginas) se lee al

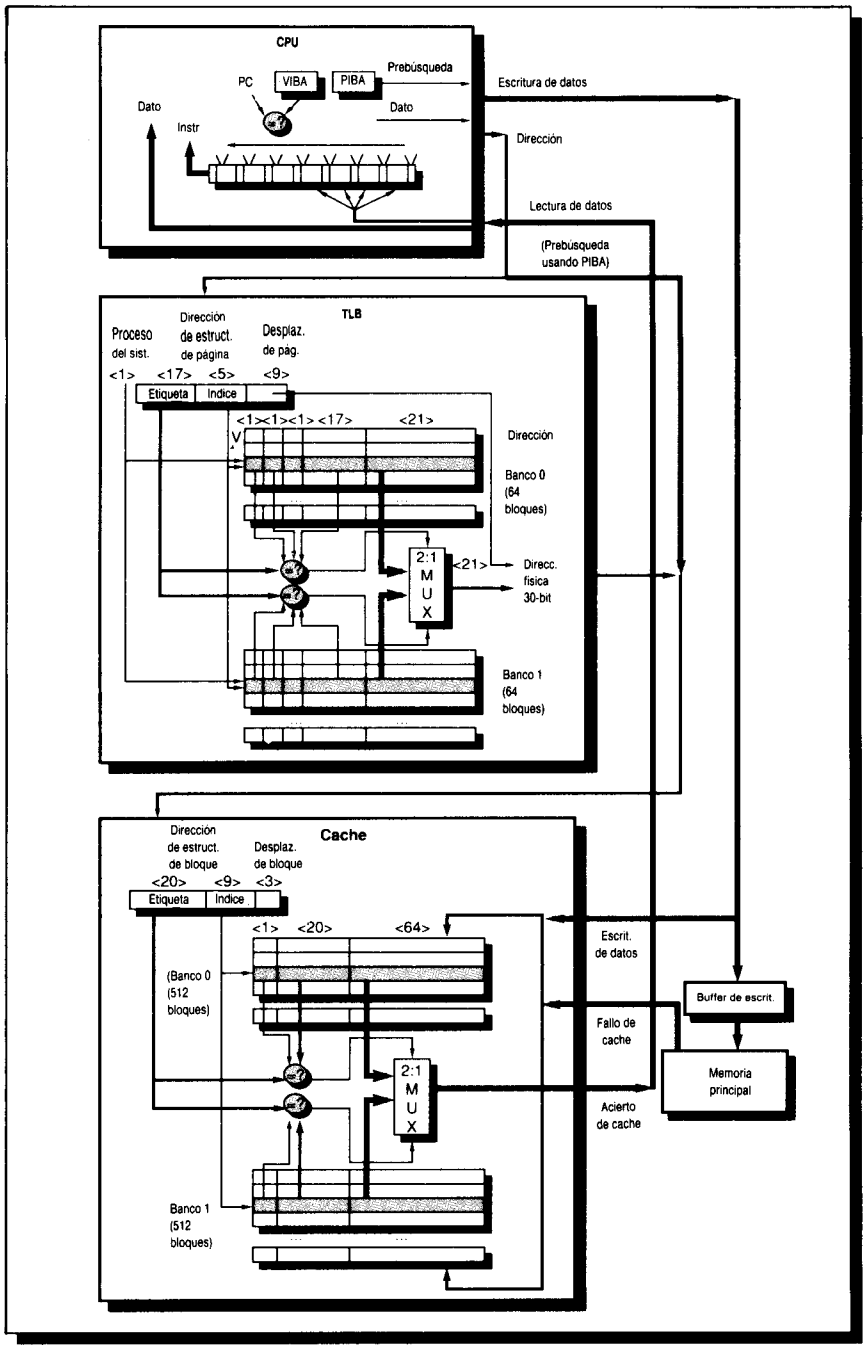


FIGURA 8.48 Visión global de la jerarquía de memoria del VAX-11/780. Los componentes individuales pueden verse con mayor detalle en las Figuras 8.11, 8.29 y 8.31.

mismo tiempo que las etiquetas, y un multiplexor 2:1, controlado por la lógica de comparación de las etiquetas, toma la PTE adecuada. Mientras se está formando la dirección, se comprueban los bits de protección de la PTE. Como esto es una búsqueda de instrucción, no hay problema mientras la página pueda ser leída por un proceso a este nivel. Si no hay violaciones de protección, esta dirección física se envía a la cache.

Al mismo tiempo que la dirección física se envía a la cache, dos registros del buffer de búsqueda de instrucciones de la CPU obtienen nuevos valores. El registro de *dirección del buffer de instrucciones virtuales* (VIBA) pasa a contener la estructura de página virtual del PC, y el registro de *dirección del buffer de instrucciones físicas* (PIBA) la dirección física correspondiente. Este truco, que se utilizó originalmente en la primera máquina con memoria virtual, evita que el buffer de prebúsqueda de instrucciones acceda mientras las instrucciones sean de la misma página. El PIBA contiene realmente la dirección del PC más 4, para que pueda comenzar la prebúsqueda de la siguiente instrucción. Este continúa intentando la prebúsqueda por delante del PC hasta una bifurcación (una ocurrencia frecuente en el VAX) o hasta que el PIBA intente cruzar un límite de página; en cualquier caso el VIBA y PIBA no se utilizan más para traducción de direcciones de instrucciones.

Mientras tanto, la cache ha recibido la dirección física de la instrucción. Con una cache asociativa por conjuntos de dos vías, bloques de 8 bytes y 512 bloques por conjunto, se necesitan nueve bits de dirección para indexar simultáneamente ambos bancos. Las direcciones parciales en las etiquetas se comparan con los bits correspondientes de la dirección física del PC para ver si hay una coincidencia. Por supuesto, hay bits de validez en cada etiqueta que deben estar a 1 o no habrá coincidencia.

Si hay una coincidencia, los bits inferiores de la dirección física del PC seleccionan la palabra del bloque de cache que se va a enviar a la unidad de prebúsqueda de instrucciones. Una vez de nuevo, leer datos y etiquetas a la vez, elimina cualquier retardo de tiempo adicional.

Cuando la palabra llega a la unidad de prebúsqueda, se coloca en los cuatro bytes de orden superior del buffer, que se marcan como válidos. El PIBA inmediatamente comienza a acceder a la cache con la dirección del PC más 4 para prebuscar la palabra siguiente. Como mencionamos antes, mientras la dirección de la estructura de página en el PC coincida con el VIBA, el PIBA evita el TLB y va directamente a la cache.

Supongamos que esta instrucción escribe un registro en memoria. El primer paso será enviar la dirección efectiva de memoria al TLB para su traducción. Como esto es una escritura, el bit de modificación de la PTE coincidente también debe estar a 1; si el bit de modificación no está ya a 1 da como resultado un trap a nivel de microcódigo de la instrucción que almacena el registro, necesitando otro ciclo de reloj para escribir el nuevo valor en el TLB. La dirección física se envía entonces a la cache. Entonces podemos ir a través del mismo proceso que antes (excluyendo la lectura), excepto que esta vez necesita un ciclo de reloj extra para modificar la parte del bloque seleccionada por la escritura y postescribirlo en la cache.

En una cache de escritura directa, el dato se debe escribir en memoria principal. Para evitar, en cada escritura, el retardo de siete ciclos de memoria principal, el VAX-11/780 utiliza un buffer de escritura de una palabra. Si el buffer

está vacío, se escribe la palabra y la CPU recibe la señal de continuación. Si está completo, la CPU se detiene hasta que esté vacío el buffer.

¿Cómo de bien funciona el 780? El punto clave de esta evaluación es el porcentaje de tiempo perdido mientras la CPU está esperando a la jerarquía de memoria. En una carga de trabajo de tiempo compartido, el número medio de ciclos de reloj por instrucción del 780 es de 10,6 ciclos de reloj. La descomposición por categorías es

Cálculo: 7,3 ciclos de reloj

Lectura: 0,8 ciclos de reloj

Detención de lectura: 1,0 ciclos de reloj

Escritura: 0,4 ciclos de reloj

Detención de escritura: 0,4 ciclos de reloj

Detención del buffer de preextracción de instrucciones: 0,7 ciclos de reloj

Aproximadamente el 20 por 100 del tiempo el VAX-11/780 está detenido esperando a memoria. Cuando el CPI base es 8,5 (calcular + leer + escribir), 2,1 ciclos de reloj para la jerarquía de memoria (detención de lectura + detención de escritura + detención de prebúsqueda) pueden ser satisfactorios, pero asolarían el rendimiento de una máquina con un CPI de 1 a 2.

Analicemos cada unidad de la jerarquía de memoria del 780. Una detención del buffer de prebúsqueda de instrucciones significa que el buffer está vacío, esperando que la cache suministre instrucciones a causa de un fallo de cache, un salto, demasiados accesos a datos (éstos tienen prioridad), bytes insuficientes para decodificar la instrucción, o alguna combinación de lo anterior. Las cargas del PIBA debidas a saltos frente a cruces de límite página varían con el benchmark, aunque del 64 al 91 por 100 de las veces (media = 76 por 100) la causa son los saltos. La unidad de prebúsqueda referencia a la cache 2,2 veces, en promedio, por instrucción VAX. El tamaño medio de instrucción es de 3,8 bytes, haciendo el tamaño efectivo de la prebúsqueda media de 1,7 bytes.

Ejemplo

La Figura 8.33 del Capítulo 3 muestra que el VAX ejecuta muchos menos bytes de instrucciones que DLX. Este ignora el buffer de prebúsqueda de instrucciones. ¿Cuánto deberíamos incrementar el número de bytes de instrucciones buscadas en la cache para incluir el efecto de prebúsqueda?

Respuesta

Podemos responder a esto de diversas formas. Cada acceso de prebúsqueda a la cache, realmente, devuelve 4 bytes, y el tamaño medio de instrucción del VAX es de 3,8 bytes; el incremento sería, por tanto

$$\frac{2{,}2 \cdot 4}{3{,}8} = 2{,}32$$

ya que la unidad de prebúsqueda referencia a la cache 2,2 veces por instrucción. Esto sugiere que el número de bytes buscados de la cache se debería in-

crementar en un 132 por 100. Sin embargo, como el mismo código puede buscar múltiples veces la unidad de prebúsqueda, el ancho de banda entre la cache y memoria no cambia ya que la unidad de prebúsqueda no puede provocar fallos de cache.

La pregunta también puede responderse en términos del número de bytes descartados a causa de un salto efectivo. Aproximadamente el 25 por 100 de las instrucciones cambian el PC en el VAX, y habrá de cero a ocho bytes en la unidad de prebúsqueda cuando se realiza un salto. Suponiendo con optimismo dos bytes, obtenemos un 13 por 100 de incremento:

$$\frac{3{,}8 + (25\,\% \cdot 2)}{3{,}8} = 1{,}13$$

Suponiendo seis bytes, obtenemos un 39 por 100 de incremento:

$$\frac{3{,}8 + (25\,\% \cdot 6)}{3{,}8} = 1{,}39$$

Mientras el tamaño variable de las instrucciones VAX mejora los bytes buscados en comparación con DLX, una evaluación más justa del VAX incrementaría los bytes buscados de la cache al menos del 13 al 39 por 100.

Con el buffer de prebúsqueda de instrucciones realizando muchas traducciones vía el PIBA y el VIBA, ¿cómo deben medirse los fallos del TLB? Las frecuencias de fallos del flujo de datos e instrucciones del TLB proporcionan una definición:

$$\text{Frecuencia de fallos del TLB debido al flujo de instrucciones} = \frac{\text{Fallos causados por IB}}{\text{Recuentos de PIBA}}$$

$$\text{Frecuencia de fallos del TLB debido al flujo de datos} = \frac{\text{Fallos}}{\text{Peticiones de palabras de datos de 32 bits}}$$

La definición para flujo de datos significa que referencias a objetos de datos mayores de cuatro bytes cuentan como accesos múltiples, como es el caso de los accesos a datos no alineados. La Figura 8.49 muestra las frecuencias de fallos del TLB.

Las referencias globales al TLB después de filtrarlas el PIBA, se dividen en: 20 por 100 flujo de instrucciones del usuario, 62 por 100 flujo de datos del usuario, 3 por 100 flujo de instrucciones del sistema y 15 por 100 flujo de datos del sistema. Para tener en cuenta el filtrado de direcciones por la optimización del PIBA, los fallos del TLB también se pueden contabilizar como una frecuencia por instrucción ejecutada, como en la Figura 8.50.

Frecuecias de fallos del TLB	Flujo de instrucciones	Flujo de datos	Total
Proceso	0,7 %	0,6 %	0,7 %
Sistema	15,4 %	5,4 %	7,2 %
Total	3,5 %	1,6 %	1,9 %

FIGURA 8.49 Frecuencias de fallos para el TLB del VAX-11/780, ignorando el impacto de las instrucciones no traducidas por el TLB. Este dato se midió en una carga de trabajo diferente de tiempo compartido de la utilizada en las medidas VAX anteriores [Clark y Emer 1985].

Fallos del TLB por 100 instrucciones	Flujo de instrucciones	Flujo de datos	Total
Proceso	0,18	0,50	0,68
Sistema	0,62	1,03	1,65
Total	0,80	1,53	2,33

FIGURA 8.50 Fallos por cien instrucciones para el TLB del VAX-11/780. De forma distinta a la Figura 8.49, esta evaluación global del TLB contabiliza el efecto del PIBA.

El TLB del VAX emplea una media de 21,6 ciclos de reloj en un fallo (incluyendo 3,5 ciclos de reloj por fallos de cache para algunas entradas de la tabla de página), añadiendo un total de 0,7 ciclos de reloj por instrucción para fallos del TLB para la instrucción media. Por ello, aproximadamente un tercio de las detenciones del sistema de memoria son debidas a fallos del TLB.

El mismo estudio de Emer y Clark [1984] mostró una variación significativa en las frecuencias de fallos de cache:

- Las frecuencias de fallos de cache, del flujo de datos, variaban a lo largo del día del 12 al 25 por 100, con una media del 17 por 100.
- Las frecuencias de fallos de cache, del flujo del buffer de instrucciones, variaban del 4 al 13 por 100, con una media del 8 por 100.
- La distribución de accesos a la cache desde la CPU eran: 68 por 100 lecturas del flujo del buffer de prebúsqueda de instrucciones, 20 por 100 lecturas del flujo de datos y 12 por 100 escrituras del flujo de datos. Calculadas por instrucción, hay aproximadamente 2,2 referencias del buffer de prebúsqueda de instrucciones, 0,8 lecturas de datos por instrucción y 0,4 escrituras de datos por instrucción.

Ejemplo

De acuerdo con el «VAX-11/780 Architecture Handbook», para la carga de trabajo medida en 1978, la frecuencia de fallos del TLB era aproximadamente del 3 por 100. ¿Qué resultado obtendríamos para la carga de trabajo de tiempo compartido, medida en 1984?

Respuesta

Suponiendo una referencia a memoria para obtener la instrucción media VAX de 3,8 bytes, la frecuencia de fallos es 1 por 100:

$$\frac{\dfrac{2{,}3 \text{ fallos TLB}}{100 \text{ instrucciones}}}{\dfrac{1 + 0{,}8 + 0{,}4 \text{ referencias}}{\text{instrucción}}} = \frac{2{,}3}{100 \cdot 2{,}2} = 0{,}01$$

Incluyendo el VIBA-PIBA, la Figura 8.49 muestra una frecuencia de fallos del 1,9 por 100.

Ejemplo

De acuerdo con el «VAX-11/780 Architecture Handbook», para la carga de trabajo medida en 1978 la frecuencia de fallos de cache era aproximadamente del 5 por 100. ¿Qué resultado obtendríamos para la carga de trabajo en tiempo compartido, medida en 1984?

Respuesta

La frecuencia de fallos de la cache varía. La frecuencia media de fallos es

$$68\,\% \cdot 8\,\% + 20\,\% \cdot 17\,\% + 12\,\% \cdot 17\,\% = 11\,\%$$

En el mejor caso, la respuesta es

$$68\,\% \cdot 4\,\% + 20\,\% \cdot 12\,\% + 12\,\% \cdot 12\,\% = 7\,\%$$

En el peor caso,

$$68\,\% \cdot 13\,\% + 20\,\% \cdot 25\,\% + 12\,\% \cdot 25\,\% = 17\,\%$$

8.10 Falacias y pifias

Como la más cuantitativa de las disciplinas de la arquitectura de computadores, la jerarquía de memoria debería ser menos vulnerable a las falacias y pifias. Aun así, los autores se vieron limitados en esta sección, no por falta de advertencias, sino por espacio.

Pifia: Un espacio de direcciones demasiado pequeño

Cinco años después que DEC y la Universidad de Carnegie-Mellon colaborasen para diseñar la nueva familia de computadores PDP-11, fue evidente que su creación tuvo un defecto fatal. Una arquitectura anunciada por IBM seis

años **antes** del PDP-11 está todavía prosperando, con modificaciones menores, veinticinco años más tarde. Y el VAX de DEC, criticado por incluir funciones innecesarias, ha vendido 100 000 unidades desde que la PDP-11 dejó de fabricarse. ¿Por qué?

El defecto fatal del PDP-11 fue el tamaño de sus direcciones, comparadas con las del IBM 360 y el VAX. El tamaño de direcciones limita la longitud del programa, ya que el tamaño de un programa y la cantidad de datos que necesita el programa deben ser menores que $2^{\text{tamaño dirección}}$. La razón por la que el tamaño de la dirección es tan difícil de cambiar, es que determina la anchura mínima de cualquier cosa que pueda contener una dirección: PC, registro, palabra de memoria y aritmética de direcciones efectivas. Si desde el principio no hay un plan para expandir las direcciones, entonces las posibilidades de cambiar con éxito su tamaño son tan escasas que normalmente significa el fin de esa familia de computadores. Bell y Strecker [1976] escribieron:

Sólo hay un error, que se puede cometer en el diseño de computadores, que es difícil de corregir: no tener suficientes bits de dirección para el direccionamiento y gestión de la memoria. PDP-11 siguió la tradición ininterrumpida de casi todos los computadores conocidos. [p. 2]

Una lista parcial de máquinas con éxito que eventualmente desaparecieron por falta de bits de dirección incluye al PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, AMI 6502, Zilog Z80, CRAY-1 y CRAY X-MP.

Falacia: Dados los recursos hardware, el diseñador de computadores que selecciona una cache asociativa por conjuntos sobre una cache de correspondencia directa del mismo tamaño obtendrá un computador más rápido.

La pregunta aquí es si la lógica extra de la cache asociativa por conjuntos afecta al tiempo de aciertos y, por tanto, posiblemente, a la frecuencia de reloj de la CPU. (Ver Fig. 8.11.) Si afecta al tiempo de aciertos, entonces la pregunta es si la ventaja en frecuencia de fallos más baja contrarresta el tiempo de aciertos más lento. A mediados de los años ochenta, muchos reconocieron este peligro y seleccionaron la ubicación de correspondencia directa; por ejemplo, MIPS M/500, Sun 3/260 y VAX 8800. Hill [1988] hace un elocuente razonamiento a favor de las caches de correspondencia directa, incluyendo costes más bajos, tiempos de acierto más rápidos y, por tanto, tiempos de accesos medios menores para caches grandes de correspondencia directa. Las caches de correspondencia directa también permiten que el dato leído se envíe a la CPU y se utilice incluso antes que se determine el acierto/fallo, particularmente útil con una CPU segmentada. Hill encontró aproximadamente un 10 por 100 de diferencia en los tiempos de aciertos para las caches TTL o ECL y un 2 por 100 de diferencia para las caches CMOS a medida, con un cambio absoluto en las frecuencias de fallos de menos del 1 por 100 para grandes caches. Ya que un acierto de cache de correspondencia directa puede ser accedido con más ra-

pidez y el tiempo de acierto normalmente determina la duración del ciclo de reloj del procesador, una CPU con una cache de correspondencia directa puede ser tan rápida o más que una CPU con una cache asociativa por conjuntos de dos vías del mismo tamaño. Przybylski, Horowitz y Hennessy [1988] muestran algunos ejemplos de estas cuestiones.

Falacia: Un sistema de memoria puede ser diseñado utilizando trazas de diferentes arquitecturas.

La Figura 8.51 muestra las frecuencias de fallos de las caches de instrucciones y de datos para el mismo programa en dos arquitecturas diferentes. Estos datos son de la primera parte de la ejecución de Spice en DLX y en el VAX. El cambio de accesos a datos en el VAX por accesos a instrucciones en DLX vistos en la Figura 3.33 del Capítulo 3 se refleja aquí: el 61 por 100 de las referencias VAX y el 52 por 100 de lo fallos son para datos. Observar que mientras DLX tiene sólo las tres cuartas partes del número absoluto de fallos de datos, su **frecuencia** de fallos de datos es tres veces mayor.

Pifia: Basar el tamaño del buffer de escritura en la velocidad de memoria y la mezcla media de escrituras.

	VAX	DLX
Referencias de instrucciones	576 169	918 537
Fallos de instrucciones	2 033	3 188
Frecuencia de fallos de instrucciones	0,4 %	0,3 %
Referencias de datos	923 831	264 453
Fallos de datos	2 200	1 595
Frecuencia de fallos de datos	0,2 %	0,6 %
Referencias totales	1 500 000	1 182 990
Porcentaje de instrucciones de referencias totales	38 %	78 %
Fallos totales	4 233	4 782
Porcentaje de fallos de instrucciones de fallos totales	48 %	67 %
Promedio de frecuencia de fallos	0,3 %	0,4 %

FIGURA 8.51 Frecuencia de fallos para VAX y DLX en una fase inicial de Spice. La simulación supone caches de datos e instrucciones separadas. Cada cache es de correspondencia directa, utiliza bloques de 16 bytes, y contiene 64 KB. Ambas usan escrituras con ubicación en escritura. (Observar que de forma distinta al Capítulo 2, estos datos se obtuvieron utilizando el compilador F77 para una parte del programa Spice.)

Esto parece una aproximación razonable:

$$\text{Tamaño buffer de escritura} = \frac{\text{referencias a memoria}}{\text{ciclo de reloj}} \cdot$$

$$\cdot \text{ Porcentaje de escrituras} \cdot \text{Ciclos de reloj para escribir memoria}$$

Si hay una referencia a memoria por ciclo de reloj, el 10 por 100 de las referencias a memoria son escrituras, y escribir una palabra de memoria necesita 10 ciclos, entonces se añade un buffer de una palabra ($1 \cdot 10\,\% \cdot 10 = 1$). El cálculo para el VAX-11/780 utilizando los datos de la última sección,

$$\frac{\text{3,4 referencias a memoria}}{\text{10,6 ciclos de reloj}} \cdot \frac{\text{0,4 escrituras}}{\text{3,4 referencias a memoria}} \cdot$$

$$\cdot \frac{\text{6 ciclos de reloj}}{\text{Escritura}} = 0{,}22$$

Por ello, un buffer de una palabra parece suficiente.

La pifia es que cuando las escrituras se presentan muy juntas, la CPU se debe detener hasta que se complete la escritura anterior. El buffer de escritura de una palabra del VAX-11/780 es la principal razón de sus detenciones de escritura (aproximadamente el 20 por 100 de todas las detenciones). La pregunta que debemos hacernos es qué tamaño de buffer se necesita para que las detenciones de escritura de la CPU sean pocas. El impacto del tamaño de buffer de escritura se puede establecer por simulación o estimar con un modelo de cola.

Pifia: Extender un espacio de direcciones añadiendo segmentos a un espacio de direcciones plano (flat).

Durante los años setenta, muchos programas crecieron hasta el punto que no podían direccionar todo el código y datos con una dirección de 16 bits. Las máquinas fueron entonces revisadas para que ofrecieran direcciones de 32 bits, bien a través de un espacio de direcciones plano de 32 bits o bien añadiendo 16 bits de segmento a la dirección existente de 16 bits. Desde el punto de vista del «marketing», añadir segmentos resuelve el problema de direccionamiento. Desgraciadamente, hay dificultades siempre que un lenguaje de programación quiera una dirección mayor que un segmento, como por ejemplo índices para grandes arrays, punteros no restringidos o parámetros de referencia. Además, añadir segmentos puede cambiar cada dirección a dos palabras —una para el número de segmento y otra para el desplazamiento de segmento— provocando problemas en el uso de direcciones en los registros. En los años 90, se agotarán las direcciones de 32 bits, y será interesante ver si se repetirá la historia con las mismas consecuencias de ir a mayores direcciones planas en contraposición a la adición de segmentos.

Falacia: Las caches son tan rápidas como los registros.

Esta falacia es importante, porque si las caches fuesen tan rápidas como los registros, no habría necesidad de registros. Sin registros no habría necesidad de un ubicador de registros y, por tanto, los compiladores podrían ser más sencillos. La falacia es difícil de probar cuantitativamente, aunque se pueden citar multitud de ejemplos. Lampson [1982] resumió esta experiencia:

Un banco de registros es más rápido que una cache, porque es más pequeño, y porque el mecanismo de direcciones es mucho más simple. Los diseñadores de máquinas de alto rendimiento normalmente han determinado que es posible leer un registro y escribir otro en un solo ciclo, mientras que se necesitan dos ciclos [latencia] para un acceso a cache... Además, como no hay muchos registros, es factible duplicarlos o triplicarlos, para que varios registros se puedan leer simultáneamente. [p. 74]

Como se mencionó en el Capítulo 3, las direcciones cortas de los registros permiten una codificación más compacta de las instrucciones. A los autores les parece que el acceso determinista a bancos de registros multipuerto ofrecerá siempre menor latencia o mayor ancho de banda o ambas cosas, cuando se compare con los accesos no deterministas a las caches.

8.11 Observaciones finales

La dificultad de construir un sistema de memoria con las mismas prestaciones que las CPU más rápidas está acentuada por el hecho de que la materia prima de la memoria principal es la misma que se encuentra en los computadores más baratos. El principio de localidad es el que nos salva aquí —su solidez se demuestra a todos los niveles de la jerarquía de memoria en los computadores actuales, desde discos a buffers de instrucciones.

Los fallos de cada nivel se pueden categorizar en tres causas —forzosos, capacidad y conflicto— y para cada caso se emplean diferentes técnicas. La Figura 8.52 resume los atributos de los ejemplos de jerarquía de memoria descritos en este capítulo

Suele haber un codo en la curva de coste/rendimiento de la jerarquía de memoria: antes del codo se desperdicia rendimiento y después del codo se malgasta hardware. Los arquitectos encuentran ese punto por simulación y análisis cuantitativo.

8.12 Perspectiva histórica y referencias

Aunque los pioneros de la computación conocían la necesidad de una jerarquía de memoria y acuñaron el término, la gestión automática de dos niveles la propuso primero Kilburn y cols. [1962] y fue demostrada con el computador Atlas en la Universidad de Manchester. Esto ocurría el año **antes** que se anunciase el IBM 360. Aunque IBM planeaba introducirla con la siguiente generación (System/370), el sistema operativo no aceptó el desafío en 1970. La

	Ventanas de registro	Buffer de prebúsqueda de instrucciones	TLB	Cache de primer nivel	Cache de segundo nivel	Memoria virtual
Tamaño de bloque	64 bytes	1 byte	4-8 (1 PTE)	4-128 bytes	32-256 bytes	512-8192 bytes
Tiempo de acierto	1 ciclo de reloj	1 ciclo de reloj	1 ciclo de reloj	1-4 ciclos de reloj	4-10 ciclos de reloj	1-10 ciclos de reloj
Penalización de fallos	32-64 ciclos de reloj	2-6 ciclos de reloj	10-30 ciclos de reloj	8-32 ciclos de reloj	30-80 ciclos de reloj	100.000-600.000 ciclos de reloj
Frecuencia de fallos (local)	1 %-3 %	10 %-25 %	0,1 %-2 %	1 %-20 %	15 %-30 %	0,00001 % 0,001 %
Tamaño	512 bytes	6-12 bytes	32-8192 (8-1024 PTEs)	1KB-256KB	256 KB-4 MB	4 MB-2048 MB
Post-almacenamiento	Cache de primer nivel	Cache de primer nivel	Cache de primer nivel	Cache de segundo nivel	DRAM de columnas estáticas	Discos
P1: ubicación de bloque	Buffer circular	N.A. (cola)	Asociativa por conjuntos	Correspondencia directa	Asociativa por conjuntos	Completamente asociativa
P2: identificación de bloques	2 registros: superior e inferior	Bits válidos + 1 registro	Etiqueta/bloque	Etiqueta/bloque	Etiqueta/bloque	Tabla
P3: reemplazo de bloque	Primero en entrar, primero en salir	N.A. (cola)	Aleatorio	N.A. (correspondencia directa)	Aleatorio	LRU
P4: estrategia de escritura	Escritura	Limpiar en escritura al buffer de instrucción (si es posible)	Limpiar en escritura a la tabla de páginas	Escritura directa o postescritura	Escritura directa o postescritura	Postescritura

FIGURA 8.52 Resumen de los ejemplos de jerarquía de memoria de este capítulo.

memoria virtual se anunció para la familia 370 en 1972, y para esta máquina se acuñó el término de «buffer de traducción anticipada» (translation-lookaside buffer) (ver Case y Padegs [1978]). Los únicos computadores sin memoria virtual, hoy día, son algunos supercomputadores y computadores personales.

Tanto el Atlas como el IBM 360 tenían protección de páginas, y con el tiempo las máquinas evolucionaron a mecanismos más elaborados. El mecanismo más elaborado fue el de las capacidades *(capabilities),* que alcanzó su máximo interés a finales de los años setenta y principios de los ochenta [Fabry,

1974, y Wulf, Levin y Harbison, 1981]. Wilkes [1982], uno de los primeros trabajadores en capacidades, había dicho esto sobre las capacidades:

Cualquiera que haya estado relacionado con una implementación del tipo descrito [sistema de capacidad], o haya tratado de explicarla a los demás, probablemente sienta que la complejidad se le va de las manos. Es particularmente decepcionante que la atractiva idea de que las capacidades sean tickets que se puedan pasar libremente de mano en mano ha llegado a perderse...

Comparado con un sistema computador convencional, habrá inevitablemente un coste para hacer frente a la provisión de un sistema en el cual los dominios de protección sean pequeños y frecuentemente cambiados. Este coste se manifestará por sí mismo en términos de hardware adicional, disminución de la velocidad de tiempo de ejecución y aumento de la ocupación de memoria. Actualmente es una pregunta abierta si, por adopción de la aproximación de capacidades, el coste puede reducirse a proporciones razonables.

Hoy en día hay poco interés en las capacidades, tanto por parte de las comunidades de sistemas operativos como de arquitectura de computadores, aunque hay un creciente interés en protección y seguridad.

Bell y Strecker [1976] reflexionaron sobre el PDP-11 e identificaron un pequeño espacio de direcciones como el único error arquitectónico que es difícil de corregir. En la época de la creación del PDP-11, las memorias de ferrita crecían a una velocidad muy lenta, y la competencia de 100 compañías de minicomputadores significaba que DEC podía no disponer de un producto competitivo en coste si cada dirección tenía que ir dos veces a través de un camino de datos de 16 bits. De aquí, la decisión de añadir sólo 4 bits más de dirección que el predecesor del PDP-11. Los arquitectos del IBM 360 eran conscientes de la importancia del tamaño de la dirección y planificaron la arquitectura para extenderla a 32 bits. Sin embargo, sólo se utilizaron 24 bits en el IBM 360, porque los modelos inferiores de la 360 habrían sido aún más lentos con mayores direcciones. Desgraciadamente, los arquitectos no revelaron sus planes a la gente del software, y el esfuerzo de expansión lo frustraron los programadores que almacenaban información extra en los bits superiores de dirección «no utilizados».

Algunos años después del artículo de Atlas, Wilkes publicó el primer artículo describiendo el concepto de una cache [1965]:

Se explica el uso de una memoria de ferrita más rápida, por ejemplo, de 32 000 palabras como esclava de una memoria de ferrita más lenta de, por ejemplo, un millón de palabras de forma tal que, en la práctica, el tiempo efectivo de acceso esté más próximo al de la memoria rápida que al de la memoria lenta. [p. 270]

Este artículo de dos páginas describe una cache de correspondencia directa. Aunque esta es la primera publicación sobre caches, la primera implementación probablemente fue una cache de instrucciones de correspondencia di-

recta construida en la Universidad de Cambridge. Estaba basada en una memoria de diodos túnel, la forma más rápida de memoria disponible en esa época. Wilkes indica que G. Scarott sugirió la idea de una memoria cache.

Después de esa publicación, IBM comenzó un proyecto que desembocó en la primera máquina comercial con cache, la IBM 360/85 [Liptay, 1968]. Gibson [1967] describe cómo medir el comportamiento de programas como tráfico de memoria así como frecuencia de fallos y muestra cómo la frecuencia de fallos varía entre programas. Utilizando una muestra de 20 programas (¡cada uno con 3 000 000 de referencias!), también Gibson se basó en el tiempo medio de acceso a memoria para comparar sistemas con y sin caches. Esto fue hace veinte años y, sin embargo, muchos utilizaron la frecuencia de fallos hasta hace poco.

Conti, Gibson y Pitkowsky [1968] describen el rendimiento resultante del 360/85. El 360/91 supera al 360/85 en sólo 3 de 11 programas del artículo, aun cuando el 360/85 tiene un ciclo de reloj más lento (80 ns frente a 60 ns), menor entrelazado de memoria (4 frente a 16) y una memoria principal más lenta (1,04 μs frente a 0,75 μs). Este es el primer artículo que utiliza el término «cache». Strecker [1976] publicó el primer artículo comparativo de diseño de caches examinando caches para el PDP-11. Smith [1982] publicó más tarde un artículo general utilizando los términos de «localidad espacial» y «localidad temporal»; este artículo ha servido como referencia a muchos diseñadores de computadores. Aunque muchos estudios se han basado en simulaciones, Clark [1983] utilizó un monitor hardware para registrar los fallos cache del VAX-11/780 durante varios días. La Sección 8.9 comenta estos hallazgos, junto con el trabajo de Clark y Emer sobre los TLB [1984, 1985]. Un estudio similar se realizó sobre el VAX 8800 [Clark y cols., 1988]. Agarwal, Sites y Horowitz [1986] cambiaron el microcódigo de un VAX para obtener trazas del código de usuario y del sistema. Estas trazas se han utilizado en este libro (y se puede disponer de ellas por medio del editor). Hill [1987] propuso las tres «C» utilizadas en la Sección 8.4 para explicar los fallos de cache. Las caches siguen siendo un área activa de investigación, tal como Smith [1986] ha registrado en su amplia bibliografía.

Muchas de las ideas de la sección avanzada de caches sólo han sido tratadas recientemente. La inclusión de caches en microprocesadores como el Motorola 68020 dieron lugar a las máquinas de cache de dos niveles; la Sun 3/260 en 1986 fue quizá la primera. En 1988, la 4D/240 de Silicon Graphics tenía dos niveles de cache para datos e instrucciones; el segundo nivel se añadió, principalmente, para que la coherencia cache permitiese multiprocesamiento de cuatro vías. El MIPS RC 6280 es, probablemente, la primera máquina con dos niveles de cache por las razones dadas en la página 501 [Roberts, Taylor y Layman, 1990]. Goodman y Chiang [1984] fueron los primeros en publicar una investigación de las DRAM de columnas estáticas en una jerarquía de memoria, mientras que Kelly [1988] refinó la idea utilizando direcciones virtuales. Goodman [1987] mostró que los alias se podían manipular en el tiempo de fallos de la cache, y Wang, Baer y Levy [1989] mostraron que el control extra para esto no parecía muy malo para dos niveles de cache.

En comparación con las demás ideas de la sección avanzada, la investigación de la coherencia cache es mucho más antigua. Tang [1976] publicó el primer protocolo de coherencia cache utilizando directorios, y esta aproxi-

mación se implementó en el IBM 3081. Censier y Feautrier [1978] describen una técnica con etiquetas de estado en memoria. La primera máquina en utilizar caches de espionaje *(snooping)* fue la Synapse N + 1 [Frank, 1984]; la primera publicación sobre caches de espionaje fue de Goodman [1983]. Archibald y Baer [1986] dan una visión general de la amplia variedad de esquemas para coherencia cache. Referencias sobre los protocolos mencionados en su artículo y en la Figura 8.45 son Frank [1984] para Synapse; Goodman [1983] para Write Once; Katz y cols. [1985] para Berkeley; McCreight [1984] para Dragon; Papamarcos y Patel [1984] para Illinois, y Thacker y Stewart [1987] para Firefly. Baer y Wang [1988] explican la inclusión multinivel. La nomenclatura de Eggers [1989] para categorizar las caches de espionaje se adopta en este texto. El Capítulo 10, Sección 10.7 menciona el uso de la prebúsqueda para mejorar el rendimiento de las caches, y Kroft [1981] describe el diseño de una cache que permite servir peticiones posteriores mientras que el dato requerido se está prebuscando. Przybylski [1990] y las disertaciones de Agarwal [1987], Eggers [1989] y Hill [1987] investigan muchos aspectos de los tópicos avanzados de las caches con más profundidad.

Artículos sobre otro uso de localidad, ventanas de registros o caches de pila, son de Patterson y Sequin [1981], Ditzel y McClellan [1982] y Lampson [1982]. Sites escribió un artículo anterior [1979] sugiriendo una forma de utilizar los recursos VLSI para obtener un mayor rendimiento utilizando muchos registros, y estos esquemas son una interpretación de esa recomendación.

Referencias

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogram ming,* Ph.D. Thesis, Stanford Univ., Tehc. Rep. No. CSL-TR-87-332 (May).

AGARWAL, A., R. L. SITES, AND M. HOROWITZ [1986]. «ATUM: A new technique for capturing address traces using microcode», *Proc. 13th Annual Symposium on Computer Architecture* (June 2-5), Tokyo, Japan, 119-127.

ARCHIBALD, J. AND J.-L. BAER [1986]. «Cache coherence protocols: Evaluation using a multiprocessor simulation model», *ACM Trans. on Computer Systems* 4:4 (November) 273-298.

BAER, J.-L. AND W.-H. WANG [1988]. «On the inclusion property for multi-level cache hierarchies», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, 73-80.

BELL, C. G. AND W. D. STRECKER [1976]. «Computer structures: What have we learned from the PDP-11?», *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 1-14.

BLAKKEN, J. [1983]. «Register windows for SOAR», in *Smalltalk On a RISC: Architectural Investigations,* Proc. of CS 292R (April) 126-140, University of California.

CASE, R. P. AND A. PADEGS [1978]. «The architecture of the IBM System/370», *Communications of the ACM* 21:1, 73-96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830-855.

CENSIER, L. M. AND P. FEAUTRIER [1978]. «A new solution to the coherence problem in multicache systems», *IEEE Trans. on Computers* C-27:12 (December) 1112-1118.

CLARK, D. W. [1983]. «Cache performance of the VAX-11/780», *ACM Trans. on Computer Systems* 1:1, 24-37.

CLARK, D. W. AND J. S. EMER [1985]. «Performance of the VAX-11/780 translation buffer: Simulation and measurement», *ACM Trans. on Computer Systems* 3:1, 31-62.

CLARK, D. W., P. J. BANNON, AND J. B. KELLER [1988]. «Measuring VAX 8800 Performance with a Histogram hardware monitor», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, Hawaii, 176-185.

CONTI, C., D. H. GIBSON, AND S. H. PITOWSKY [1968]. «Structural aspects of the System/360 Model 85, par I: General organization», *IBM Systems J.* 7:1, 2-14.

CRAWFORD, J. H. AND P. P. GELSINGER [1987]. *Programming the 80386,* Sybex, Alameda, Calif.

DITZEL, D. R., AND H. R. MCCLELLAN [1982] «Register allocation for free: The C machine stack cache», *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 48-56.

EGGERS, S. [1989] *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors,* Ph. D. Thesis, Univ. of California, Berkeley, Computer Science Division Tech. Rep. UCB/CSD 89/501 (April).

EMER, J. S. AND D. W. CLARK [1984]. «A characterization of processor performance of the VAX-11/780», *Proc. 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301-310.

FABRY, R. S. [1974]. «Capability based addressing», *Comm. ACM* 17:7 (July) 403-412.

FRANK, S. J. [1984]. «Tightly coupled multiprocessor systems speed memory access times», *Electronics* 57:1 (January) 164-169.

GIBSON, D. H. [1967]. «Considerations in block-oriented systems design», *AFIPS Conf. Proc.* 30, SJCC, 75-80.

GOODMAN, J. R. [1983]. «Using cache memory to reduce processor memory traffic», *Proc. Tenth Annual Symposium on Computer Architecture* (June 5-7), Stockholm, Sweden, 124-131.

GOODMAN, J. R. AND M.-C. CHIANG [1984]. «The use of static column RAM as a memory hierarchy», *Proc. 11th Annual Symposium on Computer Architecture* (June 5-7), Ann Arbor, Mich., 167-174.

GOODMAN, J. R. [1987]. «Coherency for multiprocessor virtual address caches», *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, Calif., 71-81.

HALBERT, C. D. AND P. B. KESSLER [1980]. «Windows of overlapping register frames», *CS 292R Final Reports* (June) 82-100.

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance,* Ph. D. Thesis, Univ. of California at Berkeley Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November).

HILL, M. D. [1988]. «A case for direct mapped caches», *Computer* 21:12 (December) 25-40.

HUGUET, M. AND T. LANG [1985]. «A reduced register file for RISC architectures», *Computer Architecture News* 13:4 (September) 22-31.

KATZ, R. S. EGGERS, D. A. WOOD, C. PERKINS, AND R. G. SHELDON [1985]. «Implementing a cache consistency protocol», *Proc. 12th Annual Symposium on Computer Architecture,* 276-283.

KELLY, E. [1988]. «'SCRAM Cache' in Sun-4/110 beats traditional caches», *Sun Technology* 1:3 (Summer) 19-21.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, F. H. SUMNER [1962]. «One-level storage system», *IRE Transactions on Electronic Computers* EC-11 (April) 223-235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135-148.

KROFT, D. [1981]. «Lockup-free instruction fetch/prefetch cache organization», *Proc. Eighth Annual Symposium on Computer Architecture* (May 12-14), Minneapolis, Minn., 81-87.

LAMPSON, B. W. [1982]. «Fast procedure calls», *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 66-75.

LIPTAY, J. S. [1968]. «Structural aspects of the System/360 Model 85, part II: The cache», *IBM Systems J.* 7:1, 15-21.

MCCALL, K. [1983]. «The Smalltalk-80 benchmarks», *Smalltalk 80: Bits of History, Words of Advice,* G. Krasner, ed., Addison-Wesley, Reading, Mass., 153-174.

MCCREIGHT, E. [1984]. «The Dragon computer system: An early overview», Tech. Rep. Xerox Corp. (September).

MCFARLING, S. [1989]. «Program optimization for instruction caches», *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* (April 3-6), Boston, Mass., 183-191.

PAPAMARCOS, M. AND J. PATEL [1984]. «A low coherence solution for multiprocessors with private cache memories», *Proc. of the 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 348-354.

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach,* Morgan Kaufmann Publishers, San Mateo, Calif.

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. «Performance tradeoffs in cache design», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, Hawaii, 290-298.

ROBERTS, D., G. TAYLOR, AND T. LAYMAN [1990]. «An ECL RISC microprocessor designed for two-level cache», *IEEE Compcon* (February).

SAMPLES, A. D. AND P. N. HILFINGER [1988]. «Code reorganization for instruction caches», Tech. Rep. UCB/CSD 88/447 (October), Univ. of Calif., Berkeley.

SITES, R. L. [1979]. «How to use 1000 registers», *Caltech Conf. on VLSI* (January).

SMITH, A. J. [1982]. «Cache memories». *Computing Surveys* 14:3 (September) 473-530.

SMITH, A. J. [1986]. «Bibliography and readings on CPU cache memories and related topics», *Computer Architecture News* (January) 22-42.

SMITH, J. E. AND J. R. GOODMAN [1983]. «A study of instruction cache organizations and replacement policies», *Proc. Annual Symposium on Computer Architecture* (June 5-7), Stockholm, Sweden., 132-137.

STRECKER, W. D. [1976]. «Cache memories for the PDP-11?», *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 155-158.

TANG, C. K. [1976]. «Cache system design in the tightly coupled multiprocessor system», *Proc. 1976 AFIPS National Computer Conf.*, 749-753.

TAYLOR, G. S., P. N. HILFINGER, J. R. LARUS, D. A. PATTERSON, AND B. G. ZORN [1986]. «Evaluation of the SPUR Lisp architecture», *Proc. 13th Annual Symposium on Computer Architecture* (June 2-5), Tokyo, Japan, 444-452.

THACKER, C. P. AND L. C. STEWART [1987]. «Firefly: a multiprocessor workstation», *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 164-172.

UNGAR, D. M. [1987]. *The Design of a High Performance Samlltalk System,* The MIT Press Distinguished Dissertation Series, Cambridge, Mass.

WANG, W.-H., J.-L. BAER, AND H. M. LEVY [1989]. «Organization and performance of a two-level virtual-real cache hierarchy», *Proc. 16th Annual Symposium on Computer Architecture* (May 28-June 1), Jerusalem, Israel, 140-148.

WILKES, M. [1965]. «Slave memories and dynamic storage allocation», *IEEE Trans. Electronic Computers* EC-14:2 (April) 270-271.

WILKES, M. V. [1982]. «Hardware support for memory protection: Capability implementations», *Proc. Sumposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 107-116.

WULF, W. A., R. LEVIN AND S. P. HARBISON [1981]. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, Nęw York.

EJERCICIOS

8.1 [15/15/12/12] <2.2,8.4> Tratemos de mostrar cómo se pueden hacer benchmarks *injustos.* Hay dos máquinas con el mismo procesador y memoria principal pero diferentes organizaciones de cache. Suponer que el tiempo de fallos es 10 veces el tiempo de un acierto cache para ambas máquinas. Suponer que la escritura de una palabra de 32 bits necesita 5 veces el tiempo de un acierto de cache (para la cache de escritura-directa), y que una escritura completa de un bloque de 16 bytes necesita 10 veces el tiempo de lectura de cache (para la cache de postescritura). Las caches son unificadas; es decir, contienen instrucciones y datos.

Cache A: 64 conjuntos, 2 elementos por conjunto, cada bloque es de 16 bytes, y utiliza escritura directa.

Cache B: 128 conjuntos, 1 elemento por conjunto, cada bloque es de 16 bytes, y utiliza postescritura.

a) [15] Describir un programa que haga que la máquina A corra lo más rápido posible en comparación a la máquina B. (Asegurarse de plantear todas las hipótesis que se necesiten, si las hay.)

	Con NOPS	Sin NOPS	Relación con/sin
Referencias totales	1 500 000	1 180 000	1,27
Fallos de cache	34 153	24 908	1,37
Frecuencia de fallos	2,28	2,10	1,09

FIGURA 8.53 Frecuencias de fallos de Spice con y sin NOP.

b) [15] Describir un programa que haga que la máquina B corra lo más rápido posible en comparación a la máquina A. (Asegurarse de plantear todas las hipótesis que se necesiten, si las hay.)

c) [12] Aproximadamente ¿cuántas veces es más rápido el programa del apartado a en la máquina A que en la máquina B?

d) [12] Aproximadamente ¿cuántas veces es más rápido el programa del apartado b en la máquina B que en la máquina A?

8.2 [20] <2.2,6.4,8.4> Para simplificar la ejecución segmentada, algunas máquinas insertan instrucciones NOP en lugar de interbloquear la segmentación (ver págs. 293-296 en el Cap. 6). Ignorando los fallos de la cache, suponer que el código Spice necesita 2 000 000 de ciclos de reloj en cualquier caso (como la versión sin NOPS todavía interbloquea, necesita un ciclo extra cada vez.) La Figura 8.53 muestra datos coleccionados para una parte de ejecución de Spice con una cache de instrucciones, de correspondencia directa y 64 KB con bloques de una palabra.
La conclusión de un estudio basado en la Figura 8.53 fue que un incremento de un 9 por 100 en la frecuencia de fallos del programa con NOPS tendrá un impacto pequeño pero medible en el rendimiento. ¿Cuál es el impacto real en el rendimiento suponiendo una penalización de fallos de 10 ciclos de reloj?

8.3 [15/15] <8.4> Se compró un computador Acme con las siguientes características:

1. El 90 por 100 de todos los accesos a memoria se encuentran en la cache;
2. Cada bloque de cache es de dos palabras, y el bloque completo se lee en cualquier fallo;
3. El procesador envía referencias a su cache a la velocidad de 10^7 palabras por segundo;
4. El 25 por 100 de las referencias de (3) son escrituras;
5. Suponer que el bus puede soportar 10^7 lecturas o escrituras de palabras por segundo;
6. El bus lee o escribe una sola palabra cada vez (el bus no puede leer o escribir dos palabras cada vez);
7. Suponer que en cualquier instante de tiempo, el 30 por 100 de los bloques de la cache han sido modificados;
8. La cache utiliza ubicación de escritura en un fallo de escritura.

Se está considerando el añadir un periférico al bus, y se quiere saber qué ancho de banda del bus se está utilizando. Calcular el porcentaje del ancho de banda del bus utilizado en promedio en los dos casos siguientes. El porcentaje en la literatura se denomina *relación de tráfico.* Asegurarse de plantear las hipótesis necesarias.

a) [15] La cache es de escritura directa.

b) [15] La cache es de postescritura.

8.4 [20] <8.4> Un inconveniente del esquema de postescritura es que las escrituras probablemente emplearán dos ciclos. Durante el primer ciclo, se detecta si se presenta un acierto, y durante el segundo (suponiendo un acierto) realmente se escribe el dato. Suponer que el 50 por 100 de los bloques están modificados para una cache de postescritura. Utilizando las estadísticas de las cargas y almacenamientos de DLX en la Figura C.4 del Apéndice C, estimar el rendimiento de una cache de escritura directa con escrituras de un ciclo comparándolo con el de un cache de postescritura con escrituras de dos ciclos para cada uno de los programas. Para esta cuestión, suponer que el buffer de escritura para escritura directa nunca detendrá la CPU (no penalización). Suponer que un acierto de cache emplea 1 ciclo de reloj, la penalización de fallos de cache es de 10 ciclos de reloj, y que una escritura de bloque desde la cache a memoria principal necesita 10 ciclos de reloj. Finalmente, suponer que la frecuencia de fallos de la cache de instrucciones es del 2 por 100 y la de la cache de datos es del 4 por 100.

8.5 [15/20/10] <8.4> Para ahorrar tiempo de desarrollo, la Sun 3/280 y la Sun 4/280 utilizan idénticos sistemas de memoria, aun cuando las CPU son bastante diferentes. Suponer que ocurre lo mismo para una nueva máquina, un modelo utiliza una CPU-VAX y el otro una CPU DLX. Suponer que la información de la frecuencia de fallos de la Figura 8.12 y de la 8.16 se aplican a ambas arquitecturas. Usar la columna de promedio de la Figura C.4 del Apéndice C cuando se necesite para la mezcla de instrucciones de DLX, y el encabezamiento de la Figura 8.16 para la mezcla de instrucciones/datos del VAX. Suponer lo siguiente:

La penalización de fallos es de 12 ciclos de reloj.

Un buffer de escritura ideal que nunca detiene la CPU.

El CPI base, suponiendo un sistema perfecto de memoria, es 6.0 para el VAX y 1.5 para DLX.

Una cache unificada añade 1 ciclo de reloj extra a cada carga y almacenamiento de DLX (ya que hay un único puerto de memoria) pero no para el VAX.

Se están considerando tres opciones:

1. Una cache unificada asociativa por conjuntos de 4 vías de 64 KB.

2. Dos caches asociativas por conjuntos de 2 vías de 32 KB cada una, una para instrucciones y otra para datos.

3. Una cache unificada de correspondencia directa de 128 KB. Suponer que la frecuencia de reloj es el 10 por 100 más rápida en este caso, ya que la correspondencia es directa, y las direcciones de la CPU no necesitan controlar dos caches, ni se necesita multiplexar el bus de datos. Esta frecuencia de reloj más rápida incrementa la penalización de fallos a 13 ciclos de reloj.

a) [15] ¿Cuál es el tiempo medio de acceso a memoria en ciclos de reloj para cada organización?

b) [20] ¿Cuál es el CPI para cada máquina y organización cache?

c) [10] ¿Qué organización cache da el mejor rendimiento medio para las dos CPU?

8.6 [25/15] <2.3,8.4,8.8> Algunos microprocesadores tienen caches a medida, de un solo chip, como compañeros de la CPU. Por ejemplo, la CPU del Motorola 88100 puede

tener hasta 8 chips de cache 88200. Estos chips suelen ser más caros que los chips estándar de RAM estática. El MIPS R3000 incluye un comparador en el chip de la CPU para que los identificadores y datos de la cache se puedan construir a partir de RAM estáticas.

a) [25] Utilizando el programa que analiza las frecuencias de fallos de cache ¿cuántas caches de RAM de 16 K por 4 debe utilizar el R3000 para obtener el mismo rendimiento que dos chips 88200? Ambos diseños utilizan caches separadas de instrucciones y datos. El diseño MIPS supone un tamaño de bloque de 16 bytes con ubicación de subbloques para cada palabra. La cache es de escritura directa con un buffer de escritura de 4 palabras. El Motorola 88200 es asociativa por conjuntos de 4 vías con 16 KB por chip y un bloque de 16 bytes que utiliza reemplazo LRU.

b) [15] A continuación se dan los datos sobre los precios de cada chip (cantidad 1 el 1/8/89):

Motorola 88100: $697
Motorola 88200: $875
MIPS R3000 (25 MHz): $300
MIPS R3010 FPU (25 MHz): $350
16K por 4 SRAM (for 25 MHz R3000): $21

¿Qué sistema será más barato y cuánto?

8.7 [15/25/15/15] <2.3,8.4> El Intel i860 tiene sus caches en el mismo chip que la CPU y el tamaño del dado es 1,2 cm · 1,2 cm. Tiene una cache de instrucciones de 4 KB asociativa por conjuntos de 2 vías, y una cache de datos de 8 KB asociativa por conjuntos de 2 vías que utiliza escritura directa o postescritura. Ambas caches utilizan bloques de 32 bytes. No hay buffers de escritura ni etiquetas de proceso para reducir la limpieza de cache. El i860 también incluye un TLB asociativo por conjuntos de 4 vías y 64 entradas para gestionar sus páginas de 4 KB. La traducción de direcciones se realiza antes que sean accedidas las caches. El tamaño del chip de la CPU 7C601 de Cypress es de 0,8 cm por 0,7 cm y no tiene cache integrada —se ofrece en un chip controlador de cache (7C604) y dos chips de cache de 16 K · 16 (7C157) para formar una cache unificada de 64 KB. El controlador incluye un TLB con 64 entradas gestionado de forma completamente asociativa con 4096 etiquetas de procesos para reducir *(flushing)* las limpiezas. Soporta bloques de 32 bytes con correspondencia directa y bien escritura directa o postescritura. Hay un buffer de escritura de un bloque para postescrituras y un buffer de escritura de cuatro palabras para escrituras directas. Los tamaños de los chips son 1,0 cm por 0,9 cm para el 7C604 y 0,8 cm por 0,7 cm para el 7C157.

a) [15] Utilizando el modelo de costes del Capítulo 2, ¿cuál es el coste del conjunto de chips de Cypress en comparación con el del chip de Intel? (Utilizar la Figura 2.11 para determinar los costes de los chips buscando en esa tabla el tamaño de dado más parecido al área de los dados de Cypress e Intel.)

b) [25] Utilizar el simulador y trazas de las caches de DLX para determinar el tiempo medio de acceso a memoria para cada organización cache. Suponer que un fallo necesita una latencia de 6 ciclos más 1 ciclo por cada palabra de 32 bits. Suponer que ambos sistemas tienen la misma frecuencia de reloj y utilizan ubicación en escritura.

c) [15] ¿Cuál es el la relación coste/rendimiento de estos chips utilizando el tiempo promedio de acceso a memoria como medida?

d) [15] ¿Cuál es el porcentaje de incremento en el coste de una estación de trabajo en color que utilice los chips más caros?

8.8 [25/10/15] <8.4> Los buffers de instrucciones del CRAY X-MP se pueden considerar como caches de sólo instrucciones. El tamaño total es de 1 KB, descompuesto en 4 bloques de 256 bytes por bloque. La cache es completamente asociativa y utiliza una política de reubicación «primero en entrar/primero en salir». El tiempo de acceso en un fallo es de 10 ciclos de reloj, con un tiempo de transferencia de 64 bytes cada ciclo de reloj. La X-MP emplea 1 ciclo de reloj en un acierto. Utilizar el simulador cache y trazas de DLX para determinar:

a) [25] Frecuencia de fallos de instrucciones

b) [10] Tiempo promedio de acceso a memoria de instrucciones medido en ciclos de reloj

c) [15] ¿Cuánto debe valer el CPI del CRAY X-MP para que la parte debida a fallos de la cache de instrucciones sea el 10 por 100 como máximo?

8.9 [25] <8.4> Las trazas de un único proceso dan estimaciones muy altas para caches utilizadas en un entorno multiproceso. Escribir un programa que mezcle las trazas uniproceso de DLX en un único flujo de referencias. Utilizar las estadísticas de cambio de procesos de la Figura 8.25 como la frecuencia media de cambio de procesos con una distribución exponencial sobre esa media. (Utilizar el número de ciclos de reloj en lugar de instrucciones, y suponer que el CPI de DLX es 1.5.) Utilizar el simulador de caches con las trazas originales y con la traza mezclada. ¿Cuál es la frecuencia de fallos para cada caso suponiendo una cache de correspondencia directa de 64 KB con bloques de 16 bytes? (Hay un identificador de proceso en la etiqueta cache para que ésta no tenga que ser limpiada en cada cambio.)

8.10 [25] <8.4> Un enfoque para reducir fallos es prebuscar el siguiente bloque. Una estrategia simple pero efectiva es asegurarse que el bloque $i + 1$ esté en la cache cuando el bloque i es referenciado, y si no, prebuscarlo. ¿Piensa que la prebúsqueda es más o menos efectiva al incrementar el tamaño de bloque? ¿Por qué? ¿Es más o menos efectiva al incrementar el tamaño de la cache? ¿Por qué? Utilizar las estadísticas del simulador y las trazas de la cache para sustentar sus conclusiones.

8.11 [20/25] <8.4> Smith y Goodman [1983] encontraron que para una cache de **sólo pequeñas instrucciones,** una cache de correspondencia directa podía superar en muchos casos a otra completamente asociativa con reemplazo LRU.

a) [20] Explicar por qué será posible esto. (Indicación: esto no se puede explicar con el modelo de las 3C porque se ignora la política de reemplazo.)

b) [25] Utilizar el simulador de cache para ver si se cumplen estos resultados para las trazas.

8.12 [Discusión] <8.4> Si se observan fallos de conflictos para una asociatividad dada en la Figura 8.14, cuando la capacidad se incrementa, los fallos de conflicto suben y bajan. Por ejemplo, para una correspondencia asociativa por conjuntos de 2 vías, la frecuencia de fallos para una cache de 2 KB es 0,10, para otra de 4 KB es de 0,013, y para otra de 8 KB es de 0,008. ¿Por qué ocurre esto?

8.13 [30] <8.5> Utilizar un simulador de la cache y las trazas para calcular la efectividad de una memoria entrelazada de 4 bancos en comparación con una memoria entrelazada de 8 bancos. Suponer que cada transferencia de palabra emplea un ciclo en el bus y que un acceso aleatorio es de 8 ciclos. Medir los conflictos de los bancos y el ancho de banda de memoria para estos casos:

a) No hay cache ni buffer de escritura.

b) Una cache de 64 KB de correspondencia directa y escritura directa con bloques de cuatro palabras.

c) Una cache de 64 KB de correspondencia directa, postescritura con bloques de cuatro palabras.

d) Una cache de 64 KB de correspondencia directa, escritura directa con bloques de cuatro palabras, pero el «entrelazado» proviene de una DRAM en modo página.

e) Una cache de 64 KB de correspondencia directa, postescritura con bloques de cuatro palabras, pero el «entrelazado» proviene de una DRAM en modo página.

8.14 [20] <8.6> Si el CPI base con un sistema perfecto de memoria es de 1,5, ¿cuál es el CPI para estas organizaciones de cache? Utilizar la Figura 8.12:

a) Cache unificada de 16 KB de correspondencia directa utilizando postescritura.

b) Cache unificada de 16 KB, asociativa por conjunto de dos vías utilizando postescritura.

c) Cache unificada de 32 KB de correspondencia directa utilizando postescritura.

Suponer que la latencia de memoria es de 6 ciclos, la velocidad de transferencia es de 4 bytes por ciclo de reloj y que el 50 por 100 de las transferencias están modificadas. Hay 16 bytes por bloque y el 20 por 100 de las instrucciones son instrucciones de transferencia de datos. Las caches buscan las palabras de un bloque según el orden de las direcciones, y la CPU se detiene hasta que no llegan todas las palabras del bloque. No hay buffer de escritura. Añadir a las suposiciones anteriores un TLB que emplea 20 ciclos de reloj en un fallo de TLB. El TLB no ralentiza un acierto de cache. Para el TLB, hacer la suposición simplificada que el 1 por 100 de todas las referencias no se encuentra en el TLB, cuando las direcciones provienen directamente de la CPU o cuando provienen de fallos de la cache. ¿Cuál es el impacto en el rendimiento del TLB si la cache anterior es física o virtual?

8.15 [30] <3.8,8.9> El ejemplo de la Sección 8.9 (pág. 515) mejora la búsqueda de instrucciones de la CPU desde la cache debido al buffer de prebúsqueda de instrucciones. ¿Cómo afecta este incremento del 13 al 39 por 100 de las palabras de instrucción buscadas a la diferencia en las palabras de instrucción buscadas en DLX en comparación con las de VAX? Las búsquedas extra de instrucciones del VAX perjudican sólo cuando traen algo a la cache que no se utilice antes que se saque fuera, mientras DLX parece necesitar una cache mayor para su programa mayor. Escribir un simulador emulando el buffer de prebúsqueda de instrucciones para medir el incremento de fallos de la cache utilizando las trazas de direcciones VAX y ver si prebuscar supone un incremento significativo en los fallos de la cache.

8.16 [25-40] <8.7> Estudiar el impacto de añadir ventanas de registros a DLX. Este estudio puede variar desde estimar simplemente los ahorros del tráfico de registros hasta modificar el compilador y el simulador de DLX para medir directamente costes y beneficios.

8.17 [10] <8.8> Data General describió el diseño de una cache de tres niveles para una implementación ECL de la arquitectura 88000. ¿Cuál es la fórmula para el tiempo medio de acceso para una cache de tres niveles?

8.18 [20] <8.8> ¿Cuál es la pérdida de rendimiento para un multiprocesador de cuatro vías con dispositivos de E/S? Suponer que el 1 por 100 de todas las referencias de datos a la cache invalidan las otras caches de datos y que todas las CPU se detienen cuatro ciclos de reloj durante una invalidación. Suponer una cache de correspondencia directa de 64 KB para los datos, y otra igual para las instrucciones, con un tamaño de bloque de 32 bytes con una frecuencia de fallos de un 1 por 100 para instrucciones y de un 2 por 100 para datos, siendo del 20 por 100 de todas las referencias de CPU a memoria para datos. El CPI de la CPU es 1,5 con un sistema perfecto de memoria y emplea 10 relojes en un fallo de la cache tanto si el dato está modificado como limpio.

8.19 [25] <8.8> Utilizar las trazas para calcular la efectividad de rearranques anticipados y búsqueda fuera de orden. ¿Cuál es la distribución de los primeros accesos a un bloque cuando el tamaño de bloque aumenta desde 2 a 64 palabras en factores de dos para:

a) ¿Una cache de sólo instrucciones de 64 KB?

b) ¿Una cache de sólo datos de 64 KB?

c) ¿Una cache unificada de 128 KB?

Suponer ubicación de correspondencia directa.

8.20 [30] <8.8> Utilizar el simulador de la cache y las trazas con un programa que usted haya escrito para comparar los esquemas de efectividad para escrituras rápidas:

a) Un buffer de 1 palabra y la CPU se detienen en un fallo de lectura de datos de la cache, con una cache de escritura directa.

b) Un buffer de 4 palabras y la CPU se detienen en un fallo de lectura de datos de la cache, con una cache de escritura directa.

c) Un buffer de 4 palabras y la CPU se detienen en un fallo de lectura de datos de la cache sólo si hay un conflicto potencial en las direcciones, con una cache de escritura directa.

d) Una cache de postescritura que escribe primero datos modificados y después carga el bloque que causó el fallo.

e) Una cache de postescritura con un buffer de escritura de un bloque, que primero carga el dato que falló y después detiene la CPU en un fallo de lectura si el buffer de escritura no está vacío.

f) Una cache de postescritura con un buffer de escritura de un bloque, que carga primero el dato que falló y después detiene la CPU en un fallo de lectura sólo si el buffer de escritura no está vacío y hay un conflicto potencial en las direcciones.

Suponer una cache de correspondencia directa de 64 KB para los datos y otra igual para las instrucciones con un tamaño de bloque de 32 bytes. El CPI de la CPU es 1,5 con un sistema perfecto de memoria y emplea 14 ciclos en un fallo de cache y 7 ciclos para escribir una palabra en memoria.

8.21 [30] <8.8> Utilizar el simulador de la cache y las trazas junto con un programa que usted haya escrito para crear un simulador cache de dos niveles. Utilizar este programa para ver cuál es el tamaño de la cache de segundo nivel que tenga aproximadamente la misma frecuencia de fallos globales que una cache de un nivel de la misma capacidad.

8.22 [Discusión] <8.6> Algunas personas han argumentado que con la creciente capacidad de memoria por chip, la memoria virtual es una idea pasada de moda, y esperan verla caer en los computadores del futuro. Encontrar razones a favor y en contra de este argumento.

8.23 [Discusión] <8.6> Hasta ahora, pocos sistemas computadores aprovechan la seguridad extra disponible con las puertas y anillos que se encuentran en una máquina como el Intel 80286. Construir algún guión en el que la industria de computadores pueda utilizar este modelo de protección.

8.24 [Discusión] <8.4> Recientes investigadores han tratado de utilizar compiladores para mejorar el rendimiento de las cache (ver McFarling [1989] y Samples and Hilfinger [1988]):

a) ¿Cuáles de las 3C intentan los compiladores mejorar y cuáles no? ¿Por qué?

b) ¿Qué correspondencia es la óptima para las mejoras del compilador? ¿Por qué?

8.25 [Discusión] <8.3> Suponer que se ha inventado una nueva tecnología de la que se esperan obtener cambios importantes en la jerarquía de memoria. Para los propósitos de esta pregunta, supongamos que la tecnología de computadores biológicos se convierte en realidad. Suponer que la tecnología de memorias biológicas tiene una característica inusual: es tan rápida como las DRAM semiconductoras más rápidas, y puede ser accedida aleatoriamente; pero sólo cuesta como la memoria de discos magnéticos. Tiene la ventaja adicional de que no se ralentiza sin que importe su tamaño. La única desventaja que tiene es que sólo se puede Escribir Una Vez (Write it Once), pero se puede Leer Muchas veces (Read it Many times). Por tanto se denomina memoria «WORM». Debido a la forma en que se fabrica, el módulo de memoria WORM se puede reemplazar fácilmente. Intente aportar algunas nuevas ideas para aprovechar las WORM para construir mejores computadores utilizando «biotecnología».

Las E/S, ciertamente, se han quedado atrás en la última década.

Seymour Cray, *Conferencia pública* (1976)

También las E/S necesitan mucho trabajo

David Kuck, nota a pie de página
XV Symposium anual sobre arquitectura de computadores (1988)

9.1 Introducción

9.2 Predicción del rendimiento del sistema

9.3 Medidas de rendimiento de E/S

9.4 Tipos de dispositivos de E/S

9.5 Buses. Conectando dispositivos de E/S a CPU/Memoria

9.6 Interfaz con la CPU

9.7 Interfaz con un sistema operativo

9.8 Diseño de un sistema de E/S

9.9 Juntando todo: el subsistema de almacenamiento IBM 3990

9.10 Falacias y pifias

9.11 Observaciones finales

9.12 Perspectiva histórica y referencias

Ejercicios

9 Entradas/salidas

9.1 Introducción

Las entradas/salidas han sido el huérfano de la arquitectura de computadores. Históricamente despreciadas por los entusiastas de la CPU, el prejuicio contra las E/S se institucionaliza en las medidas de rendimiento más ampliamente utilizadas, el tiempo de CPU. Si un computador tiene el mejor o el peor sistema de E/S en el mundo no se puede medir por el tiempo de CPU, que por definición ignora las E/S. El ciudadano de segunda clase que son las E/S es incluso aparente en la etiqueta «periférico» aplicada a los dispositivos de E/S.

Esta actitud se contradice por sentido común. Un computador sin dispositivos de E/S es como un automóvil sin ruedas —no se puede ir muy lejos sin ellas—. Y aunque el tiempo de CPU es interesante, el tiempo de respuesta —el tiempo desde que el usuario escribe una orden hasta que obtiene los resultados— es seguramente una mejor medida del rendimiento. El cliente que paga un computador se preocupa por el tiempo de respuesta, aun cuando el diseñador de CPU no lo haga. Finalmente, como las rápidas mejoras en el rendimiento de CPU acercan las tradicionales familias o clases de computadores, son las E/S las que sirven para distinguirlos:

- La diferencia entre un computador grande y un minicomputador es que, un computador grande, puede soportar muchos más terminales y discos.
- La diferencia entre un minicomputador y una estación de trabajo es que, una estación de trabajo, tiene pantalla, teclado y ratón.

- La diferencia entre un servidor de ficheros y una estación de trabajo es que el servidor de ficheros tiene discos y unidades de cinta pero no pantalla, teclado ni ratón.
- La diferencia entre una estación de trabajo y un computador personal es que las estaciones de trabajo están siempre conectadas entre ellas por medio de una red.

Puede llegar a ocurrir que los computadores desde las estaciones de trabajo de altas prestaciones a los supercomputadores de bajas prestaciones utilicen los mismos «super-microprocesadores». Las diferencias en coste y rendimiento estarían determinadas solamente por los sistemas de memoria y de E/S (y el número de procesadores).

La venganza de las E/S está cerca. Suponer que tenemos una diferencia entre el tiempo de CPU y el tiempo de respuesta de un 10 por 100, y aceleramos la CPU en un factor de 10, mientras que despreciamos las E/S. La Ley de Amdahl nos dice que podemos obtener una aceleración de sólo 5 veces mayor, malgastando la mitad del potencial de CPU. Análogamente, haciendo la CPU 100 veces más rápida sin mejorar las E/S se obtendría una aceleración 10 veces mayor, malgastando el 90 por 100 del potencial. Si, como se predijo en el Capítulo 1, el rendimiento de las CPU mejora del 50 al 100 por 100 por año, y las E/S no mejoran, cada tarea estará limitada por las E/S. No habrá razón para comprar CPU más rápidas —y no habría trabajos para los diseñadores de CPU.

Aunque este simple capítulo no pueda reivindicar completamente las E/S, como mínimo puede expiar algunos de los pecados del pasado y restablecer un cierto equilibrio.

¿Están siempre inactivas las CPU?

Algunos sugieren que el prejuicio está bien fundamentado. La velocidad de E/S no importa, arguyen, ya que siempre hay un proceso para ejecutar mientras otro está esperando un periférico.

Hay varios puntos para replicar esto. Primero, este es un argumento que mide el rendimiento como productividad —más tareas por hora— en lugar de como tiempo de respuesta. Evidentemente, si los usuarios no hubiesen cuidado el tiempo de respuesta, el software interactivo nunca se habría inventado y no habría estaciones de trabajo hoy día. (La sección siguiente da evidencia experimental de la importancia del tiempo de respuesta.) También puede ser caro confiar en procesos mientras están esperando las E/S, ya que la memoria principal debe ser mayor o el tráfico de paginación por la conmutación de procesos realmente incrementaría las E/S. Además, con los computadores de despacho sólo hay una persona por CPU y, por tanto, menos procesos que en tiempo compartido: ¡muchas veces el único proceso que espera es el ser humano! Y algunas aplicaciones, tales como tratamiento de transacciones (Sección 9.3), ponen límites estrictos en el tiempo de respuesta, como parte del análisis de rendimiento.

Pero aceptemos el argumento y explorémoslo. Suponer que la diferencia

entre el tiempo de respuesta y el tiempo de CPU es del 10 por 100, y que una CPU que es diez veces más rápida puede conseguirse sin cambiar el rendimiento de E/S. Un proceso entonces empleará el 50 por 100 de su tiempo esperando las E/S, y dos procesos tendrán que estar perfectamente alineados para evitar paradas de la CPU mientras están esperando las E/S. Cualquier mejora adicional de la CPU sólo incrementará el tiempo de inactividad de la CPU.

Por tanto, la productividad *(throughput)* de las E/S puede limitar la productividad del sistema, de la misma forma que el tiempo de respuesta de la E/S limita el tiempo de respuesta del sistema. Veamos cómo predecir el rendimiento del sistema completo.

9.2 Predicción del rendimiento del sistema

El rendimiento del sistema está limitado por la parte más lenta del camino entre la CPU y los dispositivos de E/S. El rendimiento de un sistema puede estar limitado por la velocidad de cualquiera de estas partes del camino, mostrados en la Figura 9.1:

- CPU
- Memoria cache
- Memoria principal
- Memoria: bus de E/S

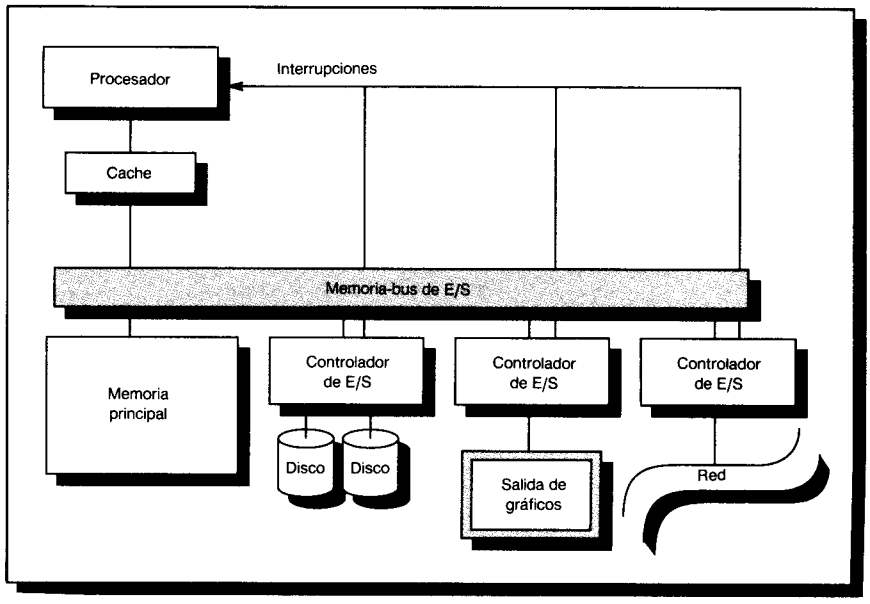


FIGURA 9.1 Colección típica de dispositivos de E/S en un computador.

- Controlador de E/S o canal de E/S
- Dispositivo de E/S
- Velocidad del software de E/S
- Eficiencia del uso del software de los dispositivos de E/S

Si el sistema no está equilibrado, el elevado rendimiento de algunos componentes puede perderse debido al bajo rendimiento de un eslabón de la cadena. El arte del diseño de E/S es configurar un sistema tal que la velocidad de todos los componentes sea la misma.

En los capítulos anteriores hemos asumido que tener CPU más rápidas era el único objeto de nuestro deseo, pero el rendimiento de CPU no es lo mismo que el rendimiento del sistema. Por ejemplo, supongamos que tenemos dos cargas de trabajo, A y B. Ambas cargas tardan diez segundos en ejecutarse. La carga de trabajo A lo hace con pocas E/S que no son dignas de mención. La carga de trabajo B mantiene los dispositivos de E/S ocupados cuatro segundos, y este tiempo se solapa completamente con actividades de la CPU. Suponer que la CPU se sustituye por un nuevo modelo cuyo rendimiento es cinco veces mayor. Intuitivamente, comprendemos que la carga de trabajo A tarda dos segundos —cinco veces más rápido— pero la carga de trabajo B está limitada por las E/S y no puede tardar menos de cuatro segundos. La Figura 9.2 ilustra nuestra intuición.

Para determinar el rendimiento de estos casos se requiere una nueva fórmula. El tiempo transcurrido *(elapsed time)* en la ejecución de una carga de trabajo puede descomponerse en tres partes

$$\text{Tiempo}_{\text{carga de trabajo}} = \text{Tiempo}_{\text{CPU}} + \text{Tiempo}_{\text{E/S}} - \text{Tiempo}_{\text{solapamiento}}$$

donde $\text{Tiempo}_{\text{CPU}}$ significa el tiempo que la CPU está ocupada, $\text{Tiempo}_{\text{E/S}}$ significa el tiempo que está ocupado el sistema de E/S y $\text{Tiempo}_{\text{solapamiento}}$ significa el tiempo que la CPU y el sistema de E/S están ambos ocupados. Utilizando, como ejemplo, la carga de trabajo B con la antigua CPU de la Figura 9.2, los tiempos en segundos son:

10 para $\text{Tiempo}_{\text{carga de trabajo}}$,

10 para $\text{Tiempo}_{\text{CPU}}$,

4 para $\text{Tiempo}_{\text{E/S}}$ y

4 para $\text{Tiempo}_{\text{solapamiento}}$.

Suponiendo que sólo mejoramos la CPU, una forma de calcular el tiempo para ejecutar la carga de trabajo es:

$$\text{Tiempo}_{\text{carga de trabajo}} = \frac{\text{Tiempo}_{\text{CPU}}}{\text{mejora}_{\text{CPU}}} + \text{Tiempo}_{\text{E/S}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{mejora}_{\text{CPU}}}$$

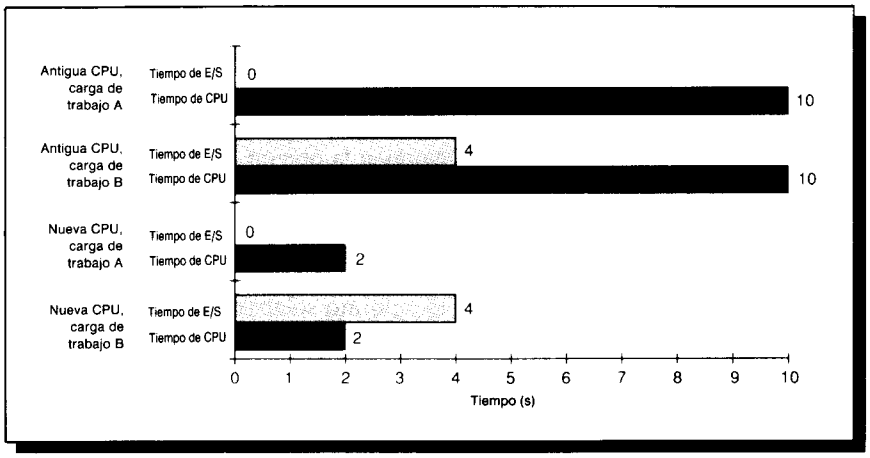


FIGURA 9.2 La ejecución solapada de las dos cargas de trabajo en la CPU original y después en una CPU cinco veces más rápida. Podemos ver que el tiempo transcurrido para la carga de trabajo A es 1/5 del tiempo con la nueva CPU, pero está limitado a cuatro segundos en la carga de trabajo B debido a que no se mejora la velocidad de E/S.

Como el tiempo de CPU disminuye, también disminuye el tiempo de solapamiento. La aceleración del sistema cuando queremos mejorar E/S es equivalente:

$$\text{Tiempo}_{\text{carga de trabajo}} = \text{Tiempo}_{\text{CPU}} + \frac{\text{Tiempo}_{\text{E/S}}}{\text{mejora}_{\text{E/S}}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{mejora}_{\text{E/S}}}$$

Tratemos un ejemplo antes de explicar una limitación de estas fórmulas.

Ejemplo

Una carga de trabajo emplea 50 segundos en ejecutarse, estando la CPU ocupada 30 segundos y la E/S 30 segundos. ¿Cuánto tiempo tardará en ejecutarse la carga de trabajo si se sustituye la CPU por otra que cuadruplique el rendimiento?

Respuesta

El tiempo de total transcurrido es de 50 segundos, y la suma del tiempo de CPU y de E/S es 60 segundos. Por tanto, el tiempo de solapamiento debe ser 10 segundos. Acudiendo a la fórmula anterior:

$$\text{Tiempo}_{\text{carga de trabajo}} = \frac{\text{Tiempo}_{\text{CPU}}}{\text{mejora}_{\text{CPU}}} + \text{Tiempo}_{\text{E/S}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{mejora}_{\text{CPU}}} =$$

$$= \frac{30}{4} + 30 - \frac{10}{4} = 35$$

Este ejemplo descubre un problema con esta fórmula: ¿qué parte del tiempo que la carga de trabajo está ocupada en la CPU más rápida se solapa

con la E/S? La Figura 9.3 muestra tres opciones. Dependiendo del solapamiento resultante después de mejorar la CPU, el tiempo para la carga de trabajo varía de 30 a 37,5 segundos.

En realidad no podemos conocer lo que es correcto, sin medir la carga de trabajo en la CPU más rápida, para ver qué solapamiento existe. Las fórmulas anteriores suponen la opción (c) de la Figura 9.3; el solapamiento se escala a la misma velocidad que la CPU; por tanto, lo llamaremos $\text{Tiempo}_{\text{escalado}}$ (en lugar de $\text{Tiempo}_{\text{carga de trabajo}}$). El máximo solapamiento supone que se mantiene tanto solapamiento como sea posible, pero el nuevo solapamiento no puede ser mayor que el solapamiento original o el tiempo de CPU después de mejorarla. El mínimo solapamiento supone que se elimina el mayor solapamiento posible, pero que el tiempo de solapamiento no disminuirá más del tiempo eliminado de CPU o de E/S. Si introducimos las abreviaturas $\text{Nuevo}_{\text{CPU}} = \text{Tiempo}_{\text{CPU}}/\text{mejora}_{\text{CPU}}$ y $\text{Nuevo}_{\text{E/S}} = \text{Tiempo}_{\text{E/S}}/\text{mejora}_{\text{E/S}}$, el tiempo de la carga de trabajo para máximo solapamiento ($\text{Tiempo}_{\text{mejor}}$) y mínimo solapamiento ($\text{Tiempo}_{\text{peor}}$) puede escribirse como:

$$\text{Tiempo}_{\text{mejor}} = \text{Nuevo}_{\text{CPU}} + \text{Tiempo}_{\text{E/S}} - \\ - \text{Mínimo}(\text{Tiempo}_{\text{solapamiento}}, \text{Nuevo}_{\text{CPU}})$$

$$\text{Tiempo}_{\text{peor}} = \text{Nuevo}_{\text{CPU}} + \text{Tiempo}_{\text{E/S}} - \\ - \text{Máximo}(0, \text{Tiempo}_{\text{solapamiento}} - (\text{Tiempo}_{\text{CPU}} - \text{Nuevo}_{\text{CPU}}))$$

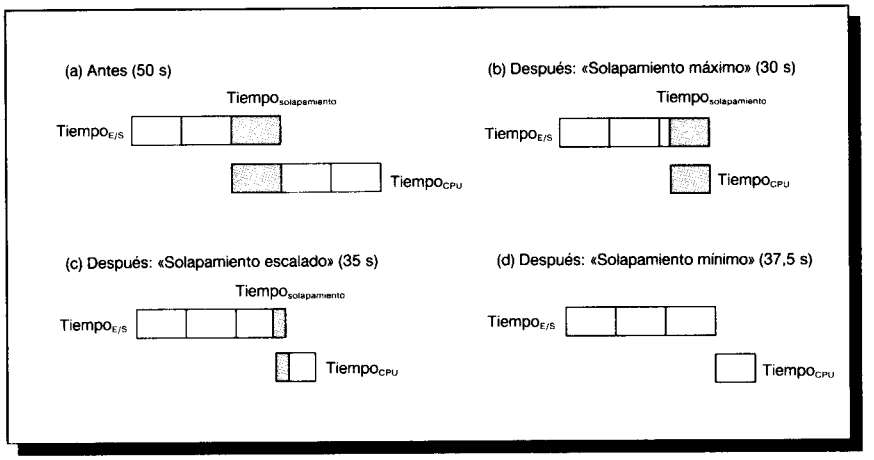


FIGURA 9.3 El solapamiento original en el ejemplo anterior (a) y tres interpretaciones del solapamiento después de mejorar la CPU. Cada bloque representa diez segundos, excepto el bloque de la nueva CPU que es de 7,5 segundos. Las partes solapadas de $\text{Tiempo}_{\text{CPU}}$ y $\text{Tiempo}_{\text{E/S}}$ están sombreadas. (b) Muestra el nuevo solapamiento del $\text{Tiempo}_{\text{CPU}}$ completamente con E/S, dando un tiempo de carga de trabajo de treinta segundos. (c) Muestra el solapamiento del $\text{Tiempo}_{\text{CPU}}$ escalado con $\text{Velocidad}_{\text{CPU}}$, dando un total de treinta y cinco segundos, con 2,5 segundos de ejecución solapada. (d) Muestra no solapamiento con E/S; así, el total es 37,5 segundos.

Ejemplo

Calcular las tres predicciones de tiempo para la carga de trabajo B de la Figura 9.2.

Respuesta

$$\text{Tiempo}_{\text{mejor}} = \frac{10}{5} + 4 - \text{Mínimo}\left(\frac{10}{5}, 4\right) = 2 + 4 - 2 = 4$$

$$\text{Tiempo}_{\text{escalado}} = \frac{10}{5} + 4 - \frac{4}{5} = 2 + 4 - 0{,}8 = 5{,}2$$

$$\text{Tiempo}_{\text{peor}} = \frac{10}{5} + 4 - \text{Máximo}(0, 4 - (10 - \frac{10}{5})) = 2 + 4 - 0 = 6$$

A veces,los cambios se harán en la CPU y en el sistema de E/S. Las fórmulas se convierten en:

$$\text{Tiempo}_{\text{escalado}} = \text{Nuevo}_{\text{CPU}} + \text{Nuevo}_{\text{E/S}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{Máximo}(\text{mejora}_{\text{CPU}}, \text{mejora}_{\text{E/S}})}$$

$$\text{Tiempo}_{\text{mejor}} = \text{Nuevo}_{\text{CPU}} + \text{Nueva}_{\text{E/S}} - \text{Mínimo}(\text{Tiempo}_{\text{solapamiento}}, \text{Nuevo}_{\text{CPU}}, \text{Nuevo}_{\text{E/S}})$$

$$\text{Tiempo}_{\text{peor}} = \text{Nuevo}_{\text{CPU}} + \text{Nuevo}_{\text{E/S}} - \text{Máx}(0, \text{Tiempo}_{\text{solapado}} - \text{Máx}(\text{Tiempo}_{\text{CPU}} - \text{Nuevo}_{\text{CPU}}, \text{Tiempo}_{\text{E/S}} - \text{Nuevo}_{\text{E/S}}))$$

La fórmula para el solapamiento escalado indica que el período de solapamiento se reduce por la mayor de las dos velocidades mejoradas. La fórmula para el solapamiento máximo ($\text{Tiempo}_{\text{mejor}}$) indica que se retiene tanto solapamiento como es posible, pero que el nuevo solapamiento no puede ser mayor que el solapamiento original o el tiempo de CPU o de E/S después de las mejoras. Finalmente, la fórmula para el solapamiento mínimo ($\text{Tiempo}_{\text{peor}}$) dice que el solapamiento se reduce por el mayor de los tiempos cambiados del tiempo de CPU y del tiempo de E/S (pero el tiempo de solapamiento no puede ser menor que 0). La Figura 9.4 muestra los tres ejemplos de aceleración donde se mejoran la E/S y CPU.

Veamos un ejemplo detallado cuando se mejoran la velocidad de la CPU y de la E/S.

Ejemplo

Supongamos que una carga de trabajo en los sistemas actuales tarda 64 segundos. La CPU está ocupada el tiempo completo, y los canales que conectan los dispositivos de E/S a la CPU están ocupados 36 segundos. El gestor del computador está considerando dos opciones de modernización: bien una sola CPU que tenga el doble de rendimiento, o dos CPU que tengan doble productividad y el doble número de canales. El tiempo de los dispositivos reales de E/S es tan pequeño que se puede ignorar. Para la opción de doble CPU suponer que la carga de trabajo se puede dividir uniformemente entre las CPU y los canales. ¿Cuál es la mejora de rendimiento para cada opción?

Respuesta

Como no hay cambio en el sistema de E/S con la CPU más rápida, el tiempo para la carga de trabajo, suponiendo el solapamiento escalado, es sencillamente

$$\text{Tiempo}_{\text{escalado}} = \frac{\text{Tiempo}_{\text{CPU}}}{\text{mejora}_{\text{CPU}}} + \text{Tiempo}_{\text{E/S}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{mejora}_{\text{CPU}}}$$

$$= \frac{64}{2} + 36 - \frac{36}{2} = 32 + 36 - 18 = 50$$

Para la doble CPU con más canales,

$$\text{Tiempo}_{\text{escalado}} =$$

$$\frac{\text{Tiempo}_{\text{CPU}}}{\text{velocidad}_{\text{CPU}}} + \frac{\text{Tiempo}_{\text{E/S}}}{\text{velocidad}_{\text{E/S}}} -$$

$$- \frac{\text{Tiempo}_{\text{solapado}}}{\text{Máximo}(\text{velocidad}_{\text{CPU}}, \text{velocidad}_{\text{E/S}})} =$$

$$= \frac{64}{2} + \frac{36}{2} - \frac{36}{\text{Máximo}(2,2)} = 32 + 18 - 18 = 32$$

Suponiendo solapamiento escalado, la doble CPU es más del 50 por 100 más rápida. Usando el escalamiento en el mejor caso, la doble CPU es el 13 por 100 más rápida, mientras que el escalamiento en el peor caso, sugiere que es el 39 por 100 más rápida.

Como demuestran estos ejemplos, necesitamos mejorar en el rendimiento de E/S para igualar las mejoras en el rendimiento de la CPU si pretendemos conseguir sistemas de computadores más rápidos. Ahora examinemos métricas de dispositivos de E/S para comprender cómo mejorar su rendimiento y, por tanto, el sistema completo.

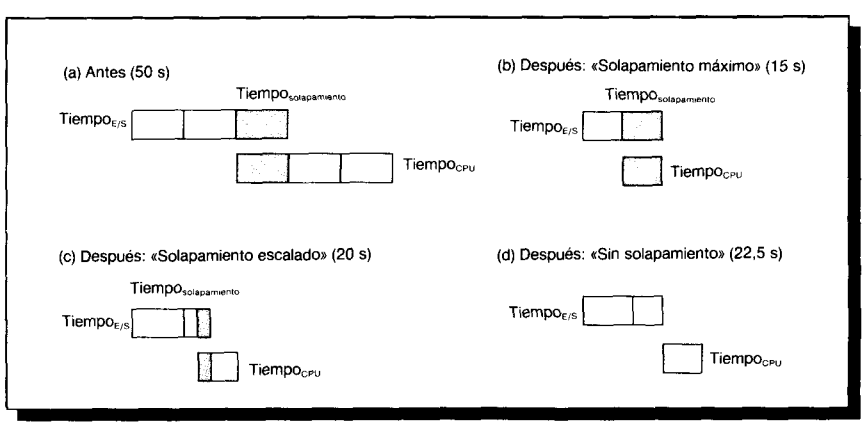


FIGURA 9.4 Tiempo para la carga de trabajo de la Figura 9.3(a) con Velocidad$_{CPU}$ = 4 y Velocidad$_{E/S}$ = 2.

9.3 Medidas de rendimiento de E/S

El rendimiento de E/S tiene medidas que no tienen contrapartida en el diseño de CPU. Una de éstas es la diversidad: ¿Qué dispositivos de E/S pueden conectarse al sistema computador? Otra es la capacidad: ¿Cuántos dispositivos de E/S pueden conectarse a un sistema computador?

Además de estas medidas únicas, las medidas tradicionales de rendimiento, tiempo de respuesta y productividad también se aplican a las E/S. (La productividad de E/S a veces se denomina «anchura de banda de E/S» y el tiempo de respuesta a veces se denomina «latencia».) Las dos siguientes figuras dan una idea de cómo están relacionados la productividad y el tiempo de respuesta. La Figura 9.5 muestra el sencillo modelo productor-servidor. El productor crea tareas para que sean ejecutadas y las coloca en la cola; el servidor toma tareas de la cola y las ejecuta.

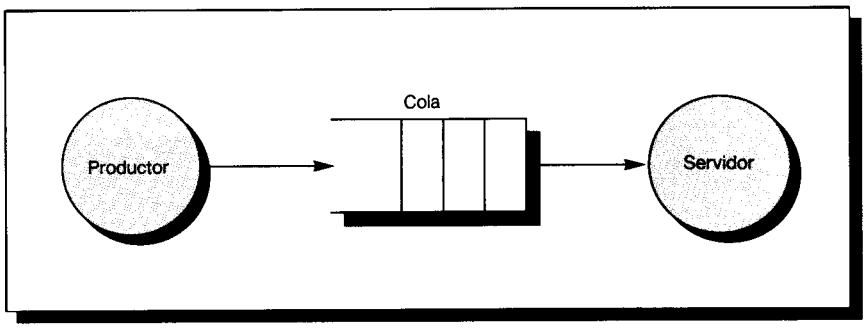


FIGURA 9.5 El modelo tradicional productor-servidor del tiempo de respuesta y productividad. El tiempo de respuesta comienza cuando una tarea se coloca en la cola y finaliza cuando la completa el servidor. El rendimiento es el número de tareas realizadas por el servidor en la unidad de tiempo.

El tiempo de respuesta se define como el tiempo que tarda una tarea desde el instante que se coloca en la cola hasta que la termina el servidor. La productividad es sencillamente el número medio de tareas completadas por el servidor en un período de tiempo. Para lograr la mayor productividad posible, el servidor nunca deberá estar desocupado y, por tanto, la cola nunca deberá estar vacía. Por otro lado, el tiempo de respuesta es el tiempo empleado en la cola y, por tanto, se minimiza al ir vaciando la cola.

Otra medida del rendimiento de las E/S es la interferencia de las E/S con la ejecución de la CPU. La transferencia de datos puede interferir con la ejecución de otro proceso. También hay un gasto adicional debido al tratamiento de interrupciones de E/S. Nuestro interés aquí es el número de ciclos de reloj que empleará un proceso a causa de las E/S de otro proceso.

Productividad frente a tiempo de respuesta

La Figura 9.6 muestra la productividad frente al tiempo de respuesta (o latencia), para un sistema típico de E/S. El codo de la curva es el área donde se logra un poco más de productividad con un tiempo de respuesta mucho mayor o, inversamente, un tiempo de respuesta un poco más corto da como resultado mucha menor productividad.

La vida sería más sencilla si, mejorar el rendimiento, siempre significase mejoras en el tiempo de respuesta y en la productividad. Añadir más servidores, como en la Figura 9.7, incrementa la productividad: difundiendo datos a través de dos discos en lugar de uno, las tareas se pueden servir en paralelo. Desafortunadamente, esto no ayuda al tiempo de respuesta, a menos que la carga de trabajo se mantenga constante y el tiempo de las colas se reduzca porque haya más recursos.

¿Cómo equilibra el arquitecto estas demandas de conflictos? Si el computador está interactuando con seres humanos, la Figura 9.8 sugiere una respuesta. Esta figura presenta los resultados de dos estudios de entornos interactivos, uno orientado al teclado y otro gráfico. Una interacción o *transacción* con un computador se divide en tres partes:

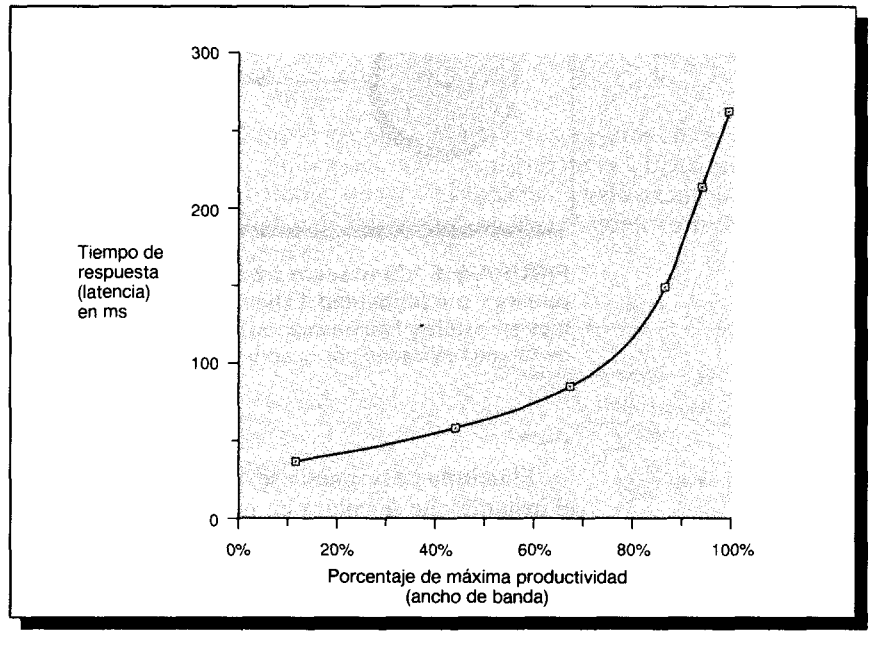


FIGURA 9.6 Productividad frente a tiempo de respuesta. La latencia normalmente se considera como tiempo de respuesta. Observar que el tiempo mínimo absoluto de respuesta consigue solamente el 11 por 100 de productividad mientras que el tiempo de respuesta para la productividad del 100 por 100 necesita siete veces el tiempo de respuesta mínimo. Chen [1989] coleccionó estos datos para un array de discos magnéticos.

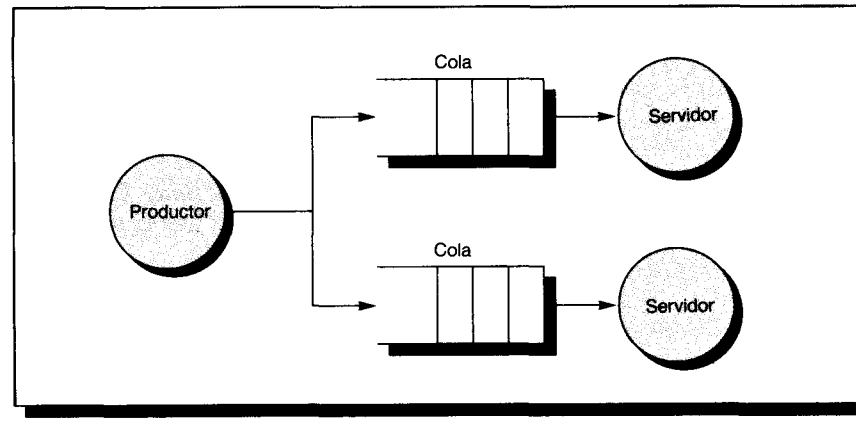


FIGURA 9.7 El modelo de un solo productor, un solo servidor de la Figura 9.5 se amplía con otro servidor y otra cola. Esto incrementa la productividad del sistema de E/S y necesita menos tiempo para servir las tareas del productor. Incrementar el número de servidores es una técnica común en sistemas de E/S. Hay un problema potencial de desequilibrio con dos colas; a menos que los datos se coloquen perfectamente en las colas, algunas veces algún servidor estará parado con una cola vacía mientras el otro estará ocupado con muchas tareas en la cola.

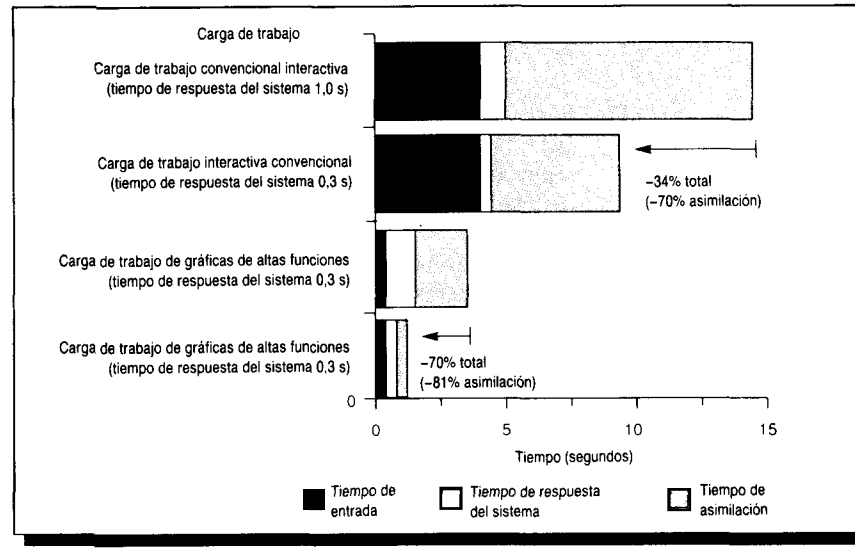


FIGURA 9.8 Una transacción de usuario con un computador interactivo dividida en tiempo de entrada, tiempo de respuesta del sistema y tiempo de asimilación del usuario para un sistema convencional y un sistema de gráficos. Los tiempos de entrada son los mismos independientemente del tiempo de respuesta del sistema. El tiempo de entrada es de cuatro segundos para el sistema convencional y de 0,25 segundos para el sistema de gráficos. (De Brady [1986].)

1. *Tiempo de entrada.* El tiempo para que el usuario introduzca la orden. En el sistema de gráficos de la Figura 9.8, emplea 0,25 segundos de media en introducir la orden, frente a 4,0 segundos para el sistema convencional.
2. *Tiempo de respuesta del sistema.* El tiempo desde que el usuario introduce la orden hasta que se visualiza la respuesta completa.
3. *Tiempo de asimilación (think time).* El tiempo que transcurre desde la recepción de la respuesta hasta que el usuario comienza a introducir la siguiente orden.

La suma de estas tres partes se denomina *tiempo de transacción.* Algunos estudios informan que la productividad del usuario es inversamente proporcional al tiempo de transacción; las transacciones por hora miden el trabajo completado por hora por el usuario.

Los resultados de la Figura 9.8 muestran que la reducción del tiempo de respuesta hace que disminuya el tiempo de transacción en una cantidad superior a la reducción del tiempo de respuesta: acortar el tiempo de respuesta del sistema en 0,7 segundos ahorra 4,9 segundos (34 por 100) de la transacción convencional y 2,0 segundos (70 por 100) de la transacción de gráficos. Este resultado poco plausible lo explica la naturaleza humana; las personas necesitan menos tiempo de asimilación cuando se da una respuesta más rápida.

Estos resultados se pueden explicar o por las características del funcionamiento de la atención humana o por su forma de trabajar, según confirman varios estudios. En efecto, cuando las respuestas del computador caen por debajo de un segundo, la productividad parece que se incrementa enormemente. La Figura 9.9 compara las transacciones por hora (el inverso del tiempo de transacción) de un principiante, un ingeniero medio y un experto en realizar trabajos de diseño físico en pantallas gráficas. El tiempo de respuesta del sistema aumenta el talento: un principiante con tiempos de respuesta de subsegundos era tan productivo como un profesional experimentado con tiempos de respuesta más lentos, y el ingeniero experimentado, a su vez, podría ser más rápido que el experto con una ventaja similar en tiempo de respuesta. En todos los casos el número de transacciones por hora crecía enormemente con tiempos de respuesta de subsegundos.

Como los humanos pueden realizar mucho más trabajo por día con mejor tiempo de respuesta, es posible conseguir un beneficio económico para el cliente al bajar el tiempo de respuesta al rango de subsegundos [IBM 1982], ayudando así al arquitecto a decidir cómo pronosticar el equilibrio entre tiempo de respuesta y productividad.

Ejemplos de medidas de rendimiento de E/S. Discos magnéticos

Se necesitan benchmarks para evaluar el rendimiento de E/S, igual que se necesitaban para evaluar el rendimiento de la CPU. Comenzamos con benchmarks para discos magnéticos. Tres aplicaciones tradicionales de discos son:

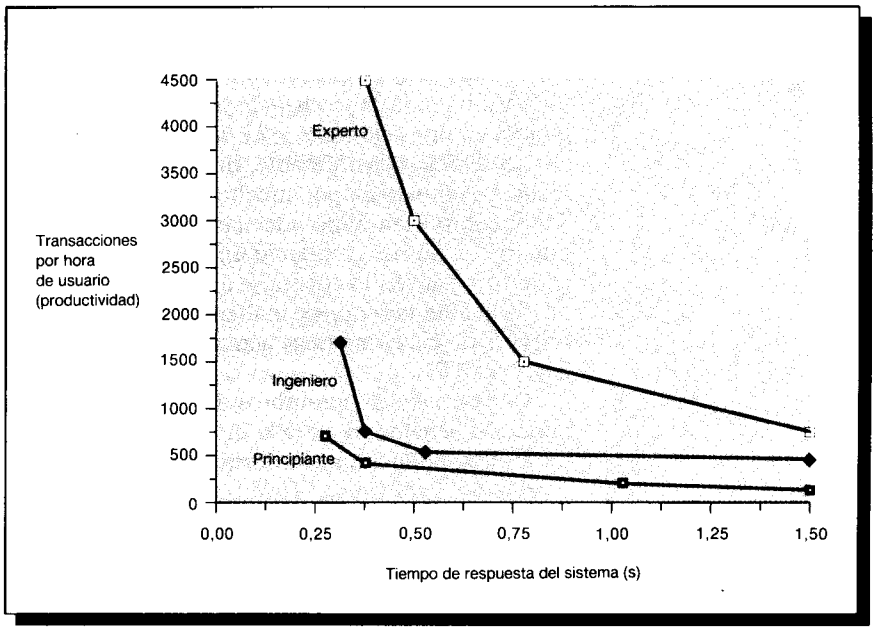


FIGURA 9.9 Transacciones por hora frente a tiempo de respuesta del computador para un ingeniero con experiencia, sin experiencia y el experto haciendo diseño físico en un sistema de gráficos. Las transacciones por hora es una medida de la productividad. (De IBM [1982].)

las de problemas científicos de gran escala, procesamiento de transacciones y sistemas de ficheros.

Benchmarks de E/S de supercomputadores

Las E/S de supercomputadores están dominadas por los accesos a grandes ficheros en discos magnéticos. Por ejemplo, Bucher y Hayes [1980] evaluaron las E/S de supercomputadores utilizando transferencias de ficheros secuenciales de 8 MB. Muchas instalaciones de supercomputadores ejecutan tareas por lotes, cada una de las cuales puede tardar horas. En estas situaciones, las E/S constan de una gran lectura seguida por escrituras que, al cambiar rápidamente el estado de la computación, estropearían el computador. Como consecuencia, las E/S de los computadores, en muchos casos, constan de más salidas que entradas. Algunos modelos de computadores de Cray Research tienen tal limitación de memoria principal que los programadores deben descomponer sus programas en recubrimientos (overlays) y los intercambian con el disco (ver Sección 8.5 del Cap. 8), lo que también provoca grandes transferencias secuenciales. Por tanto, la medida principal de las E/S de los supercomputadores es la productividad de los datos: número de bytes por segundo que se pueden transferir entre la memoria principal y los discos del supercomputador durante grandes transferencias.

Benchmarks de E/S de procesamiento de transacciones

En contraste, el *procesamiento de transacciones* (TP) está relacionado, principalmente, con la *frecuencia de E/S:* el número de accesos al disco por segundo, en oposición a la *velocidad de datos,* medida en bytes de datos por segundo. El TP, generalmente, involucra cambios en un gran cuerpo de información compartida por muchos terminales, garantizando con el sistema de TP el comportamiento adecuado en un fallo. Si, por ejemplo, falla el computador de un banco cuando un cliente retira dinero, el sistema de TP debería garantizar que en la cuenta se anota un débito si el cliente recibió el dinero y que la cuenta permanece inalterada si no se recibió el dinero. Los sistemas de reservas de las compañías aéreas, así como los bancos, son clientes tradicionales del TP.

Dos docenas de miembros de la comunidad de TP conspiraron para proponer un benchmark para la industria y, para evitar la ira de sus departamentos legales, publicaron el informe anónimamente [1985]. Este benchmark, denominado *DebitCredit (DebeHaber),* simula los cajeros de los bancos y tiene como línea inferior el número de transacciones de débito/crédito por segundo (TPS): en 1990, el TPS para las máquinas de altas prestaciones es aproximadamente 300. El DebitCredit realiza la operación de un cliente depositando o retirando dinero. La medida del rendimiento es el TPS máximo, siendo el tiempo de respuesta, del 95 por 100 de las transacciones, menor de un segundo. El DebitCredit calcula el costo por TPS, basándose en el costo de cinco años del hardware y software del sistema de computadores. Las E/S de discos para el DebitCredit son escrituras y lecturas aleatorias de registros de 100 bytes junto a escrituras secuenciales ocasionales.

Dependiendo de lo inteligentemente que se haya diseñado el sistema de procesamiento de transacciones, cada transacción necesita entre dos y diez E/S del disco y entre 5 000 y 20 000 instrucciones de CPU por E/S del disco. La variación depende principalmente de la eficiencia del software de procesamiento de transacciones, aunque en parte depende de la extensión en que se puedan evitar los accesos al disco manteniendo la información en memoria principal. El benchmark requiere que, para incrementar el TPS, también se debe incrementar el número de cajeros y el tamaño del fichero de cuentas. La Figura 9.10 muestra esta relación inusual en la cual más TPS requieren más usuarios. Esto es para asegurar que el benchmark realmente mide las E/S del disco; en cualquier otro caso, una gran memoria principal, dedicada a una cache de la base de datos con un pequeño número de cuentas, podría producir injustamente un TPS muy alto. (¡Otra perspectiva es que el número de cuentas debe crecer, ya que una persona probablemente no utiliza el banco con más frecuencia porque tenga un computador más rápido!)

Benchmarks de E/S del sistema de ficheros

Los sistemas de ficheros, para los cuales se utilizan principalmente discos en los sistemas de tiempo compartido, tienen un patrón de acceso diferente. Ousterhout y cols. [1985] midieron un sistema de ficheros UNIX y encontraron que el 80 por 100 de todos los accesos eran a ficheros de menos de 10 KB y que el 90 por 100 de **todos** los accesos a los ficheros eran secuenciales. La dis-

TPS	Número de ATM	Tamaño de fichero de contabilidad
10	1 000	0,1 GB
100	10 000	1,0 GB
1 000	100 000	10,0 GB
10 000	1 000 000	100,0 GB

FIGURA 9.10 Relación entre TPS, escrutadores y tamaño del fichero de contabilidad. El benchmark *DebitCredit* (DebeHaber) requiere que el sistema computador maneje más escrutadores y mayores archivos de contabilidad antes de que pueda demandar una mayor cantidad de transacciones por segundo. El benckmark supone que incluye gastos de «manipulación de terminales», pero esta métrica a veces se ignora.

tribución por tipo de acceso a los ficheros fue el 67 por 100 de lecturas, el 27 por 100 de escrituras y el 6 por 100 de lectura y escritura. En 1988, Howard y cols. [1988] propusieron un benchmark de sistema de ficheros que se está popularizando. Su artículo describe cinco fases del benchmark, utilizando 70 ficheros cuyo tamaño total es de 200 KB:

MakeDir. Construye un subárbol objeto que es idéntico en estructura al subárbol fuente.

Copy. Copia cada fichero del subárbol fuente en el subárbol objeto.

ScanDir. Recorre recursivamente el subárbol objeto y examina el estado de cada archivo en él. Realmente no lee el contenido de ningún fichero.

ReadAll. Explora cada byte de cada fichero en el subárbol objeto una vez.

Make. Compila y enlaza todos los ficheros del subárbol objeto. [p. 55]

Las medidas del sistema de ficheros de Howard y cols. [1988], como las de Ousterhout y cols. [1985], encontraron que la relación de lecturas a escrituras del disco era aproximadamente de 2:1. Este benchmark refleja esa medida.

9.4 Tipos de dispositivos de E/S

Ahora que hemos cubierto las medidas del rendimiento de E/S, describamos los dispositivos de E/S. Aunque el modelo de computación ha cambiado poco desde 1950, los dispositivos de E/S son numerosos y diversos. Tres características son útiles para organizar esta conglomeración dispar:

- *Comportamiento:* entrada (leer una vez), salida (escribir sólo, no se puede leer) o almacenamiento (puede ser releído y habitualmente reescrito).

- *Compañero:* o bien una persona o una máquina está en el otro extremo del dispositivo de E/S, introduciendo datos en la entrada o leyendo datos en la salida.
- *Frecuencia de datos:* la frecuencia máxima a la cual se pueden transferir datos entre el dispositivo de E/S y la memoria principal o CPU

Utilizando estas características, un teclado es un dispositivo de entrada, utilizado por una persona, con una frecuencia máxima de datos de aproximadamente 10 bytes por segundo. La Figura 9.11 muestra algunos de los dispositivos de E/S conectados a los computadores.

La ventaja de diseñar dispositivos de E/S para personas es que el objetivo de rendimiento está fijado. La Figura 9.12 muestra el rendimiento de E/S de las personas.

Para tener una idea de las frecuencias de datos de cada dispositivo, la Figura 9.13 muestra el ancho de banda máxima relativo de la memoria necesario para soportar cada dispositivo, suponiendo que cada uno de los dis-

Dispositivo	Comportamiento	Compañero	Frecuencia de datos (KB/s)
Teclado	Entrada	Humano	0,01
Ratón	Entrada	Humano	0,02
Entrada de voz	Entrada	Humano	0,02
Scanner	Entrada	Humano	200,00
Salida de voz	Salida	Humano	0,60
Impresora de línea	Salida	Humano	1,00
Impresora láser	Salida	Humano	100,00
Pantalla gráfica	Salida	Humano	30 000,00
(CPU a buffer de encuadre)	Salida	Humano	200,00
Red-terminal	Entrada o salida	Máquina	0,05
Red-LAN	Entrada o salida	Máquina	200,00
Disco óptico	Almacenamiento	Máquina	500,00
Cinta magnética	Almacenamiento	Máquina	2 000,00
Disco magnético	Almacenamiento	Máquina	2 000,00

FIGURA 9.11 Ejemplos de dispositivos de E/S clasificados por comportamiento, compañero y frecuencia de datos. Esta es la frecuencia de datos neta del dispositivo en lugar de la frecuencia que veríamos en una aplicación. Los dispositivos de almacenamiento pueden distinguirse además por si soportan accesos secuenciales (por ejemplo, cintas) o accesos aleatorios (por ejemplo, discos). Observar que las redes pueden actuar como dispositivos de entrada o de salida pero, de forma distinta al almacenamiento, no pueden releer la misma información.

Organo humano	Frecuencia E/S (KB/s)	Latencia E/S (ms)
Oído	8,000-60,000	10
Ojo—lectura de texto	0,030-0,375	10
Ojo—reconocimiento de patrones	125,000	10
Mano—teclear	0,010-0,020	100
Voz	0,003-0,015	100

FIGURA 9.12 Frecuencias máximas de E/S para las personas. La vía de entrada ver patrones es nuestra frecuencia de E/S más alta; de aquí la popularidad de los dispositivos gráficos de salida. Maberly [1966] dice que la velocidad media de lectura es de 28 bytes por segundo y la máxima es de 375 bytes por segundo. La compañía telefónica pone un límite de 170 ms al tiempo desde que un operador pulsa un botón para aceptar una llamada hasta que se establece un camino para la voz. La compañía telefónica transmite voces a 8 KB por segundo. (Ninguno de estos parámetros se espera que cambie, ¡a menos que los esteroides anabólicos se conviertan en un suplemento del desayuno!).

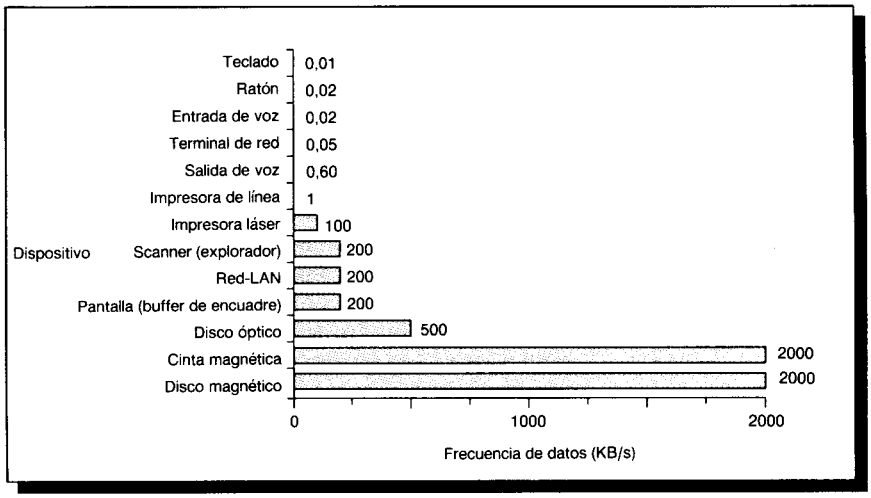


FIGURA 9.13 Dispositivos de E/S clasificados desde la frecuencia más baja de datos a la más alta. La frecuencia de datos para la pantalla de gráficos va desde la CPU al buffer de encuadre, ya que la CPU no está involucrada en la transferencia del buffer de encuadre a la pantalla (ver más abajo subsección de Pantallas Gráficas).

positivos del computador estaba transfiriendo exactamente a su frecuencia máxima.

En lugar de explicar las características de todos los dispositivos de E/S, nos concentraremos en los tres dispositivos con las velocidades de datos más grandes: discos magnéticos, pantallas gráficas y redes de área local. Estos son tam-

bién los dispositivos que tienen mayor impacto en la productividad del usuario. En este capítulo no hablaremos sobre discos flexibles, pero sí de los originales discos «duros». Estos discos magnéticos son los que IBM denominó *DASD: Dispositivos de Almacenamiento de Acceso Directo (Direct-Access Storage Devices).*

Discos magnéticos

Creo que Valle del Silicio es una mala denominación. Si mirásemos retrospectivamente los dólares generados por productos en la última década, ha habido más ingresos provenientes de los discos magnéticos que del silicio. Se debería renombrar el lugar como Valle del Oxido de Hierro.

Al Hoagland, uno de los pioneros de los discos magnéticos (1982)

A pesar de los repetidos ataques de las nuevas tecnologías, los discos magnéticos han dominado las memorias secundarias desde 1965. Los discos magnéticos juegan dos papeles en los sistemas de computadores:

- Memorias no volátiles a largo plazo para ficheros, aun cuando no se estén ejecutando los programas
- Un nivel de la jerarquía de memoria por debajo de la memoria principal utilizado como memoria virtual durante la ejecución del programa (ver Sección 8.5 en el Cap. 8)

Como descripciones sobre discos magnéticos se pueden encontrar en innumerables libros; solamente citaremos las características clave en los términos ilustrados en la Figura 9.14. Un disco magnético consta de una colección de platos (de 1 a 20), que giran sobre un eje aproximadamente a 3 600 revoluciones por minuto (RPM). Estos platos son discos metálicos cubiertos por ambas caras con material de grabación magnético. Los diámetros de los discos varían en un factor de cinco, desde 14 a 2,5 pulgadas. Tradicionalmente, los discos mayores tienen mayor rendimiento y los menores tienen el mínimo coste por unidad de disco.

Cada superficie del disco está dividida en círculos concéntricos, denominados *pistas.* Normalmente, hay de 500 a 2 000 pistas por superficie. Cada pista a su vez está dividida en *sectores* que contienen la información. Cada pista puede tener 32 sectores. El sector es la unidad más pequeña que puede ser leída o escrita. La secuencia grabada en el medio magnético es un número de sector, un espacio vacío, la información para ese sector, incluyendo código corrector de errores, un espacio vacío, el número de sector del siguiente sector, y así sucesivamente. Tradicionalmente, todas las pistas tienen el mismo número de sectores; las pistas más externas, que son más largas, graban la información con una densidad menor que las pistas más internas. La grabación de más sectores en las pistas externas que en las internas, denominada *densidad constante de bits,* se está ampliando con el advenimiento de estándares de interfaces inteligentes tal como SCSI (ver Sección 9.5). Los discos de los gran-

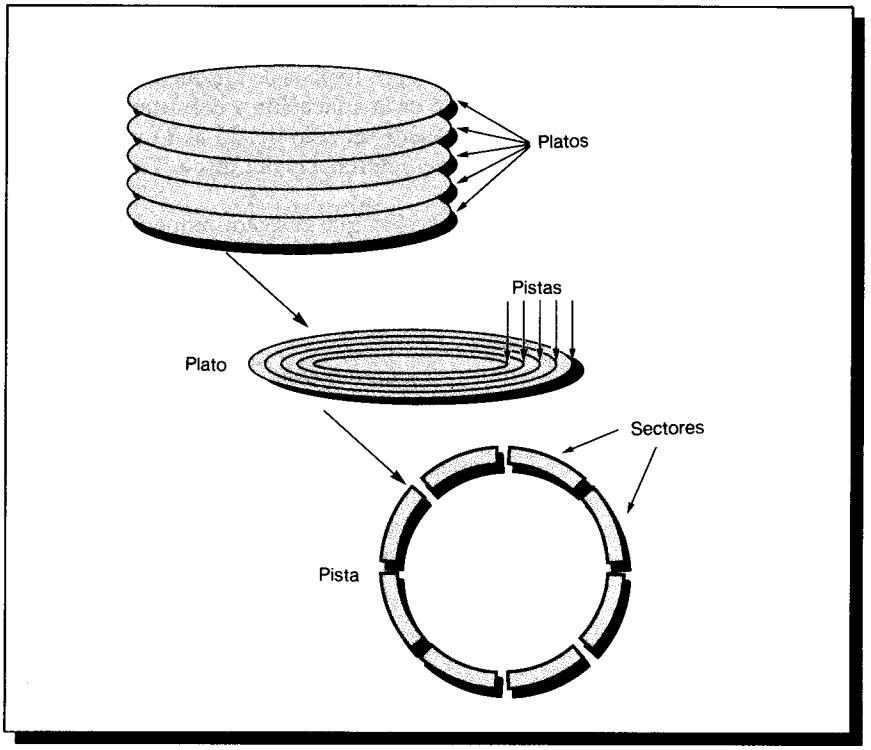


FIGURA 9.14 Los discos están organizados en platos, pistas y sectores. Ambas caras de un plato están revestidas para que la información pueda almacenarse en ambas superficies.

des ordenadores de IBM permiten a los usuarios seleccionar el tamaño de los sectores, aunque casi todos los demás sistemas fijan el tamaño del sector.

Para leer y escribir información en un sector, un *brazo* móvil que contiene una *cabeza de lectura/escritura* se coloca sobre cada superficie. Los bits son grabados utilizando un código de longitud limitada, que mejora la densidad de grabación del medio magnético. Los brazos para cada superficie se conectan juntos y se mueven en conjunción, para que cada brazo esté sobre la misma pista de cada superficie. El término *cilindro* se utiliza para referenciar todas las pistas bajo los brazos, en un punto dado, en todas las superficies.

Para leer o escribir en un sector, el controlador del disco envía una orden para mover el brazo sobre la pista adecuada. Esta operación se denomina *búsqueda (seek),* y el tiempo para desplazar el brazo a la pista deseada se denomina *tiempo de búsqueda (seek time).* El tiempo medio de búsqueda es el sujeto de considerables malos entendidos. Los fabricantes de discos informan, en los manuales, del tiempo mínimo de búsqueda, del tiempo máximo de búsqueda y del tiempo medio de búsqueda. Los dos primeros son fáciles de medir, pero el promedio está abierto a una amplia interpretación. La indus-

tria decidió calcular el tiempo medio de búsqueda como la suma de los tiempos de todas las posibles búsquedas dividido por el número de búsquedas posibles. Los tiempos medios de búsqueda se anuncian entre los 12 y 20 ms, pero, dependiendo de la aplicación y del sistema operativo, el tiempo medio de búsqueda real puede ser sólo del 25 al 33 por 100 del anunciado, debido a la localidad de las referencias del disco. La Sección 9.10 tiene un ejemplo detallado.

El tiempo para que el sector requerido gire bajo la cabeza es la *latencia de rotación* o *retardo rotacional.* La mayoría de los discos giran a 3 600 RPM, y una latencia media para la información deseada está a medio camino en torno al disco; el tiempo medio de rotación para muchos discos es, por tanto

$$\text{Tiempo medio de votación} = \frac{0{,}5}{3\,600\ \text{RPM}} = 0{,}0083\ \text{s} = 8{,}3\ \text{ms}$$

El siguiente componente de un acceso a disco, *tiempo de transferencia,* es el tiempo en transferir un bloque de bits, normalmente un sector, bajo la cabeza de lectura-escritura. Este es función del tamaño del bloque, velocidad de rotación, densidad de grabación de una pista y velocidad de los componentes electrónicos que conectan el disco al computador. Las frecuencias de transferencia en 1990, normalmente, son de 1 a 4 MB por segundo.

Además de la unidad de disco, habitualmente hay también un dispositivo denominado *controlador de disco.* Entre el controlador de disco y memoria principal hay una jerarquía de controladores y de caminos de datos, cuya complejidad varía con el coste del computador (ver Sección 9.9). Como el tiempo de transferencia es, con frecuencia, una parte pequeña de un acceso completo a disco, el controlador en sistemas de rendimientos más altos desconecta los caminos de datos de los discos mientras están buscando para que otros discos puedan transferir sus datos a memoria.

El componente final del tiempo de acceso del disco es el *tiempo del controlador,* que es el coste adicional que el controlador impone al realizar un acceso de E/S. Cuando nos referimos al rendimiento de un disco en un sistema computador, el tiempo que hay que esperar a que un disco quede libre *(retardo de cola),* se suma a este tiempo.

Ejemplo

¿Cuál es el tiempo medio para leer o escribir un sector de 512 bytes para un disco típico actual? El tiempo medio de búsqueda anunciado es de 20 ms; la frecuencia de transferencia es de 1 MB/s, y el gasto del controlador es 2 ms. Suponer que el disco está desocupado para que no haya retardo de cola.

Respuesta

El acceso medio al disco es igual al promedio del tiempo de búsqueda + retardo rotacional medio + tiempo de transferencia + gasto del controlador. Utilizando lo calculado, la respuesta para el tiempo medio de búsqueda es:

$$20\ \text{ms} + 8{,}3\ \text{ms} + \frac{0{,}5\ \text{KB}}{1{,}0\ \text{MB/s}} + 2\ \text{ms} = 20 + 8{,}3 + 0{,}5 + 2 = 30{,}8\ \text{ms}$$

Suponiendo que el tiempo medio de búsqueda medido es el 25 por 100 del número calculado, la respuesta es

$$5 \text{ ms} + 8{,}3 \text{ ms} + 0{,}5 \text{ ms} + 2 \text{ ms} = 15{,}8 \text{ ms}$$

La Figura 9.15 muestra características de discos magnéticos de cuatro fabricantes. Las unidades de gran diámetro tienen muchos más megabytes para amortizar el coste de la electrónica, ya que la sensatez tradicional indica que tienen el coste más bajo por megabyte. Pero esta ventaja se desplaza para las pequeñas unidades por el mayor volumen de ventas, que baja los costes de fabricación: los precios OEM de 1990 son de 2 a 3 dólares por megabyte, casi con completa independencia del tamaño. Las pequeñas unidades también tienen ventaja en potencia y volumen. El precio de un megabyte de memoria de disco en 1990 es de 10 a 30 veces más barato que el precio de un megabyte de DRAM en un sistema.

Características	IBM 3380	Fujitsu M2361A	Imprimis Wren IV	Conner CP3100
Diámetro del disco (pulgadas)	14	10,5	5,25	3,5
Capacidad de datos formateados (MB)	7 500	600	344	100
MTTF (horas)	52 000	20 000	40 000	30 000
Número de brazos	4	1	1	1
Máxima E/S/segundo/brazo	50	40	35	30
Típica E/S/segundo/brazo	30	24	28	20
Máximo E/S/segundo/caja	200	40	35	30
Típica E/S/segundo/caja	120	24	28	20
Transfer rate (MB/s)	3	2,5	1,5	1
Potencia/caja (W)	1 650	640	35	10
MB/W	1,1	0,9	9,8	10,0
Volumen (cu.ft.)	24	3,4	0,1	0,03
MB/cu. ft.	310	180	3 440	3 330

FIGURA 9.15 Características de los discos magnéticos de cuatro fabricantes. Comparación del modelo de disco AK4 del IBM 3380 para computadores grandes, disco «Super Eagle» M2361A de Fujitsu para minicomputadores, disco Imprimis Wren IV para estaciones de trabajo y disco CP3100 de Conner Peripherals para computadores personales. Máxima E/S/segundo significa máximo número de búsquedas medias y rotaciones medias para acceder a un único sector. (Tabla de Katz, Patterson y Gibson [1990].)

El futuro de los discos magnéticos

La industria de los discos se ha concentrado en mejorar la capacidad de los discos. Las mejoras en capacidad se expresan como *densidad de área,* medida en bits por pulgada cuadrada.

$$\text{Densidad de área} = \frac{\text{Pistas}}{\text{Pulgada}} \text{ en una superficie de disco} \cdot \frac{\text{Bits}}{\text{Pulgada}} \text{ en una pista}$$

La densidad de área se puede predecir de acuerdo con la fórmula de la *densidad máxima de área* (MAD):

$$\text{MAD} = 10^{(\text{año}-1971)/10} \text{ millones de bits por pulgada cuadrada}$$

Por tanto, la densidad de memoria mejora en un factor de 10 cada década, doblando la densidad cada tres años.

El coste por megabyte ha caído consistentemente del 20 al 25 por 100 por año, jugando las unidades más pequeñas el papel más importante en esta mejora. Como es más fácil hacer girar la masa más pequeña, los discos de diámetro más pequeño ahorran tanta potencia como volumen. Las unidades más pequeñas también tienen menos ciclindros ya que las distancias de búsqueda son más cortas. En 1990, unidades de 5,25 o 3,5 pulgadas son probablemente la tecnología líder, mientras que en el futuro se podrán ver aún unidades más pequeñas. Podemos esperar ahorros significativos en volumen y potencia, pero poco en velocidad. El incremento de densidad (bits por pulgada en una pista) ha mejorado los tiempos de transferencia y ha mejorado poco la velocidad de búsqueda. Las velocidades de rotación han permanecido estacionarias a 3 600 RPM durante una década, pero algunos fabricantes planean ir a 5 400 RPM a principios de los años noventa.

Como mencionamos antes, los discos magnéticos han sido desafiados muchas veces para la supremacía de la memoria secundaria. Una razón ha sido la tabulada diferencia *del tiempo de Acceso (Access Time Gap),* como muestra la Figura 9.16. Muchos científicos han tratado de inventar una tecnología que llene esa diferencia. Examinemos algunos de los recientes intentos.

Utilización de DRAM como discos

Un reto actual a los discos para el dominio de la memoria secundaria son los *discos de estado sólido* (SSD), construidos con DRAM, con una batería para hacer el sistema no volátil; y la *memoria expandida* (ES), una gran memoria que sólo permite transferencias de bloques a o desde memoria principal. La ES actúa como una cache controlada por software (la CPU se detiene durante la transferencia de bloques) mientras que los SSD involucran al sistema operativo de la misma forma que una transferencia de discos magnéticos. Las ventajas de los SSD y la ES son tiempos de búsqueda triviales, frecuencia de transferencias potenciales mayores y, posiblemente, mayor fiabilidad. De forma

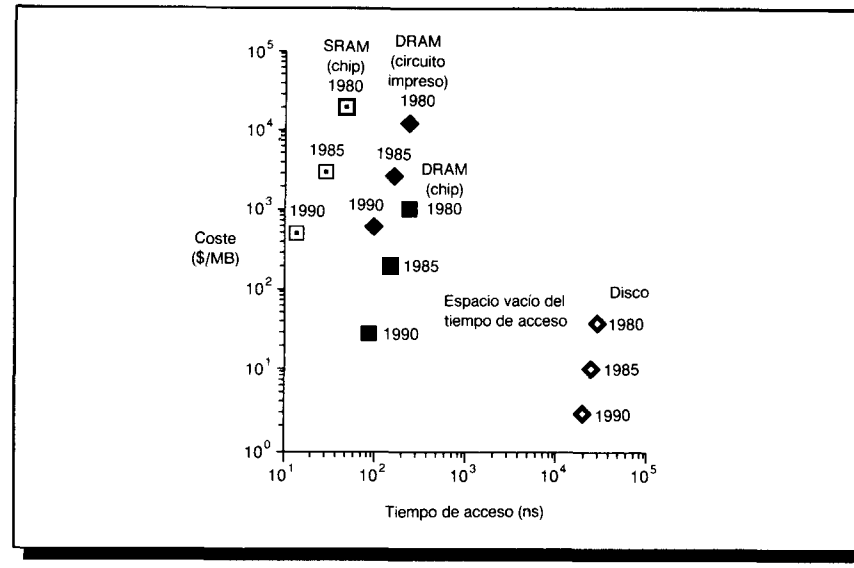


FIGURA 9.16 Coste frente a tiempo de acceso para SRAM, DRAM y disco magnético en 1980, 1985 y 1990. (Observar la diferencia en coste entre un chip DRAM y chips DRAM empaquetados en un circuito impreso y listo para conectar a un computador.) El espacio vacío de dos órdenes de magnitud en coste y tiempos de accesos entre la memoria de semiconductores y los discos magnéticos de rotación ha inspirado un sinnúmero de tecnologías competentes para tratar de rellenarlo. Hasta ahora, tales intentos han quedado obsoletos por mejoras en los discos magnéticos, DRAM o ambas cosas.

distinta a las grandes memorias principales, los SSD y las ES son autónomas: requieren órdenes especiales para acceder a sus almacenamientos y, por tanto, están «seguras» de algunos errores software que escriban sobre memoria principal. La naturaleza del acceso a los bloques de SSD y ES permite que la corrección de errores se extienda sobre más palabras, lo cual significa menor coste o mayor recuperación de errores. Por ejemplo, la ES de IBM utiliza el mayor grado de recuperación de errores para permitir que se construyan DRAM menos fiables (y menos caras) sin sacrificar la disponibilidad del producto. Los SSD, de forma distinta a la memoria principal y a la ES, pueden ser compartidos por múltiples CPU porque funcionan como unidades separadas. Colocar DRAM en un dispositivo de E/S en lugar de en memoria es también una forma de llegar a los límites del espacio de direcciones de los computadores actuales de 32 bits. La desventaja de los SSD y la ES es el coste, que es como mínimo diez veces por megabyte el coste de los discos magnéticos.

Discos ópticos

Otro contrincante para los discos magnéticos son los *discos ópticos compactos* o CD. El *CD/ROM* es removible y barato de fabricar, pero es un medio de

sólo lectura. El CD/escribible, más moderno, también es removible, pero tiene un alto coste por megabyte y bajo rendimiento. Una mala percepción común sobre los *discos ópticos de una sola escritura* es que una vez que han sido escritos, la información no puede ser destruida; en efecto, escribir una vez significa una escritura fiable y, a partir de ahí, lo que existe es una función OR bit a bit «difusa» del dato anterior y el nuevo.

Hasta ahora, los contrincantes de los discos magnéticos nunca habían tenido un producto en el mercado en el instante correcto. En el momento que aparecen nuevos productos, los discos han hecho avances como predice la fórmula MAD, y los costes han caído en consecuencia. Sin embargo, los discos ópticos, pueden tener el potencial de competir con las nuevas tecnologías de cintas para almacenamiento de archivos.

Arrays de discos

Otro futuro candidato para optimizar memoria no es una nueva tecnología, sino una nueva organización de la memoria de discos —arrays de discos pequeños y baratos—. El argumento para los arrays es que, como el precio por megabyte, es independiente del tamaño del disco, la productividad potencial se puede incrementar teniendo muchas unidades de discos y, por consiguiente, muchos brazos de discos. Dispersar datos sobre múltiples discos fuerza automáticamente a acceder a varios discos. (Aunque los arrays mejoran la productividad, la latencia no se mejora necesariamente.) El inconveniente de los arrays es que con más dispositivos la fiabilidad disminuye: N dispositivos generalmente tienen $1/N$ la fiabilidad de un solo dispositivo.

Fiabilidad y disponibilidad

Esto nos conduce a dos términos que con frecuencia se confunden: fiabilidad y disponibilidad. El término fiabilidad se usa comúnmente de manera incorrecta para significar disponibilidad; si algo falla, pero el usuario puede todavía usar el sistema, parece como si el sistema «funcionase» todavía y, por consiguiente, parece más fiable. Aquí está la distinción adecuada:

Fiabilidad (reliability): ¿está algo roto?

Disponibilidad (availability): ¿está el sistema todavía disponible al usuario?

Añadir hardware puede, por tanto, mejorar la disponibilidad (por ejemplo, ECC en memoria), pero no puede mejorar la fiabilidad (la DRAM está todavía rota). La fiabilidad se puede incrementar solamente mejorando las condiciones del entorno, construyendo componentes más fiables, o construyendo con menos componentes. Otro término, *integridad de datos,* se refiere siempre a la pérdida de información cuando se presenta un fallo; esto es muy importante en algunas aplicaciones.

Así, aunque un array de discos no pueda ser nunca más fiable que un número pequeño de discos mayores cuando cada disco tenga la misma frecuen-

cia de fallos, la disponibilidad se puede mejorar añadiendo discos redundantes. Esto es, si un simple disco falla, la pérdida de información se puede reconstruir a partir de la información redundante. El único peligro está en que ocurra otro fallo del disco entre el instante que falla el disco y el instante que se sustituye (denominado *tiempo medio de reparación* o MTTR). Como el *tiempo medio de fallo* (MTTF) de los discos es de tres a cinco años y el MTTR se mide en horas, la redundancia puede hacer la disponibilidad de 100 discos mucho mayor que la de un solo disco.

Como los fallos de los discos son autoidentificativos, la información se puede reconstruir a partir de la paridad: los discos buenos más el disco de paridad se pueden utilizar para calcular la información que estaba en el disco que falló. Por consiguiente, el coste de mayor disponibilidad es $1/N$, donde N es el número de discos protegidos por paridad. Igual que en las caches la ubicación asociativa de correspondencia directa se puede considerar como un caso especial de la ubicación asociativa por conjuntos (ver Sección 8.4), el *reflejo* o *sombreado (mirroring* o *shadowing)* de los discos se puede considerar el caso especial de un disco de datos y un disco de paridad ($N = 1$). La paridad se puede lograr duplicando los datos; por ello, los discos reflejados tienen la ventaja de simplificar el cálculo de la paridad. La duplicación de datos también significa que el controlador puede mejorar el rendimiento de lectura al leer del disco del que tiene menor distancia de búsqueda, aunque esta optimización está en el coste del rendimiento de escritura porque los brazos del par de discos no están siempre sobre la misma pista. Por supuesto, la redundancia de $N = 1$ tiene el mayor coste para incrementar la disponibilidad del disco.

La mayor productividad, medida bien como megabytes por segundo o como E/S por segundo, y la posibilidad de recuperarse de los fallos hacen atractivos a los arrays de disco. Cuando se combinan con las ventajas de menor volumen y más baja potencia de las unidades de pequeño diámetro, los arrays redundantes de unidades pequeñas o baratas pueden jugar un gran papel en los futuros sistemas de disco. El inconveniente actual es la complejidad añadida, de un controlador, para los arrays de discos.

Pantallas gráficas

Utilizando pantallas de computadores he hecho aterrizar un aeroplano en la cubierta de un portaviones en movimiento, he observado el impacto de una partícula nuclear en un pozo de potencial; he volado en un meteorito a una velocidad próxima a la de la luz y he visto a un computador revelar su funcionamiento más interno.

Ivan Sutherland (el «padre» de los gráficos de computadores), tomado de «Computer Software for Graphics», *Scientific American* (1984)

Aunque los discos magnéticos pueden dominar la productividad y coste de los dispositivos de E/S, el dispositivo de E/S más fascinante es la pantalla gráfica. Basada en la tecnología de la televisión, una *pantalla de tubo de rayos catódicos* (CRT) explora cada vez una línea de la imagen de 30 a 60 veces por segundo. A esta *frecuencia de refresco,* el ojo humano no observa ningún

«parpadeo» en la pantalla. La imagen está compuesta por una matriz de elementos o *pixels,* que se puede representar como una matriz de bits, denominada *mapa de bits.* Dependiendo del tamaño y resolución de la pantalla, la matriz de la pantalla consta de 340 · 512 a 1 560 · 1 280 pixels. Para las pantallas en blanco y negro, a menudo, el 0 es el negro y el 1 es el blanco. Para pantallas que soportan alrededor de 100 formas diferentes de blanco y negro, a veces denominadas pantallas de *escala de grises,* se necesitan 8 bits por pixel. Una pantalla de color puede usar 8 bits por cada uno de los tres colores principales (rojo, azul y verde), es decir, 24 bits por pixel.

El soporte hardware para los gráficos consta principalmente de un *buffer de refresco de exploración* o *buffer de encuadre,* para almacenar el mapa de los bits. La imagen que está representada en la pantalla se almacena en el buffer de encuadre y el patrón de bits por pixel es leído en la pantalla gráfica a la frecuencia de refresco. La Figura 9.17 muestra un buffer de encuadre con cuatro bits por pixel y la Figura 9.18 muestra cómo se conecta el buffer al bus.

El objetivo del mapa de bits es representar con exactitud lo que está en la pantalla. Cuando el computador cambia de una imagen a otra, la pantalla puede parecer «borrosa» durante el cambio. Hay dos formas de tratar esto:

- Cambiar el buffer de encuadre sólo durante el «intervalo de blanqueo vertical». Este es el tiempo que el cañón de la pantalla CRT necesita para volver a la esquina superior izquierda antes de comenzar a pintar los pixels de la siguiente imagen. Esto necesita de 1 a 2 ms cada 16 ms a la frecuencia de refresco de 60 Hz cada vez que se pinta la pantalla.
- Si el intervalo de blanqueo vertical no es suficientemente grande, el buffer de encuadre puede ser un buffer doble, para que uno sea leído mientras se escribe en el otro. De esta forma, las imágenes en secuencia (como en una

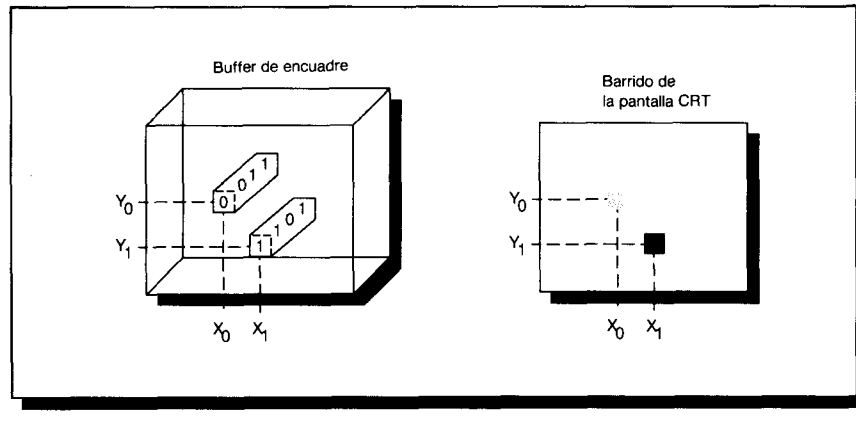


FIGURA 9.17 Cada coordenada del buffer de encuadre de la izquierda determina la sombra de la coordenada correspondiente para la pantalla CRT de la derecha. El pixel (x_0,y_0) contiene el patrón de bits 0011, que está un poco más sombreado de gris en la pantalla que el patrón de bits 1101 en el pixel (x_1,y_1).

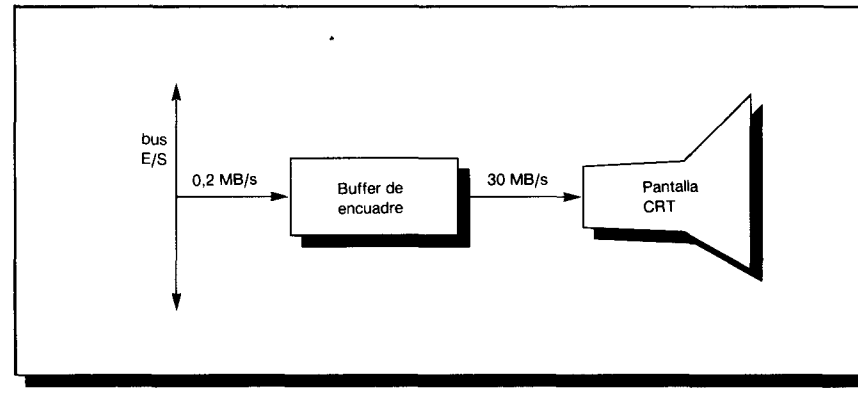


FIGURA 9.18 El buffer de encuadre está conectado al bus de E/S y a la pantalla. Debido a la alta frecuencia de datos del buffer a la pantalla, el buffer de encuadre está, frecuentemente, duplicado.

animación) se dibujan en buffers de encuadre alternos. El doble buffer, por supuesto, duplica el coste de la memoria del buffer de encuadre.

Desde el punto de vista de la CPU, las gráficas son lógicamente sólo una salida. Pero el buffer de encuadre es capaz de ser leído y escrito, permitiendo que se realicen las operaciones directamente en las imágenes de pantalla. Estas operaciones son denominadas *blts de bits,* por transferencia de bloques de bits *(bit block transfer).* Las blts de bits se utilizan comúnmente para operaciones como desplazar una ventana o cambiar la forma del cursor. Un debate actual en la arquitectura de gráficos es si la lectura del buffer de encuadre se limita al sistema operativo o también se hace extensiva a los programas de usuario.

Coste de las gráficas del computador

El monitor CRT está basado en la tecnología de televisión y es sensible a la demanda de los consumidores. Hoy día los precios varían desde 100 dólares para un monitor en blanco y negro a 15 000 dólares para un gran monitor de color de estudio, sin incluir memoria. La cantidad de memoria de un buffer de encuadre depende directamente del tamaño de la pantalla y de los bits por pixel:

$$340 \cdot 512 \cdot 1 \text{ bits} = 21{,}5 \text{ KB}$$
$$1\,280 \cdot 1\,024 \cdot 24 \text{ bits} = 3\,840 \text{ KB}$$

(Dicho sea de paso, la dimensión última es el tamaño propuesto para la televisión de alta definición.) Observar que el coste de memoria se duplica si se utiliza doble buffer.

Para reducir los costes de un buffer de encuadre de color, muchos sistemas utilizan una representación de dos niveles que se aprovecha del hecho de que pocos cuadros necesitan la paleta completa de posibles colores (ver Fig. 9.19).

El nivel intermedio contiene la anchura completa de colores de, digamos, 24 bits y una gran colección de los posibles colores que puede aparecer en la pantalla —256 colores diferentes, por ejemplo—. Aunque esta colección es grande, es todavía mucho menor que 2^{24}. Esta tabla intermedia se ha denominado de distintas formas: *mapa de colores, tabla de colores* o *tabla «look-up» de vídeo.* Cada pixel sólo necesita tener suficientes bits para indicar un color en el mapa de colores. A título de ejemplo, la Figura 9.19 utiliza un mapa de colores de 4 palabras, lo que significa que el buffer de encuadre solamente necesita 2 bits por pixel. El ahorro de una pantalla de color de tamaño completo con un mapa de 256 colores es

$$1\,280 \cdot 1\,024 \cdot 24 - (1\,280 \cdot 1\,024 \cdot 8 + 256 \cdot 24) =$$

$$= 3\,840 \text{ KB} - (1\,280 \text{ KB} + 0{,}75 \text{ KB}) \approx 2\,560 \text{ KB}$$

Esto significa una reducción de un tercio del tamaño de memoria. En 1990 un mapa de colores de 256 por 24 bits y una interfaz analógica para un CRT de color caben en un solo chip.

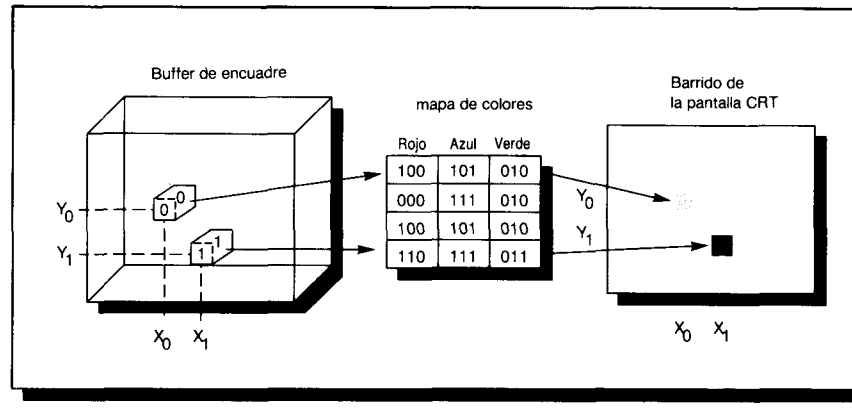


FIGURA 9.19 Ejemplo de un mapa de colores para reducir el coste del buffer de encuadre. Suponer que sólo se necesitan nueve bits por color. En lugar de almacenar los nueve bits completos por pixel en el buffer de encuadre, se almacenan bits suficientes por pixel para indexar la tabla que contiene los únicos colores de un cuadro. Sólo el mapa de colores tiene los nueve bits para los colores de la pantalla. Cuadros de colores fotográficos casi pueden producirse con aproximadamente 125 colores utilizando las tonalidades correctas del espectro de colores; ¡pero se necesitan como mínimo 24 bits para obtener las tonalidades correctas! El mapa de colores lo carga el programa de aplicación, ofreciendo a cada cuadro su propia paleta de colores para escoger.

Demandas de rendimiento de las pantallas gráficas

El rendimiento de los gráficos está determinado por la frecuencia con que una aplicación necesita nuevas imágenes y por la calidad de las mismas. La cantidad de información transferida de memoria al buffer de encuadre depende de la complejidad de la imagen, necesitando casi cuatro megabytes en una pantalla de color. La frecuencia de transferencia depende de la velocidad con la cual se debe cambiar la imagen así como de la cantidad de información. La animación requiere como mínimo 15 cambios por segundo para que un movimiento aparezca suavemente en una pantalla. Para gráficos interactivos, el tiempo para actualizar el buffer de encuadre mide la efectividad de la aplicación; para que las personas se sientan cómodas, el tiempo total de reacción debe ser menor de un segundo (ver Fig. 9.9). Con un sistema de dibujo, la parte de la pantalla que está trabajando debe cambiar casi inmediatamente, ya que la percepción visual humana es del orden de 0,02 segundos. La Figura 9.20 muestra algunos ejemplos de tareas gráficas y sus requerimientos de rendimiento. Observar que el buffer de encuadre debe tener suficiente anchura de banda para refrescar la pantalla y permitir que la CPU cambie la imagen que se está refrescando.

La alta frecuencia de datos —y el gran mercado de las pantallas gráficas— ha hecho al chip DRAM de doble puerto muy popular. Este chip tiene un puerto serie de E/S y un registro de desplazamiento interno, que está conectado a la pantalla de una aplicación de gráficos, además del tradicional puerto de datos direccionado aleatoriamente. El chip es tan utilizado en los buffers de encuadre que se denomina *DRAM de vídeo.*

Tareas gráficas	Requerimientos de ancho de banda
Editor de texto—Enrollar texto en una ventana significa desplazar todos los bits de la mitad del buffer de encuadre aproximadamente 10 veces por segundo	0,8 MB/s
Diseño VLSI—Desplazar una parte del diseño significa desplazar todos los bits de la mitad de un buffer de encuadre de color en menos de 0,1 segundos	6,3 MB/s
Televisión comercial—Mostrar imágenes de calidad en movimiento significa cambiar 24 veces por segundo	90,0 MB/s
Visualización de datos científicos— Aproximadamente igual que la televisión comercial	90,0 MB/s

FIGURA 9.20 Tareas gráficas y sus requerimientos de rendimiento. El diseño VLSI utiliza ocho bits de color mientras que la televisión comercial y visualización utilizan 24 bits. El ancho de banda se mide en el buffer de encuadre.

Futuras direcciones en las pantallas gráficas

Es seguro predecir que la gente querrá mejores imágenes en el futuro. Querrán, por ejemplo, más líneas en una pantalla y más bits por pulgada en una línea para obtener imágenes más vivas, más bits por color para obtener imágenes más coloreadas y más anchura de banda para permitir animación.

Para simplificar la visualización de imágenes de tres dimensiones, puede añadirse una dimensión z por pixel a las coordenadas x e y. Esta indica dónde se localiza el pixel, para un observador, a lo largo del eje z (por ejemplo, en el CRT). Una imagen 3D comienza con z en la posición más lejana posible desde el observador y el color inicializado con el color de fondo. Para obtener una perspectiva 3D adecuada, la coordenada z almacenada con el pixel del buffer de encuadre se comprueba antes de colocar un color en un pixel. Si el nuevo color está más próximo, se sustituye el color antiguo y la coordenada z se actualiza; si está más alejado se descarta el nuevo color. Este esquema se denomina aproximación del *buffer z* para la *eliminación de superficies ocultas.* Añade como mínimo 8 bits por pixel, más el coste del rendimiento de leer y comparar antes de escribir un pixel. La serie 4D de Silicon Graphics de las estaciones de trabajo gráficas utiliza 16 bits para la dimensión z de sus pixels, significando que a los objetos se les asigna un número de 16 bits para mostrar lo próximos que están al observador.

El número creciente de bits por chip DRAM reduce el número de chips necesarios en el buffer de encuadre, así como el número de chips que pueden transferir simultáneamente bits a la pantalla. Esta es la razón por la cual los DRAM de vídeo son tan populares. Cuando se incremente la capacidad, los puertos serie de las DRAM de vídeo tendrán que ser más rápidos y anchos para atender las demandas de los futuros sistemas de gráficos.

Redes

Hay un antiguo refrán sobre redes: Los problemas de ancho de banda se pueden curar con dinero. Los problemas de latencia son más difíciles porque la velocidad de la luz es fija —y no se puede sobornar a Dios.

David Clark, M.I.T.

Las redes son el hueso duro de los actuales sistemas de computadores; una nueva máquina sin una interfaz opcional de red sería ridícula. Al conectar los computadores electrónicamente, las redes de computadores tienen las siguientes ventajas:

- *Comunicación.* La información se intercambia entre computadores a altas velocidades.
- *Compartición de recursos.* En lugar de que cada máquina tenga sus propios dispositivos de E/S, los dispositivos pueden ser compartidos por los computadores de la red.

Distancia	0,01 a 10 000 kilómetros
Velocidad	0,001 MB/s a 100 MB/s
Topología	Bus, anillo, estrella, árbol
Líneas compartidas	Ninguna (punto-a-punto) o compartida (multi) .

FIGURA 9.21 Rango de características de las redes.

- *Accesos no locales.* Al conectar los dispositivos de E/S sobre grandes distancias, los usuarios no necesitan estar cerca del computador que estén utilizando.

La Figura 9.21 muestra las características de las redes. Estas características se ilustran a continuación con tres ejemplos.

El RS232 estándar proporciona un *terminal de red* de 0,3 a 19,2 Kbits por segundo. Un computador central se conecta a muchos terminales con cables dedicados, lentos pero baratos. Estas conexiones punto a punto forman una estrella desde el computador central, donde la distancia de cada terminal al computador varía de 10 a 100 metros.

La *red de área local* o LAN, es lo que comúnmente se entiende hoy día cuando la gente menciona una red y *Ethernet* es lo que la mayoría de la gente quiere decir cuando mencionan una LAN. (En efecto, Ethernet ha llegado a ser un término tan común que se utiliza, con frecuencia, como término genérico para LAN.) La Ethernet es esencialmente un bus de 10 000 Kbits por segundo que no tiene control central. Los mensajes o *paquetes* se envían sobre la Ethernet en bloques que varían desde 128 bytes a 1 530 bytes y necesitan 0,1 milisegundos y 1,5 milisegundos, respectivamente, para su envío. Como no hay control central, todos los nodos «escuchan» para detectar si hay un mensaje para ese nodo. Sin un árbitro central para decidir quién obtiene el bus, un computador escucha primero para asegurarse que no envía un mensaje mientras en la red está otro mensaje. Si la red está desocupada el nodo intenta enviarlo. Por supuesto, algún otro nodo puede decidir enviar en el mismo instante un mensaje. Afortunadamente, el computador puede detectar cualquier colisión resultante al escuchar lo que se envía. (Los mensajes mezclados suenan como basura.) Para evitar colisiones de cabeceras repetidas, cada nodo cuyo paquete fue destrozado se inhibe un tiempo aleatorio antes de reenviarlo. Si Ethernet no tiene una alta utilización, esta aproximación sencilla para la arbitración funciona bien. Muchas LAN, cuando están sobrecargadas debido a pobre capacidad de planificación, pueden degradar completamente el tiempo de respuesta y la productividad con una mayor utilización.

El éxito de las LAN ha conducido a muchas de ellas a un solo emplazamiento. Conectar computadores para separar Ethernets se hace necesario en un cierto punto, ya que hay un límite en el número de nodos que pueden estar activos en un bus si se quieren lograr velocidades efectivas de comunicación; un límite es de 1 024 nodos por Ethernet. También hay una limitación física a la distancia de una Ethernet, habitualmente 1 km. Para permitir que las Ethernets trabajen juntas, se han creado dos tipos de dispositivos:

- Un *puente (bridge)* conecta dos Ethernets. Hay todavía dos buses independientes que pueden enviar mensajes simultáneamente, pero el puente actúa como un filtro, permitiendo que sólo crucen el puente aquellos mensajes de los nodos de un bus a los nodos del otro bus.
- Una *pasarela (gateway)* normalmente conecta varias Ethernets. Recibe un mensaje, examina la dirección de destino en una tabla y, después, devuelve el mensaje a la red apropiada para el nodo adecuado. Esta *tabla de rutas (routing table)* puede ser cambiada durante la ejecución para reflejar el estado de las redes. Algunos utilizan el término *encaminador (router)* en lugar de pasarela ya que está más próximo a la función realizada.

Cuando las Ethernets se conectan juntas con pasarelas forman una *Internet.*

Las *redes de larga distancia* cubren distancias de 10 a 10 000 km. La primera y más famosa red de larga distancia fue la ARPANET (denominada así después de su fundación por la Advanced Research Projects Agency del gobierno de Estados Unidos). Transfería 50 Kbits por segundo y utilizaba líneas dedicadas punto a punto arrendadas a las compañías telefónicas. El computador hablaba a un *procesador de interfaz de mensajes* (IMP), que comunicaba sobre las líneas de teléfono. El IMP tomaba información y la descomponía en paquetes de 1 Kbit. En cada salto se almacenaba el paquete y después se enviaba al IMP adecuado de acuerdo con la dirección del paquete. El IMP de destino reensamblaba los paquetes en un mensaje y después los enviaba al computador. La *fragmentación y reensamblamiento,* como se denominó, se hacía para reducir la latencia debida al *retardo de almacenamiento y expedición.* Muchas redes utilizan hoy día esta aproximación de *paquetes conmutados,* donde los paquetes se envían individualmente desde la fuente al destino. La Figura 9.22 resume el rendimiento, distancia y costes de diversas redes.

Aunque estas redes se han presentado aquí como alternativas, un sistema de computadores realmente es una jerarquía de redes, como muestra la

Red	Rendimiento (Kbits/s)	Distancia (km)	Coste cable	Conecta a coste de red	Conector a coste de computador
RS232	19	0,1	0,25$ /pie	1-5$ /conector	5$/chip puerto serie
Ethernet	10 000	1	1-5$ /pie	100$ /transceptor	50$/chip interfaz Ethernet
ARPANET	50	10 000	10 000$ /mes	50 000-100 000$/IMP	5 000-10 000$ conexión/IMP

FIGURA 9.22 Rendimiento, distancia máxima y costes de tres redes ejemplo. Una Internet es simplemente múltiples Ethernets y un puente, que cuesta aproximadamente de 2 000 a 5 000 dólares, o una pasarela, que cuesta entre 20 000 y 50 000 dólares.

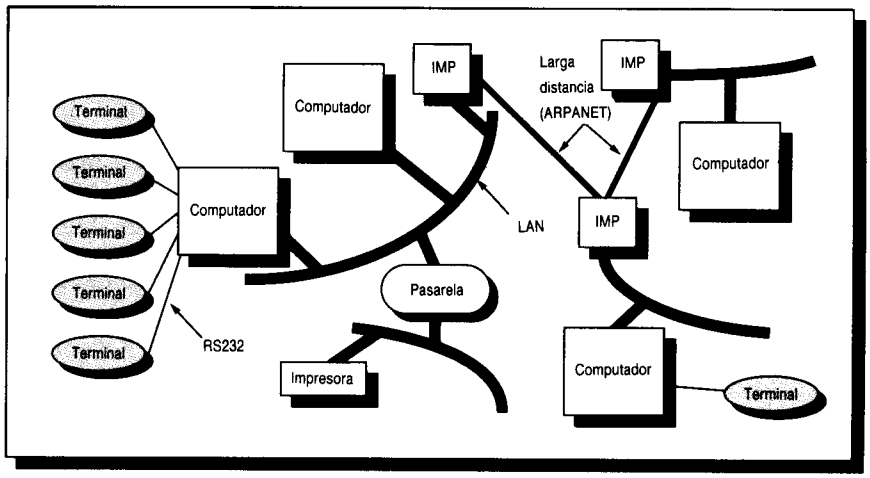


FIGURA 9.23 Un sistema de computadores actual participa en una jerarquía de redes. Idealmente, el usuario no es consciente de qué red está utilizando al realizar las tareas. Las pasarelas dirigen los paquetes a una red particular; una red dirige los paquetes a un computador particular y el computador envía los paquetes a un proceso particular.

Figura 9.23. Para tratar con esta jerarquía de redes que conecta máquinas que comunican de forma diferente, debe haber interfaces software estándares para tratar mensajes. Existen y se denominan *protocolos* y, normalmente, están estratificados para que sirvan de interfaz con los diferentes niveles de software en los sistemas de computadores. El coste adicional de estos protocolos puede robar una parte significativa del ancho de banda de la red.

Igual que con los discos en la Figura 9.6, hay un intercambio de latencia y productividad en las redes. Mensajes pequeños dan la latencia más baja en muchas redes, pero también producen una anchura de banda más baja para la red; análogamente, una red puede lograr un mayor ancho de banda con el coste de una mayor latencia.

9.5 Buses. Conectando dispositivos de E/S a CPU/memoria

En un sistema de computadores, los diversos subsistemas deben tener interfaces entre sí; por ejemplo, la memoria y la CPU necesitan comunicarse, así como la CPU y los dispositivos de E/S. Esto, normalmente, se realiza con un *bus*. El bus sirve como enlace de comunicación compartido entre los subsistemas. Las dos principales ventajas de la organización bus son bajo coste y versatilidad. Al definir un sencillo esquema de interconexión, se pueden añadir fácilmente nuevos dispositivos y los periféricos pueden incluso compar-

tirse entre sistemas de computadores que utilicen un bus común. El coste es bajo, ya que un simple conjunto de cables es un camino múltiple compartido.

La principal desventaja de un bus es que crea un cuello de botella de comunicación, limitando posiblemente la máxima productividad de las E/S. Cuando las E/S deben pasar a través de un bus central, esta limitación de ancho de banda es tan real como —y a veces más severa que— el ancho de banda de memoria. En los sistemas comerciales, donde las velocidades E/S son muy frecuentes, y en los supercomputadores, donde las velocidades de E/S necesarias son muy altas porque el rendimiento de la CPU es alto, diseñar un sistema de bus capaz de cumplir las demandas del procesador es un desafío importante.

Una razón, por la que el diseño del bus es tan difícil, es que la máxima velocidad del bus está limitada por factores físicos: la longitud del bus y el número de dispositivos (y, por consiguiente, la carga del bus). Estos límites físicos prohiben velocidades de bus arbitrarias. El deseo de altas velocidades de E/S (baja latencia) y alta productividad de las E/S también puede conducir a requerimientos de diseño conflictivos.

Los buses tradicionalmente se clasifican en *buses de CPU-memoria* o *buses de E/S*. Los buses de E/S pueden ser más largos, pueden tener muchos tipos de dispositivos conectados a ellos, tienen un amplio rango en el ancho de banda de datos de los dispositivos conectados a ellos (ver Fig. 9.1) y, normalmente, siguen un estándar de bus. Los buses CPU-memoria, por otro lado, son cortos, generalmente de alta velocidad, y adaptados al sistema de memoria para maximizar el ancho de banda memoria-CPU. Durante la fase de diseño, el diseñador de un bus CPU-memoria conoce todos los tipos de dispositivos que deben conectarse juntos, mientras que el diseñador del bus de E/S debe aceptar dispositivos que varían en posibilidades de latencia y ancho de banda. Para bajar costes, algunos computadores tienen un solo bus para memoria y dispositivos de E/S.

Consideremos una transacción típica de un bus. Una *transacción del bus* incluye dos partes: enviar la dirección y recibir o enviar el dato. Las transacciones del bus, normalmente, se definen por lo que hacen en memoria: una transacción de *lectura* transfiere datos *desde* memoria (a la CPU o a un dispositivo de E/S), y una transacción de *escritura* escribe datos en memoria. En una transacción de lectura, se envía primero la dirección desde el bus a memoria, junto con señales de control adecuadas que indican una lectura. La memoria responde devolviendo el dato al bus con señales de control adecuadas. Una transacción de escritura requiere que la CPU o dispositivo de E/S envíe dirección y dato y no requiere vuelta de datos. Habitualmente, la CPU debe esperar entre el envío de la dirección y la recepción del dato de una lectura, pero la CPU, con frecuencia, no espera las escrituras.

El diseño de un bus presenta varias opciones, como muestra la Figura 9.24. Igual que el resto del sistema de computadores, las decisiones dependerán de los objetivos de coste y rendimiento. Las tres primeras opciones de la figura son elecciones claras —líneas de datos y direcciones separadas, líneas de datos más anchas y transferencias de múltiples palabras dan mayor rendimiento a más coste.

El siguiente elemento de la tabla está relacionado con el número de *amos del bus (bus masters)*. Estos son dispositivos que pueden iniciar una transac-

Opción	Alto rendimiento	Bajo coste
Ancho del bus	Direcciones y líneas de datos separadas	Múltiples direcciones y líneas de datos
Ancho de los datos	Más ancho es más rápido (p. ej., 32 bits)	Más delgado es más barato (p. ej., 8 bits)
Tamaño de transferencia	Múltiples palabras tienen menos gasto de bus	La transferencia de una sola palabra es más simple
Amos del bus	Múltiple (requiere arbitración)	Unico amo (no arbitración)
¿Dividir transacción?	Sí—separar paquetes de Petición y Respuesta obtiene un ancho de banda mayor (necesita múltiples maestros)	No—la conexión continua es más barata y tiene menos latencia
Reloj	Síncrono	Asíncrono

FIGURA 9.24 Opciones principales para un bus. La ventaja de buses de datos y direcciones separados están principalmente en las escrituras.

ción de lectura o escritura; la CPU, por ejemplo, es siempre un amo del bus. Un bus tiene múltiples amos cuando hay múltiples CPU o cuando los dispositivos de E/S pueden iniciar una transacción del bus. Si hay múltiples amos, se requiere un esquema de arbitración entre los amos para decidir cual tiene acceso al bus en cada momento. La arbitración es, con frecuencia, de una prioridad fija, como es el caso con dispositivos encadenados por margarita o un esquema aproximadamente regular que escoja aleatoriamente el amo que obtiene el bus.

Con múltiples amos un bus puede ofrecer mayor ancho de banda al enviar los paquetes, en contraposición a mantener el bus durante la transacción completa. Esta técnica se designa de *transacciones divididas (split transactions).* (Algunos sistemas llaman a esta posibilidad *conexión/desconexión* o *bus segmentado.)* La transacción de lectura se descompone en una transacción de petición de lectura que contiene la dirección y una transacción de réplica de memoria que contiene el dato. Cada transacción debe ahora señalizarse para que CPU y memoria puedan decir quién es quién. Las transacciones divididas hacen el bus disponible para otros amos mientras la memoria lee las palabras desde la dirección requerida. También significa, normalmente, que la CPU debe arbitrar para que el bus envíe los datos, y la memoria debe arbitrar para que el bus los devuelva. Por tanto, un bus de transacciones divididas tiene mayor anchura de banda, pero habitualmente tiene mayor latencia que un bus que se mantiene durante la transacción completa.

El elemento final, el reloj, está relacionado con que el bus sea síncrono o asíncrono. Si un bus es *síncrono* incluye un reloj en las líneas de control y un protocolo fijo para direcciones y datos relativos al reloj. Como no se necesita ninguna o muy poca lógica para decidir qué hacer a continuación, estos buses pueden ser rápidos y baratos. Sin embargo, tienen dos desventajas importantes. Todo lo que pase por el bus debe correr a la misma frecuencia de reloj y, debido a los problemas de solapamiento de reloj, los buses síncronos no pueden ser largos. Los buses de CPU-memoria normalmente son síncronos.

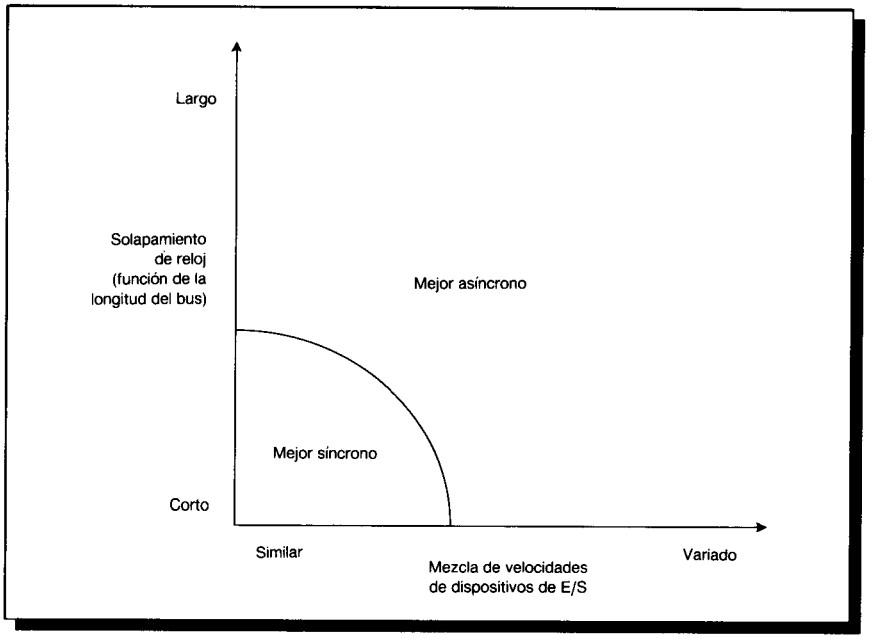


FIGURA 9.25 Tipo de bus preferido como función del solapamiento longitud/reloj y variación de la velocidad del dispositivo de E/S. Lo síncrono es mejor cuando la distancia es pequeña y los dispositivos de E/S en el bus todos transfieren a velocidades similares.

Por otro lado, un bus *asíncrono* no tiene reloj. En su lugar, se utilizan protocolos autotemporizados de establecimiento de comunicación entre emisor y el receptor en el bus. Este esquema hace mucho más fácil acomodar una amplia variedad de dispositivos y la longitud del bus sin que importen los solapamientos de reloj ni los problemas de sincronización. Si se puede utilizar un bus síncrono, habitualmente es más rápido que un bus asíncrono debido a los gastos de sincronización del bus para cada transacción. La elección del bus síncrono frente al asíncrono tiene implicaciones no sólo por el ancho de banda de los datos sino también por la capacidad del sistema de E/S en términos de distancia física y número de dispositivos que se pueden conectar al bus; los buses asíncronos escalan mejor con los cambios tecnológicos. Los buses de E/S normalmente son asíncronos. La Figura 9.25 sugiere cuándo es mejor utilizar uno que otro.

Buses estándares

El número y variedad de dispositivos de E/S no es fijo en la mayoría de los sistemas de computadores, permitiendo que los clientes confeccionen los computadores a sus necesidades. Como la interfaz a la que se conectan los dispositivos, el bus de E/S también se puede considerar como un bus de expansión para añadir dispositivos de E/S a lo largo del tiempo. Los estándares que

permiten al diseñador de computadores y al diseñador de dispositivos de E/S trabajar independientemente, por tanto, juegan un gran papel a la hora de elegir los buses. Mientras que el diseñador del sistema de computadores y el de los dispositivos de E/S cumplan los requerimientos, cualquier dispositivo de E/S se puede conectar a cualquier computador. En efecto, un estándar de bus de E/S es el documento que define cómo conectarlos.

	VME bus	FutureBus	Multibus II	IPI	SCSI
Anchura de bus (señales)	128	96	96	16	8
¿Direcciones/datos multiplexados?	No multiplexado	Multiplexado	Multiplexado	N/A	N/A
Anchura de datos (principal)	16 a 32 bits	32 bits	32 bits	16 bits	8 bits
Tamaño de transferencia	Simple o múltiple	Simple o múltiple	Simple o múltiple	Simple o múltiple	Simple o múltiple
Número de amos del bus	Múltiple	Múltiple	Múltiple	Simple	Múltiple
¿División de transacciones?	No	Opcional	Opcional	Opcional	Opcional
Reloj	Asíncrono	Asíncrono	Síncrono	Asíncrono	Cualquiera
Ancho de banda, palabra simple, 0 ns acceso a memoria	25,0 MB/s	37,0 MB/s	20,0 MB/s	25,0 MB/s	5,0 MB/s o 1,5 MB/s
Ancho de banda, palabra simple, 150 ns acceso a memoria	12,9 MB/s	15,5 MB/s	10,0 MB/s	25,0 MB/s	5,0 MB/s o 1,5 MB/s
Ancho de banda, palabras múltiples, (longitud de bloque infinita), 0 ns acceso a memoria	27,9 MB/s	95,2 MB/s	40,0 MB/s	25,0 MB/s	5,0 MB/s o 1,5 MB/s
Ancho de banda, palabras múltiples (longitud de bloque infinita), 150 ns acceso a memoria	13,6 MB/s	20,8 MB/s	13,3 MB/s	25,0 MB/s	5,0 MB/s o 1,5 MB/s
Máximo número de dipositivos	21	20	21	8	7
Longitud máxima de bus	0,5 metros	0,5 metros	0,5 metros	50 metros	25 metros
Estándar	IEEE 1014	IEEE 896.1	ANSI/IEEE 1296	ANSI X3.129	ANSI X3.131

FIGURA 9.26 Información sobre cinco buses estándares. Los tres primeros fueron definidos originalmente como buses CPU-memoria y los dos últimos como buses de E/S. Para los buses CPU-memoria los cálculos de ancho de banda suponen un bus completamente cargado y están dados por transferencias de palabras simples y de bloques de longitud ilimitada; las medidas se muestran ignorando la latencia de memoria y suponiendo un tiempo de acceso a 150 ns. El ancho de banda supone que la distancia media de una transferencia es una tercio de la longitud del corredor de fondo. (Los datos de las tres primeras columnas son de Borril [1986].) El ancho de banda para los buses de E/S está dada como su máxima frecuencia de transferencia de datos. El estándar SCSI ofrece E/S asíncrona o síncrona; la versión asíncrona transfiere a 1,5 MB/s y la síncrona a 5 MB/s.

Las máquinas, a veces, se hacen tan populares que sus buses de E/S se convierten de hecho en estándares; ejemplos son el Unibus del PDP-11 y el Bus del PC-AT de IBM. Una vez que se han construido muchos dispositivos de E/S para la máquina popular, otros diseñadores de computadores construyen sus interfaces de E/S para que aquellos dispositivos puedan conectarse también en sus máquinas. A veces los estándares también provienen de un esfuerzo explícito de estandarización por parte de los fabricantes de dispositivos de E/S. La *interfaz de periféricos inteligentes* (IPI) y Ethernet son ejemplos de estándares conseguidos por la cooperación de fabricantes. Si los estándares tienen éxito, eventualmente son bendecidos por un cuerpo sancionador como ANSI o IEEE. Ocasionalmente, un bus estándar proviene directamente de arriba a abajo de un comité de estándares —el FutureBus es un ejemplo.

La Figura 9.26 resume las características de varios buses estándares. Observar que las entradas del ancho de banda de la figura no se listan como simples números para los buses de CPU-memoria (VME, FutureBus y Multibus II). A causa de los gastos del bus, el tamaño de la transferencia afecta significativamente a la anchura de banda. Como el bus, habitualmente, transfiere, a o desde, memoria, la velocidad de memoria también afecta a la anchura de banda. Por ejemplo, con tamaño de transferencia infinita y memoria infinitamente rápida (0 ns), FutureBus es el 240 por 100 más rápido que VME, pero FutureBus es sólo aproximadamente el 20 por 100 más rápido que VME para transferencias de una palabra desde una memoria de 150 ns.

9.6 Interfaz con la CPU

Una vez que se han descrito los dispositivos de E/S y examinado algunas de las posibilidades de conexión al bus, estamos preparados para explicar la interfaz con la CPU. La primera pregunta es cómo puede hacerse la conexión física del bus de E/S. Las dos elecciones son conectarlo a la memoria o a la cache. En la sección siguiente explicaremos los pros y contras de conectar un bus de E/S directamente a la cache; en esta sección examinamos el caso más usual, en el cual el bus de E/S se conecta al bus de memoria principal. La Figura 9.27 muestra una organización típica. En los sistemas de bajo coste, el bus de E/S es el bus de memoria; esto significa que una orden de E/S en el bus puede interferir con la extracción de una instrucción de la CPU, por ejemplo.

Una vez elegida la interfaz física, la pregunta se convierte en ¿cómo la CPU direcciona un dispositivo de E/S que necesite para enviar o recibir datos? La práctica más común se denomina E/S *mapeada en memoria (memory-mapped)*. En este esquema, parte del espacio de direcciones se asigna a los dispositivos de E/S. Las lecturas y escrituras en aquellas direcciones pueden hacer que se transfieran los datos; parte del espacio de E/S también puede inicializarse además para los dispositivos de control; así, las órdenes al dispositivo son exactamente accesos a esas direcciones mapeadas en memoria. La alternativa práctica es utilizar códigos de operación de E/S específicos en la CPU. En este caso, la CPU envía una señal de que esta dirección es para dispositivos

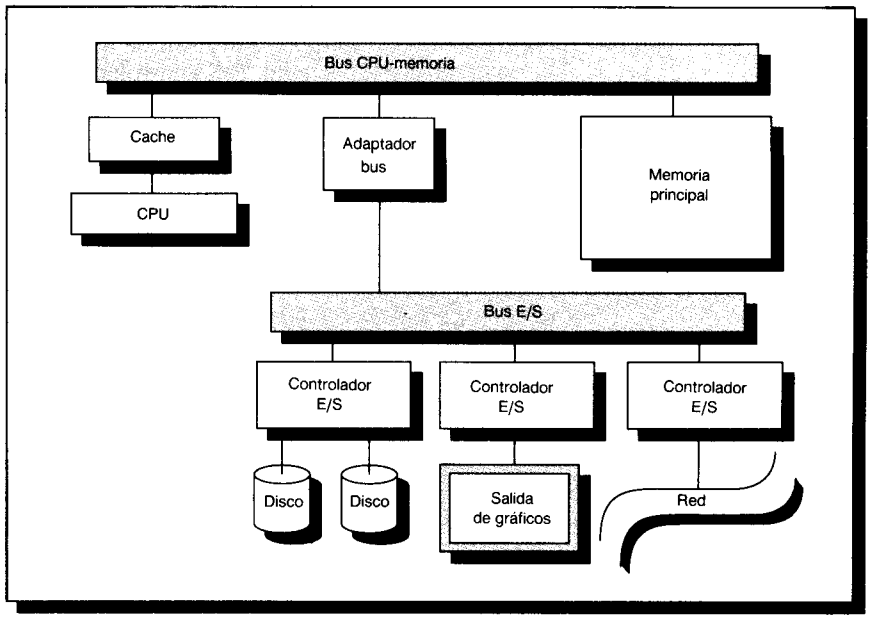


FIGURA 9.27 Una interfaz típica de dispositivos de E/S y un bus de E/S al bus CPU-memoria.

de E/S. Ejemplos de computadores con instrucciones de E/S son el Intel 80 $\times$ 86 y los computadores IBM 370. Sin importar qué esquema de direccionamiento se seleccione, cada dispositivo de E/S tiene registros que proporcionan información sobre el estado y control. Bien mediante cargas y almacenamientos de las E/S mapeadas en memoria o a través de instrucciones especiales, la CPU inicializa los señalizadores para determinar la operación que realizará el dispositivo de E/S.

La E/S raramente es una simple operación. Por ejemplo, la impresora de línea DEC LP11 tiene dos registros de dispositivos de E/S: uno para información sobre el estado y otro para los datos que se van a imprimir. El registro de estado contiene un *bit de hecho (done bit),* inicializado por la impresora cuando ha impreso un carácter, y un *bit de error,* que indica que la impresora está atascada o sin papel. Cada byte de datos que se van a imprimir se coloca en el registro de datos; la CPU debe entonces esperar hasta que la impresora ponga a 1 el bit de hecho antes que se pueda colocar otro carácter en el buffer.

Esta sencilla interfaz, en la cual la CPU comprueba periódicamente los bits de estado para ver si es el momento para realizar la siguiente operación de E/S, se denomina *encuesta (polling).* Como se puede esperar, el hecho que las CPU sean mucho más rápidas que los dispositivos de E/S significa que la encuesta puede emplear mucho tiempo de CPU. Esto fue reconocido hace mucho tiempo, dejando a la invención de las interrupciones notificar a la CPU cuándo es el momento de hacer algo para el dispositivo de E/S. Las E/S *controladas por interrupciones,* utilizadas por muchos sistemas para algunos dispositivos, permiten que la CPU funcione en otros procesos mientras se espera

al dispositivo de E/S. Por ejemplo, la LP11 tiene un modo que permite interrumpir a la CPU siempre que el bit de hecho o de error esté a 1. En aplicaciones de propósito general, E/S controladas por interrupciones, son la clave para los sistemas operativos multitarea y buenos tiempos de respuesta.

El inconveniente para las interrupciones es el coste adicional de sistema operativo en cada evento. En aplicaciones de tiempo real con cientos de eventos de E/S por segundo, este gasto puede ser intolerable. Una solución híbrida para sistemas de tiempo real es utilizar un reloj para interrumpir periódicamente la CPU; en ese tiempo la CPU explora todos los dispositivos de E/S.

Delegación de la responsabilidad de E/S de la CPU

Las E/S controladas por interrupciones alivian a la CPU de esperar para cada evento de E/S, pero todavía hay muchos ciclos de CPU que se gastan en la transferencia de datos. La transferencia de un bloque del disco de 2 048 palabras, por ejemplo, requeriría como mínimo 2 048 cargas y 2 048 almacenamientos, así como el gasto de la interrupción. Como los eventos de E/S involucran, con frecuencia, transferencias de bloque, a muchos sistemas de computadores se añade hardware de *acceso directo a memoria —direct memory access (DMA)—* para permitir transferencias de grupos de palabras sin intervención de la CPU.

El DMA es un procesador especializado que transfiere datos entre memoria y un dispositivo de E/S, mientras la CPU realiza otras tareas. Por tanto, es externo a la CPU y debe actuar como un amo en el bus. La CPU inicializa primero los registros del DMA, que contienen una dirección de memoria y el número de bytes que se van a transferir. Una vez que se completa la transferencia del DMA, el controlador interrumpe la CPU. Puede haber múltiples dispositivos del DMA en un sistema de computadores; por ejemplo, el DMA normalmente es parte del controlador de un dispositivo de E/S.

Incrementando la inteligencia de los dispositivos del DMA se puede además aliviar a la CPU. Los dispositivos denominados *procesadores de E/S* (o *controladores de E/S* o *controladores de canal)* operan desde programas fijos o desde programas cargados por el sistema operativo. El sistema operativo normalmente inicializa una cola de *bloques de control de E/S* que contiene información, tal como posición de los datos (fuente y destino) y tamaño de los datos. El procesador de E/S entonces toma elementos de la cola, haciendo todo lo que se necesite y envía una simple interrupción cuando se complete la tarea especificada en el bloque de control de E/S. Mientras que la impresora de línea LP11 puede provocar 4 800 interrupciones para imprimir una página de 60 líneas por 80 caracteres, un procesador de E/S podría ahorrar 4 799 de esas interrupciones.

Los procesadores de E/S pueden compararse a los multiprocesadores, ya que facilitan la ejecución de varios procesos simultáneamente en el sistema de computadores. Los procesadores de E/S son menos generales que las CPU; sin embargo, puesto que tienen tareas dedicadas, por ello, el paralelismo está también mucho más limitado. Además, un procesador de E/S normalmente no modifica la información, como hace una CPU, sino que la desplaza de un sitio a otro.

9.7 Interfaz con un sistema operativo

De manera análoga a la forma que los compiladores utilizan un repertorio de instrucciones (ver Sección 3.7 del Cap. 3), los sistemas operativos controlan qué técnicas de E/S implementadas por hardware serán actualmente usadas. Por ejemplo, muchos controladores de E/S utilizados en los primeros sistemas UNIX eran microprocesadores de 16 bits. Para evitar problemas con las direcciones de 16 bits en los controladores, UNIX se cambió para que limitase la máxima transferencia de E/S a 63 KB; en el instante de la publicación de este libro, ese límite se mantiene todavía. Por tanto, un nuevo controlador de E/S diseñado para transferir eficientemente ficheros de 1 MB nunca vería más que 63 KB a la vez bajo UNIX, sin importar lo grande que sean los ficheros.

Las caches causan problemas a los sistemas operativos. Datos obsoletos

La prevalencia de las caches en los sistemas de computadores se ha sumado a las responsabilidades del sistema operativo. Las caches implican la posibilidad de dos copias de los datos —una para la cache y otra para memoria principal— mientras la memoria virtual puede dar como resultado tres copias —para la cache, memoria y disco—. Esto conlleva la posibilidad de *datos obsoletos:* la CPU o el sistema de E/S podrían modificar una copia sin actualizar las demás (ver Sección 8.8 del Cap. 8). Bien el sistema operativo o el hardware deben asegurar que la CPU lea el dato introducido más recientemente y que las E/S saquen el dato correcto, en la presencia de caches y memoria virtual. El problema de datos obsoletos depende en parte de cómo estén conectadas las E/S al computador. Si están conectadas a la cache de la CPU, como muestra la Figura 9.28, no hay problemas de datos obsoletos; todos los dispositivos de E/S y la CPU ven la versión más correcta en la cache, y los mecanismos existentes en la jerarquía de memoria aseguran que las otras copias de los datos se actualizarán. El efecto lateral es la pérdida de rendimiento de la CPU, ya que las E/S reemplazarán bloques de la cache con datos que son improbablemente necesarios para la ejecución del proceso en la CPU en el instante de la transferencia. En otras palabras, todos los datos de E/S van a través de la cache pero pocos se referencian. Esta organización necesita también arbitración entre la CPU y las E/S para decidir quién accede a la cache. Si la E/S está conectada a memoria, como en la Figura 9.27, entonces no interfiere con la CPU, con tal que la CPU tenga una cache. Sin embargo, en esta situación se presenta el problema de datos obsoletos. Alternativamente, las E/S pueden invalidar los datos —o todos los datos que puedan coincidir (no comprueban etiqueta) o sólo los datos que coinciden.

Hay dos partes en el problema de datos obsoletos:

1. El sistema de E/S ve datos obsoletos en la salida porque la memoria no está actualizada.

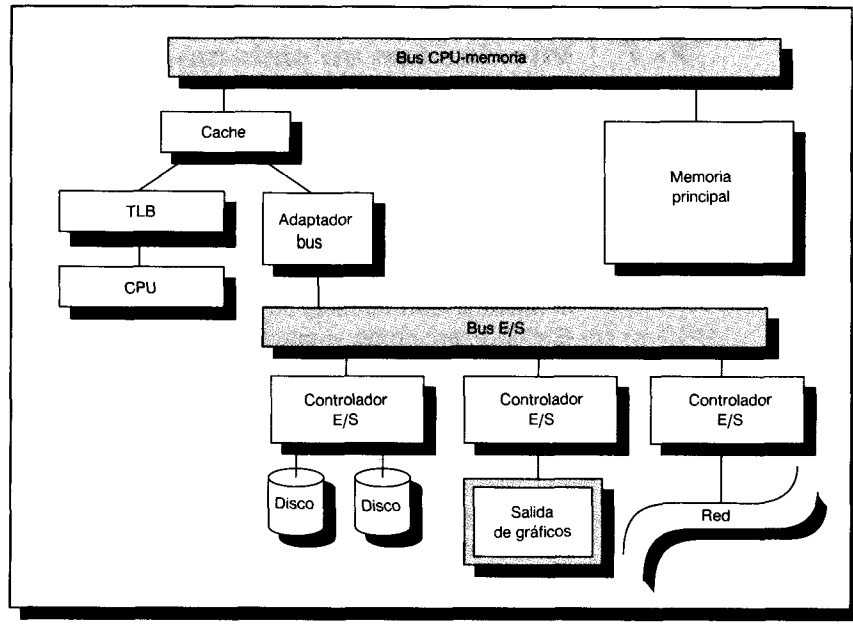


FIGURA 9.28 Ejemplo de E/S conectada directamente a la cache.

2. La CPU ve datos obsoletos en la cache o en la entrada después que el sistema de E/S haya actualizado memoria.

El primer dilema es cómo sacar datos correctos si hay una cache y la E/S está conectada a memoria. Una cache de escritura directa resuelve esto asegurando que la memoria tendrá los mismos datos que la cache. Una cache de postescritura requiere que el sistema operativo limpie las direcciones de salida para asegurarse que no están en la cache. Esto lleva tiempo, aun cuando el dato no esté en la cache, ya que las comprobaciones de direcciones son secuenciales. Alternativamente, el hardware puede comprobar las etiquetas de la cache durante la salida para ver si están en una cache de postescritura, y sólo interactúa con la cache si la salida intenta leer el dato que está en la cache.

El segundo problema es asegurar que la cache no tenga datos obsoletos después de la entrada. El sistema operativo puede garantizar que, posiblemente, el área de entrada de datos no pueda estar en la cache. Si no se puede garantizar esto, el sistema operativo limpia las direcciones de entrada para asegurarse que no están en la cache. De nuevo, esto lleva tiempo, estén o no en la cache las direcciones de entrada. Como antes, se puede añadir hardware extra para comprobar etiquetas durante una entrada e invalidar los datos si hay un conflicto. Estos problemas son básicamente los mismos que la coherencia cache en un multiprocesador, explicada en la Sección 8.8 del Capítulo 8; las E/S se pueden considerar como un segundo procesador dedicado en un multiprocesador.

DMA y memoria virtual

Dado el uso de la memoria virtual, existe el problema si el DMA debe hacer transferencias utilizando direcciones virtuales o físicas. Hay algunos problemas con el DMA utilizando E/S con correspondencia física:

- Transferir un buffer que es mayor que una página causará problemas, ya que las páginas del buffer, habitualmente, no corresponderán a páginas secuenciales en la memoria física.
- Suponer que el DMA está funcionando entre memoria y buffer de encuadre y que el sistema operativo elimina algunas de las páginas de memoria (o las reubica). El DMA entonces transferiría datos a o desde la página errónea de memoria.

Una respuesta a estas preguntas es el *DMA virtual.* Permite que el DMA utilice direcciones virtuales que se corresponden a direcciones físicas durante el DMA. Por ello, un buffer debe ser secuencial en memoria virtual pero las páginas pueden estar dispersas en la memoria física. El sistema operativo podría actualizar las tablas de dirección de un DMA si se desplaza un proceso utilizando un DMA virtual o también podría «bloquear» las páginas en memoria hasta que se complete el DMA. La Figura 9.29 muestra registros de traducción de direcciones añadidos al dispositivo de DMA.

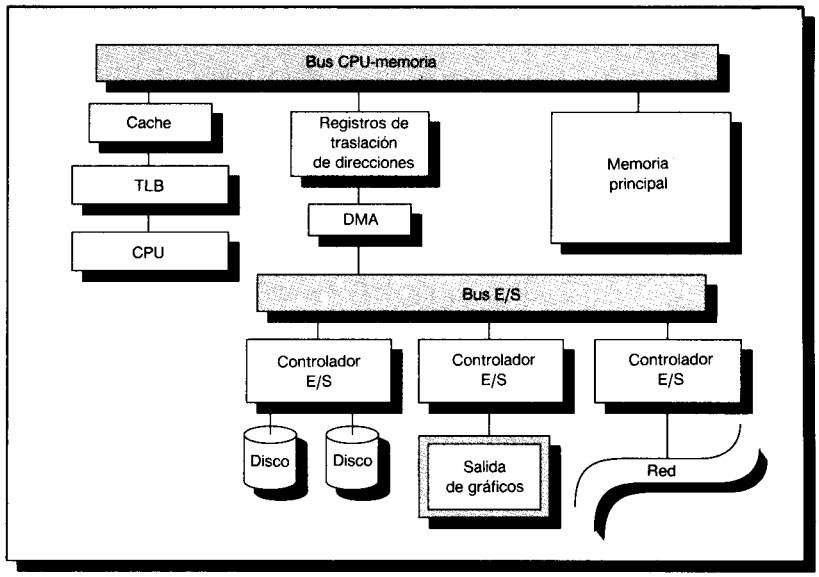


FIGURA 9.29 El DMA virtual requiere un registro para que cada página sea transferida al controlador DMA, mostrando los bits de protección y la página física correspondiente a cada página virtual.

Caches ayudando a los sistemas operativos. Caches de fichero o discos

Aunque la invención de las caches hizo la vida de los diseñadores de sistemas operativos más difícil, los diseñadores de sistemas operativos interesados por el rendimiento realizan optimizaciones como las de las caches, utilizando la memoria principal como una «cache» para el tráfico del disco con el fin de mejorar el rendimiento de las E/S. El impacto de utilizar la memoria principal como un buffer o una cache para los accesos a ficheros o discos se demuestra en la Figura 9.30. Esta figura muestra el cambio en las E/S del disco para un sistema sin cache medido como frecuencia de fallos (ver Sección 8.2 del Cap. 8). Las caches de los ficheros o de los discos cambian el número de E/S del disco y de la mezcla de lecturas y escrituras; dependiendo del tamaño de la cache y de la política de escritura, entre el 50 y el 70 por 100 de todos los accesos al disco podrían llegar a ser escrituras con estas caches. Sin caches de ficheros o de disco, entre el 15 y el 33 por 100 de todos los accesos son escrituras, dependiendo del entorno.

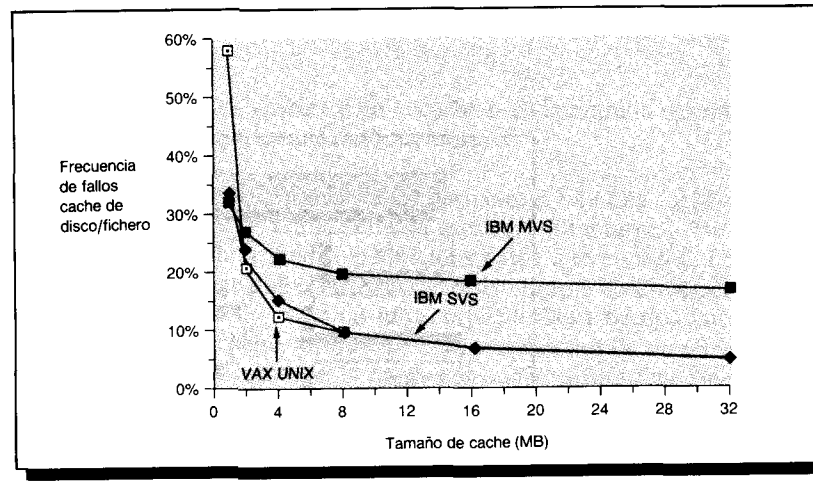


FIGURA 9.30 Efectividad de una cache de fichero o de disco al reducir E/S del disco frente a tamaño de cache. Ousterhout y cols. [1985] coleccionaron los datos VAX UNIX en VAX-11/785s *con una memoria principal de* 8 MB a 16 MB, corriendo UNIX 4.2 BSD utilizando un tamaño de bloque de 16 KB. Smith [1985] coleccionó las trazas IBM SVS e IBM MVS en IBM 370/168 utilizando un tamaño de bloque de una pista (que variaba desde 7 294 bytes a 19 254 bytes, dependiendo del disco). La diferencia entre una cache de fichero y una de disco es que la de fichero utiliza números lógicos de bloque mientras que la de disco utiliza direcciones que tienen una correspondencia con un sector físico y una pista en el disco. Esta diferencia es similar a la diferencia entre una cache direccionada virtualmente y direccionada físicamente (ver Sección 8.8 del Capítulo 8).

9.8 Diseño de un sistema de E/S

El arte de las E/S es encontrar un diseño que cumpla los objetivos de coste y diversidad de dispositivos mientras se evitan cuellos de botella en el rendimiento de las E/S. Esto significa que los componentes deben estar equilibrados entre memoria principal y el dispositivo de E/S porque el rendimiento —y, por tanto, el coste/rendimiento efectivo— puede solamente ser tan bueno como el eslabón más débil de la cadena de E/S. El arquitecto también debe planificar la expansión, ya que los clientes pueden confeccionar las E/S para sus aplicaciones. Esta expansibilidad, tanto en número como en tipos de dispositivos de E/S, tiene su coste en conectores mayores, mayores fuentes de alimentación para soportar los dispositivos de E/S y mayores carcasas.

Al diseñar un sistema de E/S, analizar rendimiento, coste y capacidad utilizando diversos esquemas de conexión de E/S y diferentes números de dispositivos de E/S de cada tipo. Hay una serie de seis pasos a seguir en el diseño de un sistema de E/S. Las respuestas en cada paso pueden estar dictadas por requerimientos de mercado o simplemente por objetivos de coste/rendimiento.

1. Listar los diferentes tipos de dispositivos de E/S que se van a conectar a la máquina, o buses estándares que soportará la máquina.
2. Listar los requerimientos físicos para cada dispositivo de E/S. Esto incluye volumen, alimentación, conectores, ranuras del bus, carcasas de expansión, etc.
3. Listar el coste de cada dispositivo de E/S, incluyendo la parte de coste de cualquier controlador necesario para este dispositivo.
4. Registrar las demandas de recursos de la CPU de cada dispositivo de E/S. Esto debería incluir:

   Ciclos de reloj por instrucciones utilizadas para iniciar una E/S, soportar operaciones de un dispositivo de E/S (como, por ejemplo, tratamiento de interrupciones) y completar la E/S

   Detenciones de reloj de la CPU debidas a esperas para que las E/S terminen de utilizar la memoria, bus o cache

   Ciclos de reloj de la CPU para recuperarse de una actividad de E/S, tal como una limpieza de cache

5. Listar las demandas de recursos del bus de E/S y de memoria de cada dispositivo. Aun cuando la CPU no esté utilizando la memoria, la anchura de banda de la memoria principal y del bus de E/S son limitadas.
6. El paso final es establecer el rendimiento de las diferentes formas de organizar estos dispositivos de E/S. El rendimiento sólo se puede evaluar adecuadamente con la simulación, aunque se pueda estimar utilizando la teoría de colas.

Después, se selecciona la mejor organización, dados sus objetivos de coste y rendimiento.

Los objetivos de coste y rendimiento afectan a la selección del diseño físico y esquema de E/S. El rendimiento se puede medir o en megabytes por segundo o E/S por segundo, dependiendo de las necesidades de la aplicación. Para un rendimiento alto, los únicos límites deberían ser: velocidad de los dispositivos de E/S, número de dispositivos de E/S, y velocidad de memoria y CPU. Para bajo coste, los únicos gastos deberían ser los de los dispositivos de E/S y la conexión con la CPU. El diseño del coste/rendimiento, por supuesto, presenta lo mejor de ambos mundos.

Para aclarar estas ideas, veamos algunos ejemplos.

Ejemplo

Primero, examinemos el impacto en la CPU, de leer una página del disco directamente en la cache. Hacer las siguientes suposiciones:

Cada página tiene 8 KB y el tamaño de bloque de la cache es de 16 bytes.

Las direcciones correspondientes a una nueva página **no** están en la cache.

La CPU no accederá a ninguno de los datos de la nueva página.

El 90 por 100 de los bloques que fueron desplazados de la cache se leerán de nuevo y cada uno provocará un fallo.

La cache usa postescritura y el 50 por 100 de los bloques son modificados en promedio.

El sistema de E/S mantiene provisionalmente un bloque completo de la coincidencache antes de escribir en la cache (esto se denomina *buffer de adaptación de velocidad (buffer speed-matching)* coincidiendo la anchura de banda de transferencia del sistema de E/S y de memoria).

Los accesos y fallos están uniformemente distribuidos en todos los bloques de cache.

No hay ninguna otra interferencia entre la CPU y E/S para los huecos de la cache.

Hay 15 000 fallos por cada millón de ciclos de reloj cuando **no** hay E/S.

La penalización de faltas es de 15 ciclos de reloj, más 15 ciclos adicionales para escribir el bloque si estaba modificado.

Suponiendo que una página se trae cada millón de ciclos de reloj, ¿cuál es el impacto en el rendimiento?

Respuesta

En cada página caben 8192/16 o 512 bloques. Las transferencias de E/S no provocan fallos de cache de su propiedad porque se transfieren bloques enteros de la cache. Sin embargo, desplazan bloques ya en la cache. Si la mitad de los bloques desplazados están ocupados, cuesta 256 · 15 ciclos de reloj volverlos a escribir en memoria. Hay también fallos del 90 por 100 de los bloques desplazados en la cache porque son referenciados más tarde, añadiendo otras 90 % · 512 o sea 461 fallos. Desde que este dato se colocó en la cache desde

el sistema de E/S, todos estos bloques están modificados y necesitarán ser escritos de nuevo cuando se sustituyan. Por tanto, el total es 256 · 15 + 461 · 30 ciclos de reloj más que el 1 000 000 + 15 000 · 15 original. Esto lleva a un 1 por 100 de disminución en el rendimiento:

$$\frac{265 \cdot 15 + 461 \cdot 30}{1\,000\,000 + 15\,000 \cdot 15} = \frac{17\,670}{1\,225\,000} = 0{,}014$$

Ahora, examinemos el coste/rendimiento de diferentes organizaciones de E/S. Una forma sencilla de realizar este análisis es examinar la máxima productividad, suponiendo que los recursos puedan ser utilizados al 100 por 100 o a su máxima frecuencia sin efectos laterales de interferencia. Un ejemplo posterior emplea una visión más realista.

Ejemplo

Dadas las siguientes informaciones de coste y rendimiento:

una CPU de 50 MIPS que cuesta 50 000 dólares

una memoria de 8 bytes de ancho con un tiempo de ciclo de 200 ns

bus de E/S de 80 MB/s con espacio para 20 buses y controladores SCSI

Buses SCSI que pueden transferir 4 MB/s y soportar hasta 7 discos por bus (éstos también se denominan *cadenas* SCSI)

un controlador SCSI de 2 500 dólares que añade 2 milisegundos (ms) de gasto para realizar una E/S de disco

un sistema operativo que utiliza 10 000 instrucciones CPU para una E/S de disco

una elección de un gran disco que contiene 4 GB o un pequeño disco que contiene 1 GB, cada uno de los cuales cuesta 3 dólares por MB

ambos discos giran a 3 600 RPM, siendo el tiempo medio de búsqueda de 12 ms y pueden transferir 2 MB/s

la capacidad de memoria debe ser de 100 GB, y

el tamaño medio de E/S es de 8 KB

Evaluar el coste de E/S por segundo (IOPS) al utilizar unidades pequeñas o grandes. Suponer que cada E/S del disco necesita un retardo medio de giro y de búsqueda. Utilizar la hipótesis optimista de que todos los dispositivos se pueden utilizar al 100 por 100 de su capacidad y que la carga de trabajo está uniformemente distribuida entre todos los discos.

Respuesta

El rendimiento de E/S está limitado por el eslabón más débil de la cadena; por ello evaluaremos el máximo rendimiento de cada eslabón de la cadena de E/S para cada organización con el fin de determinar el rendimiento máximo de esa organización.

Comencemos calculando el máximo número de IOPS para la CPU, memoria principal y bus de E/S. El rendimiento de E/S de la CPU está deter-

minado por la velocidad de la CPU y el número de instrucciones para realizar una E/S del disco:

$$\text{Máximo IOPS por CPU} = \frac{50 \text{ MIPS}}{10\,000 \text{ instrucciones por E/S}} = 5\,000$$

El máximo rendimiento del sistema de memoria se determina por la duración del ciclo de memoria, la anchura de la memoria y el tamaño de las transferencias de E/S:

$$\text{Máximo IOPS para memoria principal} = \frac{(1/200 \text{ ns}) \cdot 8}{8 \text{ KB por E/S}} = 5\,000$$

El máximo rendimiento del bus de E/S está limitado por la anchura de banda del bus y el tamaño de la E/S:

$$\text{Máximo IOPS para el bus de E/S} = \frac{80 \text{ MB/s}}{8 \text{ KB por E/S}} = 10\,000$$

Por tanto, sin importar lo que seleccione el disco, la CPU y memoria principal limitan el rendimiento máximo a no más de 5 000 IOPS.

Ahora es el momento de examinar el rendimiento del siguiente eslabón en la cadena de E/S, los controladores SCSI. El tiempo para transferir 8 KB sobre el bus SCSI es

$$\text{Tiempo de transferencia del bus SCSI} = \frac{8 \text{ KB}}{4 \text{ MB/s}} = 2 \text{ ms}$$

Añadir 2 ms más de coste adicional del controlador SCSI significa 4 ms por E/S, haciendo que la máxima frecuencia por controlador sea

$$\text{Máximo IOPS por controlador SCSI} = \frac{1}{4 \text{ ms}} = 250 \text{ IOPS}$$

Todas las organizaciones utilizarán varios controladores; así que 250 IOPS no es el límite para el sistema completo.

El eslabón final de la cadena son los mismos discos. El tiempo promedio de una E/S del disco es

$$\text{Tiempo de E/S} = 12 \text{ ms} + \frac{0{,}5}{3\,600 \text{ RPM}} + \frac{8 \text{ KB}}{2 \text{ MB/s}} =$$

$$= 12 + 8{,}3 + 4 = 24{,}3 \text{ ms}$$

por tanto, el rendimiento del disco es

$$\text{Máximo IOPS (usando búsquedas medias) por disco} = \frac{1}{24{,}3 \text{ ms}} \approx 41 \text{ IOPS}$$

El número de discos en cada organización depende del tamaño de cada disco: 100 GB pueden ser 25 discos de 4 GB o 100 discos de 1 GB. El máximo número de E/S para todos los discos es:

$$\text{Máximo IOPS para 25 discos de 4 GB} = 25 \cdot 41 = 1\,025$$

$$\text{Máximo IOPS para 100 discos de 1-GB} = 100 \cdot 41 = 4\,100$$

Por tanto, con tal que haya suficientes cadenas SCSI, los discos llegarán a ser el nuevo límite para el máximo rendimiento: 1 025 IOPS para los discos de 4 GB y 4 100 para los discos de 1 GB.

Aunque hemos determinado el rendimiento de cada eslabón de la cadena de E/S, todavía hay que determinar cuántos buses y controladores SCSI utilizar y cuántos discos conectar a cada controlador, y cómo esto puede, además, limitar el máximo rendimiento. El bus de E/S está limitado a 20 controladores SCSI y la SCSI estándar limita los discos a 7 por cadena SCSI. El mínimo número de controladores es, para los discos de 4 GB

$$\text{Mínimo número de cadenas SCSI para 25 discos de 4-GB} = \frac{25}{7} \text{ o } 4$$

y para discos de 1-GB

$$\text{Mínimo número de cadenas SCSI para 100 discos de 1-GB} = \frac{100}{7} \text{ o } 15$$

Podemos calcular el máximo IOPS para cada configuración:

$$\text{Máximo IOPS para } 4 \text{ cadenas SCSI} = 4 \cdot 250 = 1\,000 \text{ IOPS}$$

$$\text{Máximo IOPS para } 15 \text{ cadenas SCSI} = 15 \cdot 250 = 3\,750 \text{ IOPS}$$

El máximo rendimiento de este número de controladores es ligeramente más bajo que la productividad de E/S del disco, por ello calculemos también el número de controladores para que no lleguen a constituir un cuello de botella. Una forma es determinar el número de discos que pueden soportar por cadena:

$$\text{Número de discos por cadena SCSI en anchura de banda completa} = \frac{250}{41} = 6{,}1 \text{ o } 6$$

y después calcular el número de cadenas:

$$\text{Número de cadenas SCSI para anchura de banda completa de discos de 4-GB} = \frac{25}{6} = 4{,}1 \text{ o } 5$$

$$\text{Número de cadenas SCSI para anchura de banda completa de discos de 1-GB} = \frac{100}{6} = 16{,}7 \text{ o } 17$$

Esto establece el rendimiento de cuatro organizaciones: 25 discos de 4 GB con 4 o 5 cadenas SCSI y 100 discos de 1 GB con 15 a 17 cadenas SCSI. El rendimiento máximo de cada opción está limitado por el cuello de botella (en negrita):

4 cadenas de discos de 4-GB = Min(5 000,5 000,10 000,1 025,**1 000**) =
= 1 000 IOPS

5 cadenas de discos de 4-GB = Min(5 000,5 000,10 000,**1 025**,1 250) =
= 1 025 IOPS

15 cadenas de discos de 1-GB = Min(5 000,5 000,10 000,4 100,**3 750**) =
= 3 750 IOPS

17 cadenas de discos de 1-GB = Min(5 000,5 000,10 000,**4 100**,4 250) =
= 4 100 IOPS

Ahora podemos calcular el coste de cada organización:

4 cadenas de discos de 4-GB = 50 000$ + 4 · 2 500$ + 25 · (4 096 · 3$) =
= 367 200$

5 cadenas de discos de 4-GB = 50 000$ + 5 · 2 500 + 25 · (4 096 · 3$) =
= 369 700$

15 cadenas de discos de 1-GB
= 50 000$ + 15 · 2 500$ + 100 · (1 024 · 3$) =
= 394 700$

17 cadenas de discos de 1-GB
= 50 000$ + 17 · 2 500$ + 100 · (1 024 · 3$) =
= 399 700$

Finalmente, el coste por IOPS para cada una de las cuatro configuraciones es 367 dólares, 361 dólares, 105 dólares y 97 dólares, respectivamente. Calculando el máximo número de E/S promedio por segundo suponiendo un 100 por 100 de utilización de los recursos críticos, el mejor coste/rendimiento es la organización con discos pequeños y mayor número de controladores. Los discos pequeños tienen de 3,4 a 3,8 veces mejor coste/rendimiento que los discos grandes en este ejemplo. El único inconveniente es que el mayor número de discos afectará a la disponibilidad del sistema a menos que se añada algo de redundancia (ver págs. 560-561).

Este ejemplo anterior suponía que los recursos podían utilizarse al 100 por 100. Es instructivo ver cuál es el cuello de botella de cada organización.

Ejemplo

Para las organizaciones del último ejemplo, calcular el porcentaje de utilización de cada recurso del sistema de computadores.

Respuesta

La Figura 9.31 da la respuesta.

Recurso	Discos de 4-GB, 4 cadenas	Discos de 4-GB, 5 cadenas	Discos de 1-GB, 15 cadenas	Discos de 1-GB, 17 cadenas
CPU	20%	21%	75%	82%
Memoria	20%	21%	75%	82%
Bus E/S	10%	10%	38%	41%
Buses SCSI	100%	82%	100%	96%
Discos	98%	100%	91%	100%

FIGURA 9.31 Porcentaje de utilización de cada recurso dadas las cuatro organizaciones del ejemplo anterior. Bien los buses SCSI o los discos son el cuello de botella.

En realidad, los buses no pueden aproximarse al 100 por 100 de la anchura de banda sin severos incrementos en la latencia y reducción en el rendimiento debido a la contención. Se ha desarrollado una serie de reglas empíricas para guiar los diseños de E/S:

Ningún bus de E/S debe ser utilizado más del 75 al 80 por 100;

Ninguna cadena del disco debe ser utilizada más del 40 por 100;

Ningún brazo de disco debe estar buscando más del 60 por 100 del tiempo.

Ejemplo

Recalcular el rendimiento del ejemplo anterior utilizando estas reglas y mostrar la utilización de cada componente. ¿Hay otras organizaciones que sigan estas directrices y mejoren el rendimiento?

Respuesta

La Figura 9.31 muestra que el bus de E/S está lejos de las directrices sugeridas; por ello nos concentraremos en la utilización de la búsqueda y del bus SCSI. La utilización del tiempo de búsqueda por disco es

$$\frac{\text{Tiempo medio de búsqueda}}{\text{Tiempo entre E/S}} = \frac{12 \text{ ms}}{\frac{1}{41 \text{ IOPS}}} = \frac{12}{24} = 50\ \%$$

que es inferior a la regla empírica. El mayor impacto está en el bus SCSI:

$$\text{IOPS sugerido por cadena SCSI} = \frac{1}{4 \text{ ms}} \cdot 40\ \% = 100 \text{ IOPS}$$

Con estos datos podemos recalcular IOPS para cada organización:

$$\begin{aligned} \text{4 cadenas de discos de 4-GB} &= \text{Min}(5\,000, 5\,000, 7\,500, 1\,025, \mathbf{400}) = \\ &= 400 \text{ IOPS} \end{aligned}$$

5 cadenas de discos de 4-GB = Min(5 000,5 000,7 500,1 025,**500**) =
= 500 IOPS

15 cadenas de discos de 1-GB = Min(5 000,5 000,7 500,4 100,**1 500**) =
= 1 500 IOPS

17 cadenas de discos de 1-GB = Min(5 000,5 000,7 500,4 100,**1 700**) =
= 1 700 IOPS

Bajo estas hipótesis, los discos pequeños tienen entre 3,0 y 4,2 veces el rendimiento de los discos grandes.

Claramente, la anchura de banda de la cadena es ahora el cuello de botella. El número de discos por cadena que no excederían la directriz es

Número de discos por cadena SCSI en anchura de banda completa =

$$= \frac{100}{41} = 2{,}4 \text{ o } 2$$

y el número ideal de cadenas es

Número de cadenas SCSI para anchura de banda completa de discos de 4-GB =

$$= \frac{25}{2} = 12{,}5 \text{ o } 13$$

Número de cadenas SCSI para ancho de banda completa de discos de 1-GB =

$$= \frac{100}{2} = 50$$

Esta sugerencia es buena para discos de 4 GB, pero el bus de E/S está limitado a 20 controladores y cadenas SCSI para que llegue a ser el límite para discos de 1 GB:

13 cadenas de discos de 4-GB = Min(5 000,5 000,7 500,**1 025**,1 300) =
= 1 025 IOPS

20 cadenas de discos de 1-GB = Min(5 000,5 000,7 500,4 100,**2 000**) =
= 2 000 IOPS

Podemos ahora calcular el coste para cada organización:

13 cadenas de discos de 4-GB = 50 000$ + 13·2 500$ + 25·(4 096·3$) =
= 389 700$

20 cadenas de discos de 1-GB = 50 000$ + 20·2 500$ + 100·(1 024·3$) =
= 407 200$

En este caso, los discos pequeños cuestan el 5 por 100 más y tienen todavía, aproximadamente, el doble de rendimiento que los discos grandes. La utilización de cada recurso se muestra en la Figura 9.25. Esto muestra que siguiendo la regla empírica de utilización de la cadena, el 40 por 100 pone el límite del rendimiento en todos los casos menos en uno.

Recurso	Discos de 4-GB 4 cadenas	Discos de 4-GB 5 cadenas	Discos de 1-GB 15 cadenas	Discos de 1-GB 17 cadenas	Discos de 4-GB 13 cadenas	Discos de 1-GB 20 cadenas
CPU	8%	10%	30%	34%	21%	40%
Memoria	8%	10%	30%	34%	21%	40%
Bus E/S	5%	7%	20%	23%	14%	27%
Buses SCSI	40%	40%	40%	40%	32%	40%
Discos	39%	49%	37%	41%	100%	49%
Utilización de búsqueda	19%	24%	18%	20%	49%	24%
IOPS	400	500	1 500	1 700	1 025	2 000

FIGURA 9.32 Porcentaje de utilización de cada recurso dadas las seis organizaciones de este ejemplo, que intentan limitar la utilización de recursos clave a las reglas empíricas dadas anteriormente.

9.9 Juntando todo: el subsistema de almacenamiento IBM 3990

Si los arquitectos de computadores hubiesen de seleccionar la compañía líder en diseño de E/S, IBM ganaría con facilidad. Una buena política de los computadores de IBM es la de las aplicaciones comerciales, que se sabe que es intensiva. Aunque haya dispositivos gráficos y redes que se puedan conectar a un computador IBM, la reputación de IBM proviene del rendimiento de los discos. Este es el aspecto en el que nos concentraremos en esta sección.

La arquitectura de E/S del IBM 360/370 ha evolucionado durante un período de veinticinco años. Inicialmente, el sistema de E/S era de propósito general y no prestaba atención especial a ningún dispositivo en particular. Cuando quedó claro que los discos magnéticos eran los principales consumidores de las E/S, la IBM 360 se modificó para soportar E/S rápidas de disco. La filosofía dominante de IBM fue elegir latencia sobre productividad siempre que hubiese una diferencia. IBM casi nunca utiliza un gran buffer fuera de la CPU; su objetivo es establecer un camino claro desde memoria principal al dispositivo de E/S, de manera que cuando un dispositivo esté listo, nada pueda interponerse en el camino. Quizá IBM siguió un corolario a los citados

en la página 567: se puede comprar anchura de banda, pero se necesita buen diseño para la latencia. Como filosofía secundaria, la CPU se descarga lo máximo posible, para permitir que continúe los cálculos, mientras otros realizan las actividades de E/S deseadas.

El ejemplo para esta sección es la CPU de la IBM 3090 y el subsistema de almacenamiento 3990. La IBM 3090, modelos 3090/100 a 3090/600, puede contener de una a seis CPU. Esta máquina de un ciclo de reloj de 18,5 ns tiene una memoria entrelazada con 16 módulos que puede transferir ocho bytes en cada ciclo de reloj a cada uno de los dos (3090/100) o cuatro (3090/600) buses. Cada procesador 3090 tiene una cache con postescritura, asociativa por conjuntos con 4 módulos y la cache soporta accesos segmentados que necesitan dos ciclos. La velocidad de cada CPU es aproximadamente 30 MIPS IBM (ver pág. 83), dando como máximo 180 MIPS para la IBM 3090/600. La supervisión de las instalaciones de grandes computadores IBM sugieren una regla de aproximadamente 4 GB de memoria de disco por MIPS de potencia de CPU (ver Sección 9.12).

Es justo avisar que la terminología de IBM pueda no ser autoevidente, aunque las ideas no sean difíciles. Recordar que esta arquitectura de E/S ha evolucionado desde 1964. Aunque puede haber ideas que IBM no incluiría si comenzase de nuevo, pueden hacer que este esquema funcione y hacer que funcione bien.

Jerarquía de transferencia de datos del subsistema de E/S 3990 y jerarquía de control

El subsistema de E/S se divide en dos jerarquías:

1. Control. Esta jerarquía de controladores negocia un camino a través de un mazo de conexiones posibles entre la memoria y el dispositivo de E/S y controla la duración de la transferencia.
2. Datos. Esta jerarquía de conexiones es el camino sobre el cual fluyen los datos entre memoria y el dispositivo de E/S.

Después de ir sobre cada una de las jerarquías, seguimos la pista de una lectura de disco para ayudar a comprender la función de cada componente.

Por simplicidad, comenzamos explicando la jerarquía de la transferencia de datos, mostrada en la Figura 9.33. Esta figura muestra una sección de la jerarquía que contiene hasta 64 discos grandes de IBM; utilizando 64 de los discos IBM 3390 recientemente anunciados, ¡esta parte podría conectar aproximadamente un trillón de bytes de almacenamiento! Sin embargo, esta parte representa sólo un sexto de la capacidad de la CPU de la IBM 3090/600. Esta posibilidad de expansión desde un pequeño sistema de E/S a cientos de discos y terabytes de memoria es lo que da a los grandes computadores de IBM su reputación en el mundo de las E/S.

El miembro mejor conocido de la jerarquía de datos es el *canal.* El canal no es más que 50 cables que conectan dos niveles de la jerarquía de E/S. Sólo se utilizan 18 de los 50 cables para transferir datos (8 datos más 1 paridad en

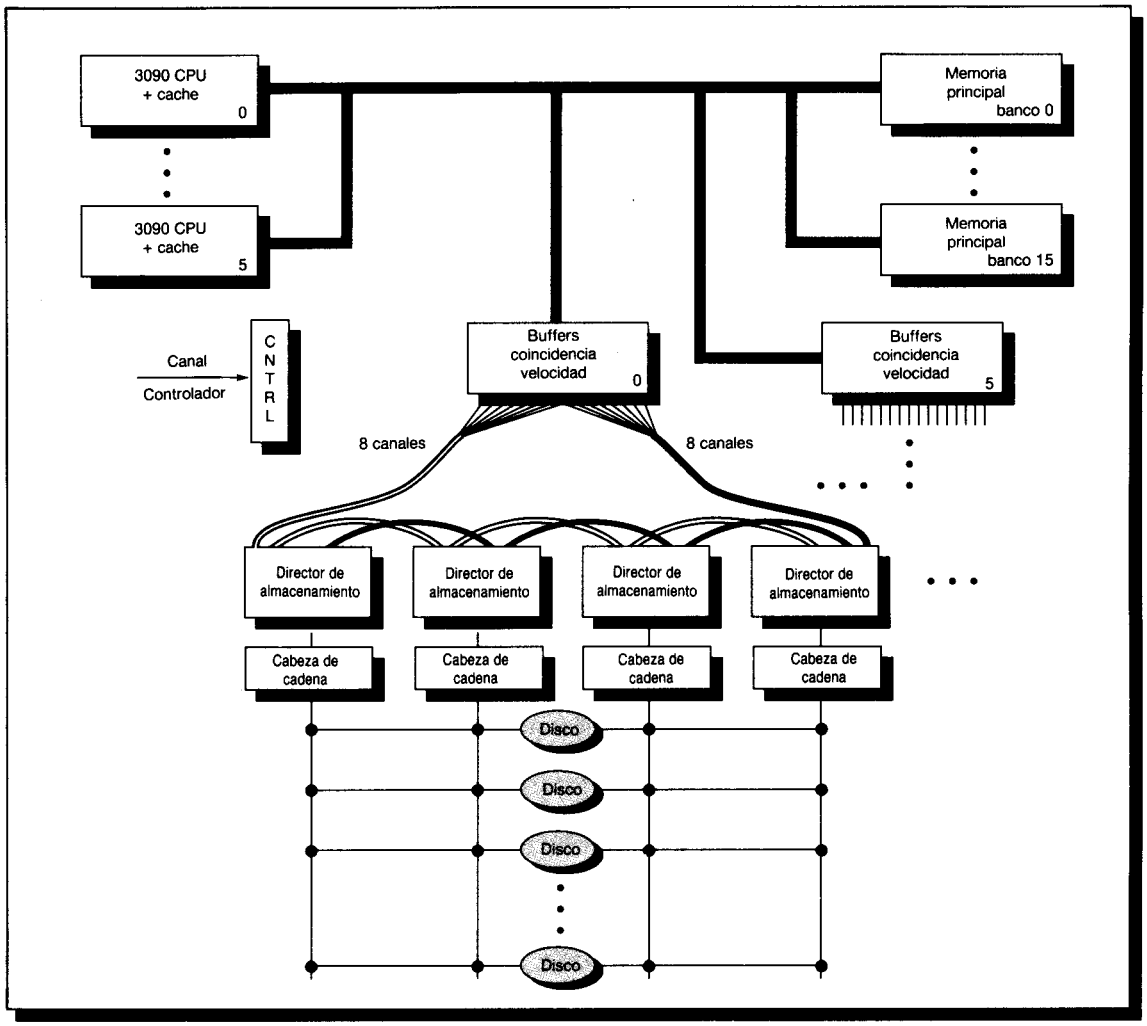


FIGURA 9.33 Jerarquía de la transferencia de datos en el subsistema de E/S IBM 3990. Observar que todos los canales están conectados a todos los directores de almacenamiento. Los discos en la parte inferior representan los controladores de discos de cuatro puertos IBM 3380, con un máximo de 64 discos. La colección de discos en el mismo camino a la cabeza del controlador de cadena se denomina una *cadena*.

cada dirección), mientras que el resto son para información de control. Durante años la máxima frecuencia de datos fue de 3 MB por segundo, pero recientemente se elevó a 4,5 MB por segundo. Se pueden conectar hasta 48 canales a una CPU 3090/100 y hasta 96 canales a una 3090/600. Debido a que están multiprogramados, los canales pueden dar un servicio real a varios discos. Por razones históricas, IBM llama a esto *multiplexación de bloques.*

Los canales se conectan a la memoria principal de la 3090 vía dos *buffers de adaptación de velocidades (speed-matching buffers),* que llevan todos los

canales de un solo puerto a memoria principal. Estos buffers sencillamente hacen coincidir la anchura de banda del dispositivo de E/S con la anchura de banda del sistema de memoria. Hay dos buffers de 8 bytes por canal.

El siguiente nivel bajando en la jerarquía de datos es el *director de almacenamiento (storage director).* Este es un dispositivo intermediario que permite que se comuniquen muchos canales con diferentes dispositivos de E/S. De cuatro a dieciséis canales van a través del director de almacenamiento dependiendo del modelo, y de dos a cuatro caminos salen de la parte inferior a los discos. Estos se denominan *cadenas de dos caminos (two-path strings)* o *cadenas de cuatro caminos (four-path strings)* en el lenguaje de IBM. Por tanto, cada director de almacenamiento puede comunicar con cualquiera de los discos utilizando una de las cadenas. En la parte superior de cada cadena está la *cabeza de cadena,* y todas las comunicaciones entre el disco y unidades de control deben pasar a través de ella.

En el extremo inferior de la jerarquía del camino de datos están los dispositivos de disco. Para incrementar la disponibilidad, los dispositivos de disco como el IBM 3380 proporcionan cuatro caminos para conectarse al director de almacenamiento; si falla un camino, todavía se puede conectar el dispositivo.

Los caminos redundantes desde memoria principal al dispositivo de E/S no sólo mejoran la disponibilidad, sino que también mejoran el rendimiento. Como la filosofía de IBM es evitar grandes buffers, el camino del dispositivo de E/S a memoria principal debe permanecer conectado hasta que se complete la transferencia. Si hubiese un simple camino jerárquico de los dispositivos al buffer de coincidencia de velocidades, solamente podría transferir cada vez un dispositivo de E/S en un subárbol. En lugar de ello, los múltiples caminos permiten que múltiples dispositivos transfieran simultáneamente a través del director de almacenamiento y en memoria.

La tarea de establecer la conexión del camino de datos es la de la jerarquía de control. La Figura 9.34 muestra ambas jerarquías de control y de datos del subsistema de E/S 3990. El nuevo dispositivo es el procesador de E/S. El controlador del canal 3090 y el procesador de E/S son máquinas de carga/almacenamiento similares a DLX, excepto que no hay jerarquía de memoria. En la siguiente sección veremos cómo funcionan juntas las jerarquías para leer un sector del disco.

Seguimiento de una lectura de disco en el subsistema de E/S IBM 3990

Los 12 pasos siguientes describen una lectura de un sector de un disco IBM 3380. Cada uno de los 12 pasos está etiquetado en un dibujo de la jerarquía completa en la Figura 9.34.

1. El usuario pone una estructura de datos en memoria que contiene las operaciones que deben ocurrir durante este evento de E/S. Esta estructura de datos se denomina *bloque de control de E/S* o IOCB, que también señala a una lista de palabras de control de canal (CCW). Esta lista se denomina *pro-*

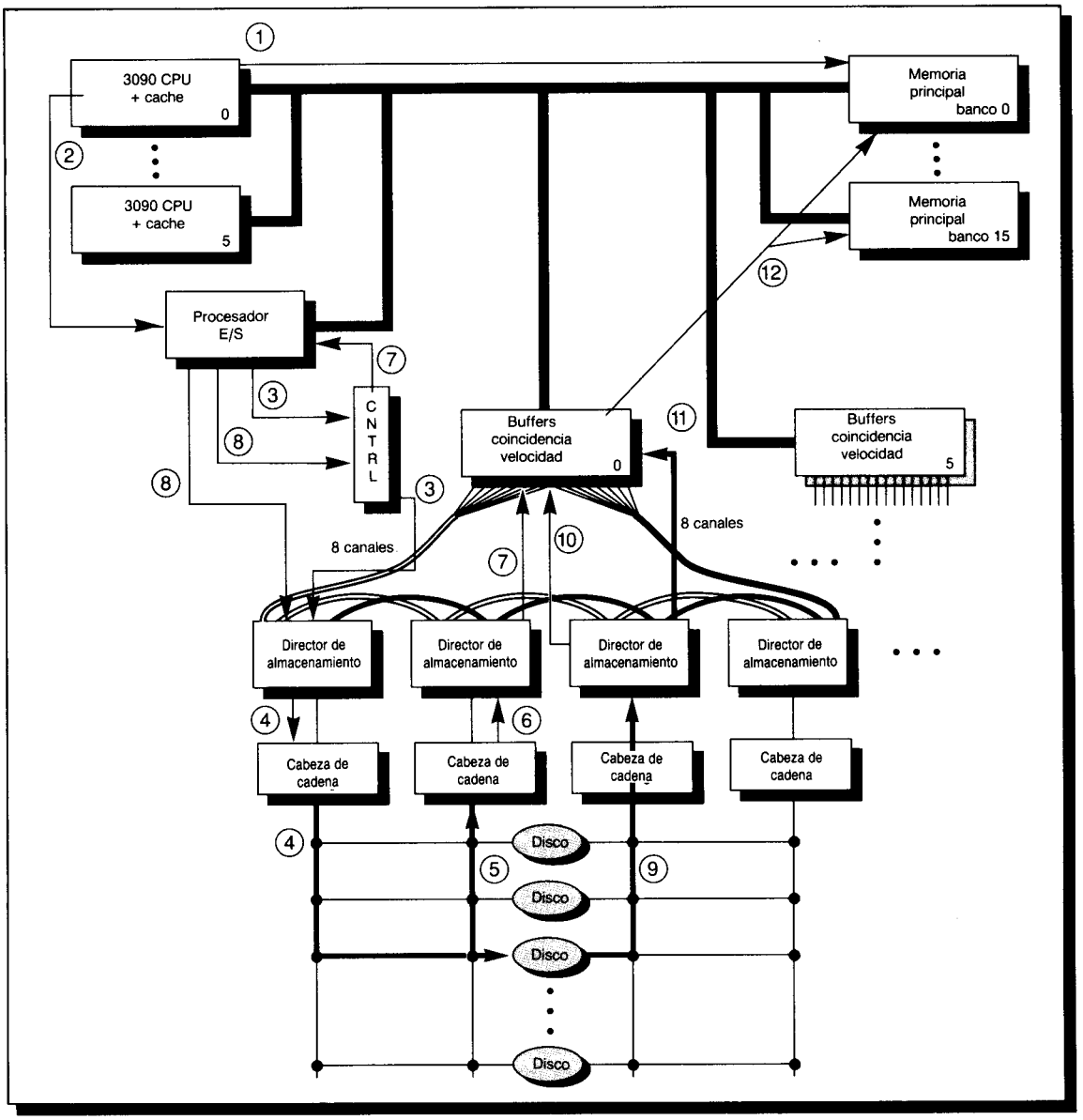


FIGURA 9.34 Jerarquías de control y datos en el subsistema de E/S IBM 3990 etiquetadas con los 12 pasos para leer un sector de un disco. La única nueva caja con respecto a la Figura 9.33 (pág. 591) es el procesador de E/S.

grama de canal. Normalmente, el sistema operativo proporciona el programa de canal, pero algunos usuarios escriben el suyo propio. El sistema operativo comprueba el IOCB para protección de violaciones antes de que la E/S pueda continuar.

Posición	CCW	Comentario
CCW1:	Define Extent	Transfiere un parámetro de 16 bytes al director de almacenamiento. El canal ve esto como una transferencia de dato de escritura.
CCW2:	Locate Record	Transfiere un parámetro de 16 bytes al director de almacenamiento como antes. El parámetro identifica la operación (lectura en este caso) más búsqueda, número de sector, y registro ID. El canal de nuevo ve esto como una transferencia de dato de escritura.
CCW3:	Read Data	Transfiere el dato del disco deseado al canal y después a memoria principal

FIGURA 9.35 Un programa de canal para realizar una lectura de disco consta de tres palabras de órdenes de canal (CCWs). El sistema operativo comprueba las violaciones de los accesos a la memoria virtual de las CCWs simulándolos para comprobar las violaciones. Estas instrucciones están enlazadas para que solamente se necesite una instrucción START SUBCHANNEL.

2. La CPU ejecuta una instrucción START SUBCHANNEL. La petición actual está definida en el programa de canal. Un programa de canal que lea un registro puede ser como el de la Figura 9.35.

3. El procesador de E/S utiliza las conexiones de control de uno de los canales para indicar al gestor de memoria el disco que va a ser accedido y la dirección del disco a la que se va a leer. El canal se libera entonces.

4. El gestor de almacenamiento envía una orden SEEK (BUSQUEDA) al controlador de la cabeza de la cadena y éste conecta con el disco deseado, indicándole que busque la pista apropiada y después desconecta. La desconexión ocurre entre CCW2 y CCW3 en la Figura 9.35.

Después de completarse estos cuatro primeros pasos de la lectura, el brazo del disco busca la pista correcta en la unidad correcta de disco IBM 3380. Otras operaciones de E/S pueden utilizar la jerarquía de control y datos mientras este disco está buscando y el dato está girando bajo la cabeza de lectura. El procesador de E/S entonces actúa como un sistema multiprogramado, trabajando en otras peticiones mientras espera completar un evento de E/S.

Surge una pregunta interesante: Cuando hay múltiples usos para un solo disco, ¿qué previene de que otra búsqueda estropee el trabajo, antes que la petición original pueda continuar con el evento de E/S en progreso? La respuesta es que el disco aparece ocupado para los programas en el 3090 en el intervalo de tiempo desde que una instrucción de START SUBCHANNEL comienza un programa de canal (paso 2) hasta que finaliza ese programa de canal. Un intento de ejecutar otra instrucción START SUBCHANNEL recibirá estatus de ocupado desde el canal o desde el dispositivo del disco.

Después que la búsqueda se completa y el disco gira al punto deseado relativo a la cabeza de lectura, el disco se reconecta a un canal. Para determinar la posición de giro del disco 3380, IBM proporciona un sensor de posición de

rotación (RPS), una característica que da avisos previos cuando el dato gire bajo la cabeza de lectura. IBM, esencialmente, extiende el tiempo de búsqueda para incluir algo del tiempo de rotación, intentando así que el camino de datos sea lo más pequeño posible. Entonces la E/S puede continuar:

5. Cuando el disco completa la búsqueda y gira a la posición correcta, contacta el controlador de la cabeza de la cadena.

6. El controlador de la cabeza de la cadena busca un gestor de almacenamiento libre para enviar la señal de que el disco está en la pista correcta.

7. El gestor de almacenamiento busca un canal libre de forma que pueda utilizar las conexiones físicas de control para indicar al procesador de E/S que el disco está en la pista correcta.

8. El procesador de E/S simultáneamente contacta el gestor de almacenamiento con el dispositivo de E/S (el disco IBM 3380) para dar la conformidad para transferir datos, e indicar al controlador del canal dónde poner la información en memoria principal cuando llegue al canal.

Ahora hay un camino directo entre el dispositivo de E/S y memoria y puede comenzar la transferencia:

9. Cuando el disco está preparado para transferir, envía el dato a 3 megabytes por segundo sobre una línea serie de bits al gestor de almacenamiento.

10. El gestor de almacenamiento recoge 16 bytes en uno de los dos buffers y envía la información al controlador del canal.

11. El controlador del canal tiene un par de buffers de 16 bytes por gestor de almacenamiento y envía 16 bytes entre 3 MB o 4,5 MB por segundo por un camino de datos de 8 bits de ancho a los buffers de adaptación de velocidades.

12. Los buffers de adaptación de velocidades toman la información que proviene de todos los canales. Hay dos buffers de 8 bytes por canal que cada vez envían 8 bytes a las posiciones apropiadas de memoria principal.

Como no hay nada gratis en el diseño de computadores, se puede esperar que haya un coste al anticipar el retardo de rotación utilizando RPS. A veces un camino libre no puede establecerse en el tiempo disponible debido a otra actividad de E/S, dando como consecuencia una *falta de RPS*. Una falta de RPS significa que el subsistema de E/S 3990 debe:

- Esperar otra rotación completa —16,7 ms— antes de que el dato vuelva a estar bajo la cabeza, o
- ¡Romper el camino jerárquico de datos y comenzar todo de nuevo!

Muchas faltas de RPS pueden arruinar los tiempos de respuesta.

Como se mencionó antes, el sistema de E/S de IBM evolucionó durante muchos años y la Figura 9.36 muestra el cambio en el tiempo de respuesta para algunos de esos cambios. La primera mejora concierne al camino para los datos después de su reconocimiento. Antes del Sistema/370-XA, el camino de datos a través de los canales y gestor de almacenamiento (pasos 5 a

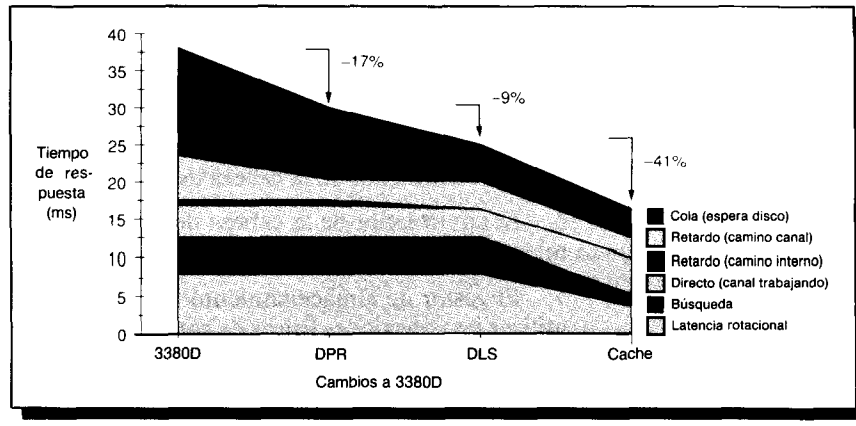


FIGURA 9.36 Cambios en el tiempo de respuesta con mejoras en 3380D descompuesto en seis categorías [Friesenborg y Wicks, 1985]. Los retardos de colas se refieren al tiempo cuando el programa espera a que otro programa termine en el disco. El retardo del camino de canal es el tiempo que la operación espera debido a que el camino de canal y el director de almacenamiento están ocupados con otra tarea. El retardo del camino interno es similar al retardo del camino de canal excepto que se refiere a caminos internos en el 3380D. Directos significa el tiempo que el camino de canal está ocupado con la operación. El tiempo de búsqueda y la latencia rotacional son las definiciones estándares. Robinson y Blount [1986] informan en el estudio del 3880-23 que la frecuencia de aciertos de lecturas para la cache de 32 MB de escritura directa en algunos sistemas grandes promedia aproximadamente al 90 por 100, contabilizando las lecturas el 92 por 100 de los accesos del disco.

12) tenía que ser el mismo que el tomado para pedir la búsqueda (pasos 1 a 4). La 370-XA permite que el camino después del reconocimiento sea diferente y esta opción se llama *reconocimiento dinámico de caminos* (DPR). Este cambio reducía el tiempo de espera para el camino del canal y el tiempo de espera para los discos (retardo de cola), obteniéndose una reducción en el tiempo medio de respuesta total del 17 por 100. El segundo cambio en la Figura 9.36 involucra un nuevo diseño de disco. Mejoras en el control del microcódigo del 3380D consiguen ligeras mejoras en el tiempo de búsqueda además de eliminar una restricción de que los brazos del disco que estén en el mismo camino interno se les evite de operar al mismo tiempo. IBM denomina a esta opción *Selección de Nivel de Dispositivo —Device Level Select—* (DLS). Este cambio reducía los retardos internos del camino a 0. Esto tuvo poco impacto ya que no había mucho tiempo de espera en los retardos internos porque los clientes intencionadamente colocaban los datos en los discos intentando evitar retardos de caminos internos. Este segundo cambio redujo el tiempo de respuesta otro 9 por 100. El cambio final fue añadir una cache de disco de escritura directa de 32 MB a un 3380D, denominado IBM 3880-23. La cache del disco redujo la latencia media de rotación, tiempo de búsqueda y retardos de la cola, dando otra reducción del 44 por 100 en el tiempo de respuesta.

Una indicación de la efectividad de la DPR es el número de dispositivos de disco conectados a una cadena. Estudios de los sistemas IBM que utilizan DPR, que promedian 16 dispositivos de disco por cadena frente a 12 sin DPR, sugieren que la reconexión dinámica permite una mayor velocidad de E/S con tiempo de respuesta comparable [Henly y McNutt, 1989].

Resumen del subsistema de E/S IBM 3990

Los objetivos de los sistemas de E/S son los siguientes:

- Bajo coste
- Diversidad de tipos de dispositivos de E/S
- Gran número de dispositivos de E/S a la vez
- Alto rendimiento
- Baja latencia

La expansibilidad sustancial y menor latencia son difíciles de obtener al mismo tiempo. Los sistemas IBM basados en canal consiguen el tercer y cuarto objetivo al utilizar caminos jerárquicos de datos para conectar un gran número de dispositivos. Muchos dispositivos y caminos paralelos permiten transferencias simultáneas y, por tanto, alta productividad. Al evitar grandes buffers y proporcionar suficientes caminos extra para minimizar el retardo de congestión, los canales ofrecen también E/S de baja latencia. Al maximizar el uso de la jerarquía, IBM utiliza la percepción de la posición de rotación para alargar el tiempo que otras tareas pueden utilizar la jerarquía durante una operación de E/S.

Además, una clave para el rendimiento del subsistema de E/S de IBM es el número de faltas de posición de rotación y la congestión en los caminos del canal. Una regla empírica es que los canales de un solo camino no deberían utilizarse más del 30 por 100 y los canales de cuádruples caminos no deberían utilizarse más del 60 por 100, o se producirán muchas faltas de posición de rotación. Esta arquitectura de E/S domina la industria, pero con todo sería interesante ver qué haría IBM en el caso que partiera de cero en el diseño de E/S.

9.10 Falacias y pifias

Falacia: Las E/S juegan un pequeño papel en el diseño de los supercomputadores.

El objetivo de la Illiac IV fue ser el computador más rápido del mundo. Puede que no lograse ese objetivo, pero mostró las E/S como el talón de Aquiles de las máquinas de alto rendimiento. En algunas tareas, se gastaba más tiempo

en cargar datos que en realizar cálculos. La Ley de Amdahl demostraba la importancia del alto rendimiento en todas las partes de un computador de alta velocidad. (En efecto, Amdahl hizo este comentario en reacción a las afirmaciones sobre el rendimiento a través del paralelismo, hechas en nombre de la Illiac IV.) La Illiac IV tenía una velocidad de transferencia muy rápida (60 MB/s), pero discos muy pequeños de cabeza fija (capacidad 12 MB). Como no eran suficientemente grandes, se suministró más memoria en un computador separado. Esto condujo a dos formas de medir el gasto de E/S:

Arranque caliente. Suponiendo que el dato está en los pequeños discos más rápidos, el gasto de E/S es el tiempo en cargar la memoria de la Illiac IV desde los discos.

Arranque frío. Suponiendo que el dato está en el otro computador, los gastos de E/S deben incluir el tiempo para transferir primero los datos a los discos rápidos de la Illiac IV.

La Figura 9.37 muestra diez aplicaciones escritas para la Illiac IV en 1979. Suponiendo arranques calientes, el supercomputador estaba ocupado el 78 por 100 del tiempo y esperando el 22 por 100 del tiempo para las E/S; suponiendo arranques fríos, estaba ocupado el 59 por 100 del tiempo y esperando el 41 por 100 del tiempo para las E/S.

Pifia: Desplazar funciones de la CPU al procesador de E/S para mejorar el rendimiento.

Hay muchos ejemplos de esta pifia, aunque los procesadores de E/S pueden realzar el rendimiento. Un problema inherente con una familia de computadores es que la migración de una característica de E/S habitualmente cambia la arquitectura del repertorio de instrucciones o la arquitectura del sistema de una forma visible al programador, haciendo que todas las máquinas futuras tengan que vivir con una decisión que se tomó en el pasado. Si las CPU se mejoran en coste/rendimiento más rápidamente que el procesador de E/S (y este probablemente es el caso) entonces desplazar la función puede dar como consecuencia una máquina más lenta en la siguiente CPU.

El ejemplo más notable proviene de la IBM 360. Se decidió que el rendimiento del sistema ISAM, un primitivo sistema de base de datos, mejoraría si algunas búsquedas de registros se realizasen en el mismo controlador del disco. Se asoció un campo de clave a cada registro y el dispositivo buscaba cada clave cuando el disco giraba hasta que encontraba una coincidencia. Entonces transferiría el registro deseado. Para que el disco encontrase la clave, tenía que haber un espacio vacío extra en la pista. Este esquema es aplicable a búsquedas mediante índices, así como a datos.

La velocidad a la que se puede buscar una pista está limitada por la velocidad del disco y por el número de claves que se pueden empaquetar en una pista. En un disco IBM 3330 la clave, normalmente, es de 10 caracteres, pero el espacio vacío total entre registros es equivalente a 191 caracteres si hubiese una clave. (El espacio es de sólo 135 caracteres si no hubiese clave, ya que no

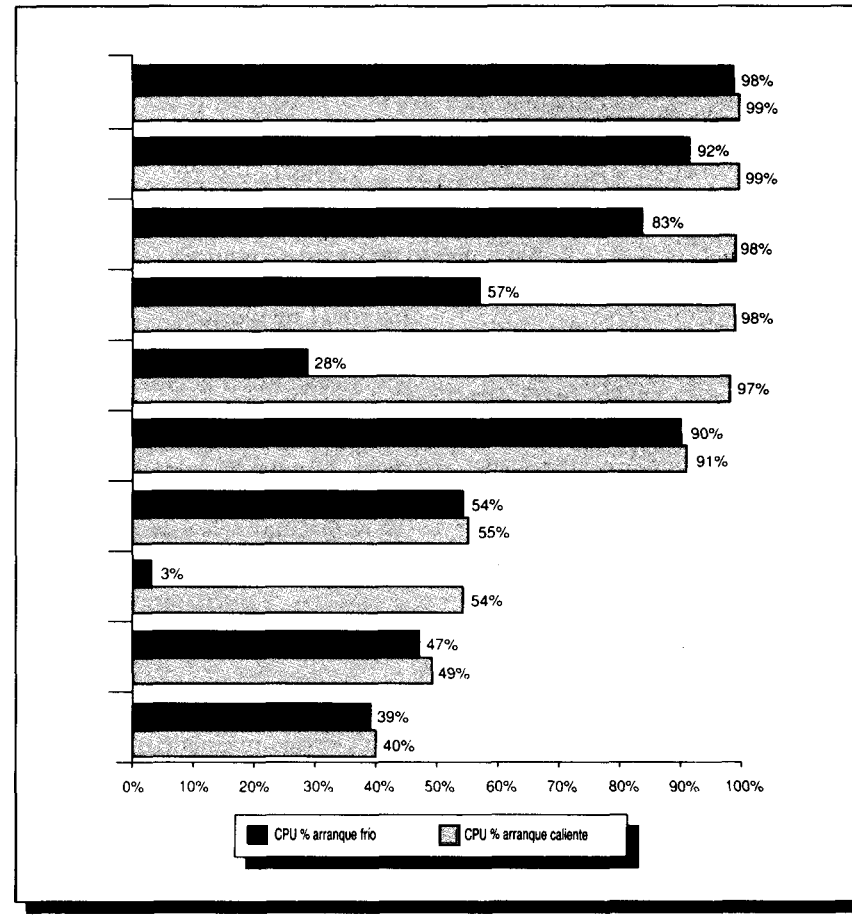


FIGURA 9.37 Feierback y Stevenson [1979] resumieron las aplicaciones importantes de Illiac IV y el porcentaje de tiempo empleado en el cálculo frente a las esperas de E/S. La media aritmética de 10 programas son 78 por 100 de cálculo para arranque caliente y el 59 por 100 de cálculo para arranque frío.

hay necesidad de espacio extra para la clave.) Si suponemos que el dato también tiene 10 caracteres y que la pista no tiene nada sobre ella, entonces una pista de 13 165 bytes puede contener

$$\frac{13\,165}{191 + 10 + 10} = 62 \text{ registros de claves y datos}$$

Este rendimiento es

$$\frac{16{,}7 \text{ ms (1 revolución)}}{62} \approx 0{,}25 \text{ ms/búsqueda clave}$$

En lugar de este esquema, podríamos poner varias parejas de datos-clave en un solo bloque y tener espacios más pequeños interregistros. Suponiendo que hay 15 pares de datos clave por bloque y que la pista no tiene nada en ella, entonces

$$\frac{13\,165}{135 + 15 \cdot (10 + 10)} = \frac{13\,165}{135 + 300} = 30 \textbf{ bloques} \text{ de pares de datos clave}$$

El rendimiento revisado es entonces

$$\frac{16{,}7 \text{ ms (1 revolución)}}{30 \cdot 15} \approx 0{,}04 \text{ ms/búsqueda clave}$$

Sin embargo, como las CPU son más rápidas, el tiempo de CPU para una búsqueda era trivial. Aunque la estrategia hizo a las primeras máquinas más rápidas, ¡los programas que utilizan la operación búsqueda de claves en el procesador de E/S se ejecutan seis veces con más lentitud en las máquinas actuales!

Falacia: Comparar el precio de las partes con el precio del sistema empaquetado.

Esto ocurre con mucha frecuencia cuando las nuevas tecnologías de memoria se comparan con los discos magnéticos. Por ejemplo, comparar el precio de un chip DRAM con el precio de un disco magnético empaquetado en la Figura 9.16 sugiere que la diferencia es menor que un factor de 10, pero es mucho mayor cuando se incluye el precio de la DRAM. Un error común con los medios desmontables es comparar el coste del medio sin incluir el dispositivo para leer el medio. Por ejemplo, el medio óptico costaba sólo 1 dólar por MB en 1990, pero la inclusión del coste de la unidad óptica puede acercar el precio a 6 dólares por MB.

Falacia: El tiempo de una búsqueda media en un disco en un sistema de computadores es el tiempo para una búsqueda de un tercio del número de cilindros.

Esta falacia proviene de confundir la forma que los fabricantes venden discos con el rendimiento esperado y con la falsa hipótesis de que los tiempos de búsqueda son lineales con la distancia. La regla empírica de un tercio de la distancia proviene de calcular la distancia de una búsqueda desde una posición aleatoria a otra posición aleatoria, sin incluir el cilindro actual y suponiendo que hay un gran número de ciclindros. En el pasado, los fabricantes listaban la búsqueda de esta distancia para ofrecer una base consistente para comparaciones. (Como mencionamos en la pág. 555, hoy día calculan el «promedio» sumando todos los tiempos de búsqueda y dividiendo por el número.) Suponiendo (incorrectamente) que el tiempo de búsqueda es lineal con la distancia y usando el mínimo indicado por los fabricantes y los tiempos «me-

dios» de búsqueda, una técnica común para predecir el tiempo de búsqueda es:

$$\text{Tiempo}_{\text{búsqueda}} = \text{Tiempo}_{\text{mínimo}} + \frac{\text{Distancia}}{\text{Distancia}_{\text{promedio}}} \cdot$$

$$\cdot (\text{Tiempo}_{\text{promedio}} - \text{Tiempo}_{\text{mínimo}})$$

La falacia relativa al tiempo de búsqueda es doble. Primero, el tiempo de búsqueda **no** es lineal con la distancia; el brazo debe acelerar para vencer la inercia, conseguir su máxima velocidad de crucero, desacelerar cuando logra la posición requerida y después esperar para permitir que el brazo se detenga vibrando (tiempo de asentamiento —*settle time*—). Sin embargo, en discos recientes a veces el brazo debe detenerse brevemente para controlar las vibraciones. La Figura 9.38 dibuja el tiempo frente a la distancia de búsqueda para un disco de ejemplo. También muestra el error en la fórmula anterior del

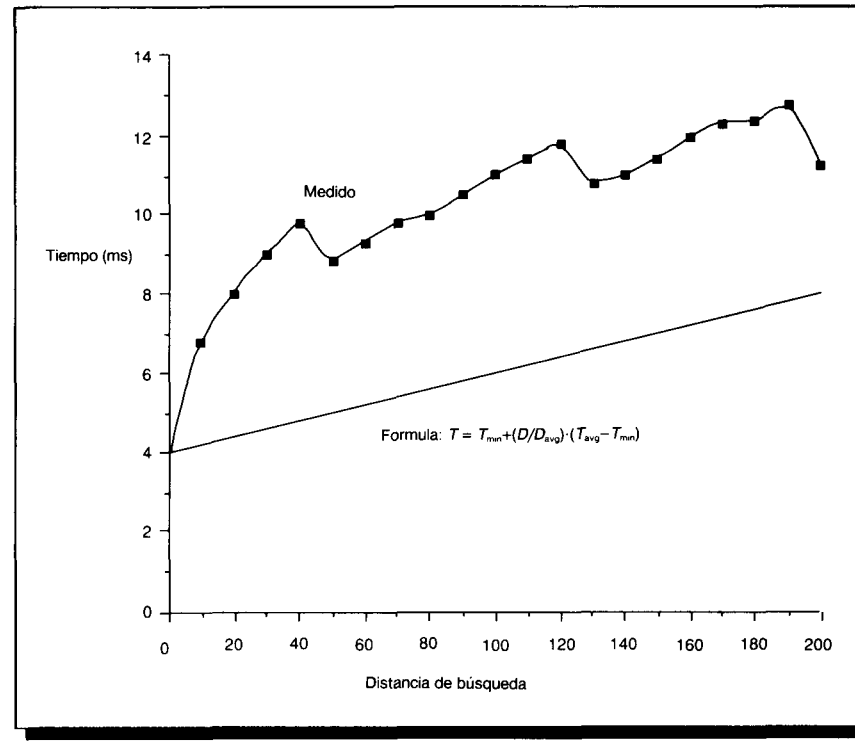


FIGURA 9.38 Tiempo de búsqueda frente a distancia de búsqueda para los 200 primeros cilindros. El Imprimis Sabre 97209 contiene 1.2 GB que utiliza 1.635 cilindros y tiene la interfaz IPI-2 [Imprimis, 1989]. Este es un disco de 8 pulgadas. Observar que búsquedas mayores pueden emplear **menos** tiempo que búsquedas más cortas. Por ejemplo, una búsqueda de 40 cilindros emplea casi 10 ms, mientras que una búsqueda de 50 cilindros emplea menos de 9 ms.

IBM 3380D Rango para fórmula		Fórmulas	IBM 3380J Rango para fórmula		Fórmulas
⩾	⩽		⩾	⩽	
1	50	$1{,}9 + \sqrt{\text{Distancia}} - \frac{\text{Distancia}}{50}$	1	50	$2{,}48 + \sqrt{\text{Distancia}} - \frac{\text{Distancia}}{20}$
51	100	$8{,}1 + 0{,}044 \cdot (\text{Distancia} - 50)$	51	130	$7{,}28 + 0{,}0320 \cdot (\text{Distancia} - 50)$
101	500	$10{,}3 + 0{,}025 \cdot (\text{Distancia} - 100)$	131	500	$10{,}08 + 0{,}0166 \cdot (\text{Distancia} - 130)$
501	884	$20{,}4 + 0{,}017 \cdot (\text{Distancia} - 500)$	501	884	$16{,}00 + 0{,}0114 \cdot (\text{Distancia} - 500)$

FIGURA 9.39 Fórmulas para el tiempo de búsqueda en ms para dos discos IBM. Thisquen [1988] midió estos discos y propuso estas fórmulas para modelarlas. Las dos columnas de la izquierda muestran el rango de las distancias de búsqueda en los cilindros para los que se aplica cada fórmula. Cada disco tiene 885 cilindros; asi, que la búsqueda máxima es 884.

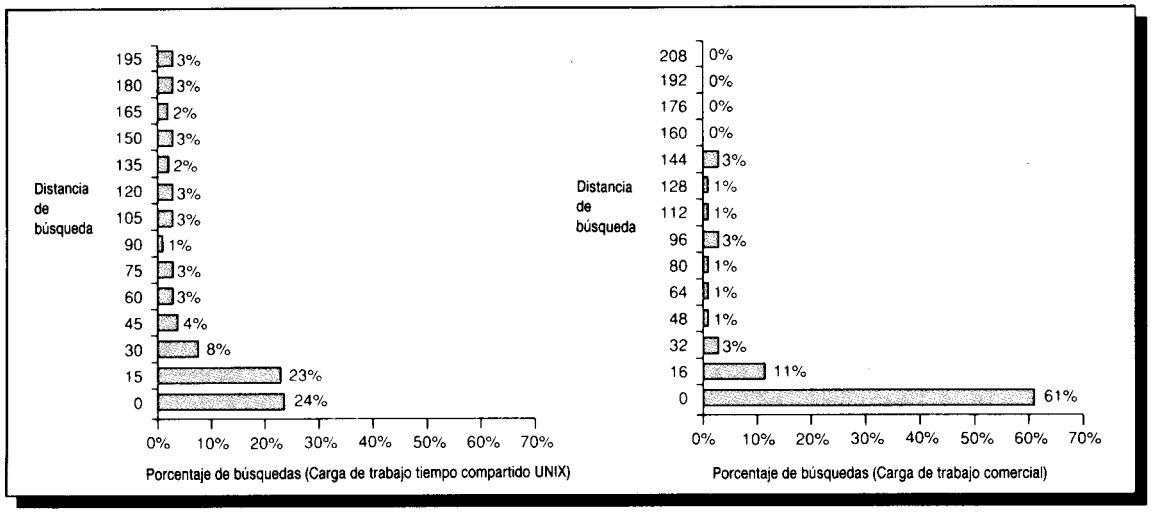


FIGURA 9.40 Medidas ejemplo de distancia de búsqueda para dos sistemas. Las medidas de la izquierda se tomaron en un sistema UNIX de tiempo compartido. Las medidas de la derecha se tomaron de una aplicación de procesamiento comercial en las cuales la actividad de búsqueda del disco estaba planificada. La distancia de búsqueda 0 significa que el acceso se realiza en el mismo cilindro. El resto de los números muestra el porcentaje colectivo para distancias crecientes entre números en el eje y. Por ejemplo, el 11 por 100 para la barra etiquetada 16 en la gráfica comercial significa que el porcentaje de búsquedas entre 1 y 16 cilindros fue del 11 por 100. Las medidas UNIX se pararon en 200 cilindros, pero éste capturó el 85 por 100 de los accesos. El total fue 1 000 cilindros. Las medidas comerciales siguieron todos los 816 cilindros de los discos. Las únicas distancias de búsquedas con 1 por 100 o más de las búsquedas que no están en el gráfico son 224, con el 4 por 100 y 304, 336, 512 y 624 cada una teniendo el 1 por 100. Este total es 94 por 100, siendo la diferencia pequeña pero de distancias no cero en otras categorías. Las medidas son cortesía de Dave Anderson, de Imprimis.

tiempo de búsqueda. Para búsquedas cortas, la fase de aceleración juega un papel mayor que la velocidad máxima de crucero y esta fase normalmente se modela como la raíz cuadrada de la distancia. La Figura 9.39 muestra fórmulas precisas utilizadas para modelar el tiempo de búsqueda frente a distancia para dos discos.

El segundo problema es que el promedio de la especificación del producto sólo sería cierto si no hubiese localidad para la actividad del disco. Afortunadamente, hay localidad temporal y espacial (pág. 433 en el Cap. 8): los bloques del disco usados más de una vez y los bloques del disco próximos al cilindro actual tienen más probabilidad de ser utilizados que los que están más lejos. Por ejemplo, la Figura 9.40 muestra medidas ejemplo de distancias de búsqueda para dos cargas de trabajo: una carga de trabajo de tiempo compartido UNIX y otra de procesamiento de oficinas. Observar el alto porcentaje de accesos del disco al mismo cilindro, etiquetado distancia 0 en los gráficos, en ambas cargas de trabajo.

Por tanto, esta falacia no puede ser más engañosa. Los Ejercicios desenmascaran esta falacia con más detalle.

9.11 Observaciones finales

Los sistemas de E/S se juzgan por la variedad de dispositivos de E/S, número máximo de dispositivos de E/S, coste y rendimiento, medidos en latencia y productividad. Estos objetivos comunes conducen a esquemas que varían ampliamente, algunos confiando extensivamente en el «buffering» y otros evitando el «buffering» a toda costa. Si uno es claramente mejor que otro, no es obvio hoy día. Quizá esta situación es como el debate del repertorio de instrucciones de los años ochenta, y la validez o invalidez de las alternativas se ha aclarado en los años noventa.

De acuerdo con la Ley de Amdahl, la ignorancia de las E/S conducirá a un rendimiento desaprovechado cuando las CPU consiguieran ser más rápidas. El rendimiento de los discos está creciendo a un ritmo del 4 al 6 por 100 por año, mientras que las CPU están creciendo a una velocidad mucho más rápida. Las demandas futuras para las E/S incluyen mejores algoritmos, mejores organizaciones y más capturas *(caching)* en un esfuerzo por evitar conflictos.

9.12 Perspectiva histórica y referencias

El precursor de las estaciones de trabajo actuales fue el «Alto» desarrollado por Xerox Palo Alto Research Center en 1974 [Thacker y cols., 1982]. Esta máquina invirtió la tradicional prudencia, haciendo que la interpretación del repertorio de instrucciones tuviese sitio en la pantalla: la pantalla utilizaba la mitad de la anchura de banda de memoria de Alto. Además del mapa de bits de la pantalla, esta máquina histórica tuvo la primera Ethernet [Metcalfe y Boggs, 1976] y la primera impresora láser. También tenía un ratón, inventado

anteriormente por Doug Engelbart de SRI, y un disco de cartucho móvil. La CPU de 16 bits implementaba un repertorio de instrucciones similar al de Nova de Data General y ofrecía memoria de control escribible (ver Cap. 5, Sección 5.8). En efecto, una simple máquina programable conducía la pantalla de gráficos, ratón, discos, red, y, cuando no tenía nada que hacer, interpretaba el repertorio de instrucciones.

La atracción de un computador personal es que no hay que compartirlo con nadie. Ello significa que el tiempo de respuesta es predecible, de forma distinta a los sistemas de tiempo compartido. Los primeros experimentos sobre la importancia de un tiempo de respuesta rápido lo realizaron Doherty y Kelisky [1979]. Mostraron que si el tiempo de respuesta de un sistema de computadores aumentaba en un segundo, también aumentaba un segundo el tiempo de asimilación del usuario. Thadhani [1981] mostró un salto en la productividad cuando los tiempos de respuesta de los computadores cayeron a un segundo y otro salto cuando cayeron a medio segundo. Sus resultados inspiraron una multitud de estudios que soportaron sus observaciones [IBM, 1982]. ¡De hecho, algunos estudios comenzaron para refutar sus resultados! Brady [1986] propuso diferenciar tiempo de entrada de tiempo de asimilación (ya que el tiempo de entrada se estaba haciendo significativo cuando los dos se agruparon juntos) y proporcionó un modelo cognoscitivo para explicar la más que lineal relación entre el tiempo de respuesta del computador y el tiempo de asimilación del usuario.

La ubicuidad del microprocesador no sólo ha inspirado los computadores personales de los años setenta, sino la tendencia actual a desplazar las funciones del controlador a los dispositivos de E/S a finales de los años ochenta y en los noventa. Por ejemplo, las rutinas microcodificadas en una CPU central tenían sentido para el Alto en 1975, pero los cambios tecnológicos hicieron pronto económicos los dispositivos de E/S del controlador microprogramable. Estos se sustituyeron entonces por circuitos integrados de aplicación específica. Los dispositivos de E/S continuaban esta tendencia llevando los controladores a los mismos dispositivos. Estos se denominaron *dispositivos inteligentes* y algunos buses estándares (por ejemplo, IPI y SCSI) se han creado justo para estos dispositivos. Los dispositivos inteligentes pueden relajar las ligaduras de tiempo manipulando muchas de las tareas de bajo nivel y colocando los resultados en la cola. Por ejemplo, muchas unidades de discos SCSI compatibles incluyen un buffer de pista en el mismo disco, soportando lectura anticipada y conexión/desconexión. Por ello, en una cadena SCSI algunos discos pueden estar buscando y otros cargando su buffer de pista aunque uno esté transfiriendo datos de su buffer sobre el bus SCSI.

Hablando de buses, el primer bus multivendedor puede haber sido el Unibus del PDP-11 en 1970. DEC animó a otras compañías a construir dispositivos que pudiesen conectarse a su bus, y muchas compañías lo hicieron. Un ejemplo más reciente es la SCSI, que son las siglas de *small computer systems interface (interfaz de pequeños sistemas de computadores).* Este bus, originalmente denominado SASI, fue inventado por Shugart y más tarde fue estandarizado por el IEEE. Algunos buses se desarrollaron en la academia: el *NuBus* lo desarrollaron Steve Ward y cols. en el MIT y lo utilizaron varias compañías. Afortunadamente, esta política de puertas abiertas sobre los buses contrasta con las compañías propietarias de buses que utilizan interfaces paten-

tadas, evitando así la competencia de los vendedores de conexiones compatibles. Esta práctica también eleva los costes y baja la disponibilidad de los dispositivos de E/S que se conectan en los buses propietarios, ya que estos dispositivos deben tener una interfaz diseñada para ese bus. Levy [1978] tiene un agradable artículo de visión general sobre buses.

También debemos dar algunas referencias para dispositivos específicos de E/S. Los lectores interesados en ARPANET deberían ver Kahn [1972]. Como mencionamos en una de las citas de la sección, el padre de los gráficos de computadores es Ivan Sutherland, que recibió el «ACM Turing Award» en 1988. El sistema Sketchpad de Sutherland [1963] puso el estándar para las interfaces y pantallas de hoy día. Ver Foley y Van Dam [1982] y Newman y Sproull [1979] para ampliar sobre gráficos de computadores. Scranton, Thompson y Hunter [1983] fueron de los primeros en informar sobre los mitos relativos a los tiempos de búsqueda y distancias para los discos magnéticos.

Comentarios sobre el futuro de los discos se pueden encontrar en diversas fuentes. Goldstein [1987] proyecta la capacidad y velocidades de E/S para las instalaciones de grandes computadores IBM en 1995, sugiriendo que la relación no es menor de 3,7 GB por MIPS de gran computador IBM de hoy día, y que crecerán a 4,5 GB por MIPS en 1995. Frank [1987] especulaba sobre la densidad de grabación física, proponiendo la fórmula MAD sobre crecimiento de discos que se utilizó en la Sección 9.4. Katz, Patterson y Gibson [1990] dan una visión general actual sobre los discos de alto rendimiento y sistemas de E/S y especulan sobre futuros sistemas. La posibilidad de lograr sistemas de E/S de mayor rendimiento utilizando conexiones de discos se encuentra en artículos de Kim [1986]; Salem y García-Molina [1986], y Patterson, Gibson y Katz [1987].

Mirando hacia atrás en lugar de hacia adelante, la primera máquina en ampliar interrupciones que detectaban anormalidades aritméticas a la detección de eventos asincrónicos de E/S está acreditada como la DYSEAC de NBS en 1954 [Leiner y Alexander, 1954]. El año siguiente estuvo operativa la primera máquina con DMA, la SAGE de IBM. Igual que los DMA actuales, la SAGE tenía contadores de dirección que realizaban transferencias de bloques en paralelo con operaciones de la CPU. El primer canal de E/S podía haber estado en la IBM 709 en 1957 [Bashe y cols., 1981 y 1986]. Smotherman [1989] explora la historia de las E/S con más profundidad.

Referencias

ANON ET AL. [1985]. «A measure of transaction processing power», Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation,* April 1, 1985.

BASHE, C. J., W. BUCHHOLZ, G. V. HAWKINS, J. L. INGRAM, AND N. ROCHESTER [1981]. «The architecture of IBM's early computers», *IBM J. of Research and Development* 25:5 (September) 363-375.

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM's Early Computers,* MIT Press, Cambridge, Mass.

BORRILL, P. L. [1986]. «32-bit buses-An objective comparison», *Proc. Buscon 1986 West,* San Jose, Calif., 138-145.

BRADY, J. T. [1986]. «A theory of productivity in the creative process», *IEEE CG&A* (May) 25-34.

BUCHER, I. V. AND A. H. HAYES [1980]. «I/O Performance measurement on Cray-1 and CDC 7000 computers», *Proc. Computer Performance Evaluation Users Group, 16th Meeting,* NBS 500-65, 245-254.

CHEN, P. [1989]. *An Evaluation of Redundant Arrays of Inexpensive Disks Using an Amdahl 5890,* M. S. Thesis, Computer Science Division, Tech. Rep. UCB/CSD 89/506.

DOHERTY, W. J. AND R. P. KELISKY [1979]. «Managing VM/CMS systems for user effectiveness», *IBM Systems J.* 18:1, 143-166.

FEIERBACK, G. AND D. STEVENSON [1979]. «The Illiac-IV», in *Infotech State of the Art Report on Supercomptuers,* Maidenhead, England. This data also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982). McGraw-Hill, New York, 268-269.

FOLEY, J. D. AND A. VAN DAM [1982]. *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass.

FRANK, P. D. [1987]. «Advances in Head Technology», presentation at *Challenges in Winchester Technology* (December 15), Santa Clara Univ.

FRIESENBORG, S. E. AND R. J. WICKS [1985]. «DASD expectations: The 3380, 3380-23, and MVS/XA», Tech. Bulletin GG22-9363-02 (July 10), Washington Systems Center.

GOLDSTEIN, S. [1987]. «Storage performance—an eight year outlook», Tech. Rep: TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif.

HENLY, M. AND B. MCNUTT [1989]. «DASD I/O characteristics: A comparison of MVS to VM», Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif.

HOWARD, J. H. ET AL. [1988]. «Scale and performance in a distributed file system», *ACM Trans. on Computer Systems* 6:1, 51-81.

IBM [1982]. *The Economic Value of Rapid Response Time,* GE20-0752-0 White Plains, N.Y., 11-82.

IMPRIMIS [1989]. «Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB», Document No. 64402302 (May).

KAHN, R. E. [1972]. «Resource-sharing computer communication networks», *Proc. IEEE* 60:11 (November) 1397-1407.

KATZ, R. H., D. A. PATTERSON, AND G. A. GIBSON [1990]. «Disk system architectures for high performance computing», *Proc. IEEE* 78:2 (February).

KIM, M. Y. [1986]. «Synchronized disk interleaving», *IEEE Trans. on Computers* C-35-11 (November).

LEINER, A. L. [1954]. «System specifications for the DYSEAC», *J. ACM* 1:2 (April) 57-81.

LEINER, A. L. AND S. N. ALEXANDER [1954]. «System organization of the DYSEAC», *IRE Trans. of Electronic Computers* EC-3:1 (March) 1-10.

LEVY, J. V. [1978]. «Buses: The skeleton of computer structures», in *Computer Engineering: A DEC View of Hardware Systems Design,* C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Digital Press, Bedford, Mass.

MABERLY, N. C. [1966]. *Mastering Speed Reading,* New American Library, Inc., New York.

METCALFE, R. M. AND D. R. BOGGS [1976]. «Ethernet: Distributed packet switching for local computer networks», *Comm. ACM* 19:7 (July) 395-404.

NEWMAN, W. N. AND R. F. SPROULL [1979]. *Principles of Interactive Computer Graphics,* 2nd ed., McGraw-Hill, New York.

OUSTERHOUT, J. K. ET AL. [1985]. «A trace-driven analysis of the UNIX 4.2 BSD file system», *Proc. Tenth ACM Symposium on Operating Systems Principles,* Orcas Island, Wash., 15-24.

PATTERSON, D. A., G. A. GIBSON, AND R. H. KATZ [1987]. «A case for redundant arrays of inexpensive disks (RAID)», Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.,* Chicago, Illinois, June 1-3, 1988, 109-116.

ROBINSON, B. AND L. BLOUNT [1986]. «The VM/HPO 3880-23 performance results», IBM Tech. Bulletin, GG66-0247-00 (April), Washington Systems Center, Gathersburg, Md.

SALEM, K. AND H. GARCIA-MOLINA [1986]. «Disk striping», *IEEE 1986 Int'l Conf. on Data Engineering.*

SCRANTON, R. A., D. A. THOMPSON, AND D. W. HUNTER [1983]. «The access time myth», Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N.Y.

SMITH, A. J. [1985]. «Disk cache—miss ratio analysis and design considerations», *ACM Trans. on Computer Systems* 3:3 (August) 161-203.

SMOTHERMAN, M. [1989]. «A sequencing-based taxonomy of I/O systems and review of historical machines», *Computer Architecture News* 17:5 (September) 5-15.

SUTHERLAND, I. E. [1963]. «Sketchpad: A man-machine graphical communication system», *Spring Joint Computer Conf.* 329.

THACKER, C. P., E. M. MCCREIGHT, B. W. LAMPSON, R. F. SPROULL, AND D. R. BOGGS [1982]. «Alto: A personal computer», in *Computer Structures: Principles and Examples,* D.P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549-572.

THADHANI, A. J. [1981]. «Interactive user productivity», *IBM Systems J.* 20:4, 407-423.

THISQUEN, J. [1988]. «Seek time measurements», *Amdahl Peripheral Products Division Tech. Rep.* (May).

EJERCICIOS

9.1 <9.10> [10/25/10] Utilizando las fórmulas de la Figura 9.39

a) [10] Calcular el tiempo de búsqueda para desplazar el brazo un tercio de los cilindros para ambos discos.

b) [25] Escribir un programa que calcule el tiempo «medio» de búsqueda estimando el tiempo para todas las posibles búsquedas, utilizando estas fórmulas y después dividiendo por el número de búsquedas.

c) [10] ¿Cómo aproximar a) a b)?

9.2 <9.10> [15/20] Utilizando las fórmulas de la Figura 9.39 y las estadísticas de la Figura 9.40, calcular la distancia media de búsqueda y el tiempo medio de búsqueda en el IBM 3380J. Utilizar el punto medio de un rango como la distancia de búsqueda. Por ejemplo, usar 98 como distancia de búsqueda para la entrada que representa 91-105 en la Figura 9.40. Para la carga de trabajo comercial, ignorar la pérdida del 5 por 100 de las búsquedas. Para la carga de trabajo de UNIX, suponer que las pérdidas del 15 por 100 de las búsquedas tienen una distancia media de 300 cilindros.

a) [15] Si se hubiese equivocado por la falacia, podría calcular la distancia media como 884/3. ¿Cuál es la distancia medida para cada carga de trabajo?

b) [20] El tiempo para buscar 884/3 cilindros en la IBM 3380J es aproximadamente 12,8 ms. ¿Cuál es el tiempo medio de búsqueda para cada carga de trabajo en la IBM 3380J utilizando las medidas?

9.3 <1.4,8.4,9.4> [20/10/Discusión] Suponer que las mejoras en densidad de las DRAM y discos magnéticos continúan como predice la Figura 1.5. Suponiendo que la mejora en coste para megabyte sigue las mejoras de densidad y que 1990 es el comienzo de la generación de DRAM de 4 megabits, ¿cuando el coste por megabyte de DRAM igualará al coste por megabyte de disco magnético? dado:

- La diferencia de coste en 1990 es que la DRAM es 10 veces más cara.
- La diferencia de coste en 1990 es que la DRAM es 30 veces más cara.

a) [20] ¿Qué generación de chips DRAM —medidas en bits por chip— alcanzarán la equidad para cada hipótesis de la diferencia de costes? ¿Qué año ocurrirá eso?

b) [10] ¿Cuál será la diferencia de coste en la generación anterior?

c) [Discusión] ¿Piensa que la diferencia de coste en la generación anterior es suficiente para sustituir los discos por DRAM?

9.4 <9.2> [12/12/12] Suponer que una carga de trabajo necesita en total 100 segundos, empleando la CPU 70 segundos y las E/S 50 segundos.

a) [12] Suponer que la unidad de punto flotante es responsable de 25 segundos del tiempo de CPU. Se está considerando un acelerador de punto flotante que va cinco veces más rápido. ¿Cuál es el tiempo de la carga de trabajo para máximo solapamiento, solapamiento escalado, y no solapamiento?

b) [12] Suponer que los retardos de búsqueda y de rotación de los discos magnéticos son responsables de 10 segundos del tiempo de E/S. Se está considerando sustituir los discos magnéticos por discos de estado sólido, que eliminen todos los retardos rotacionales y de búsqueda. ¿Cuál es el tiempo de la carga de trabajo para: máximo solapamiento, solapamiento escalado y no solapamiento?

c) [12] ¿Cuál es el tiempo de la carga de trabajo para el solapamiento escalado si se realizan ambos cambios?

9.5-9.9 Rendimiento del tratamiento de transacciones. El bus de E/S y el sistema de memoria de un computador son capaces de sustentar 100 MB/s sin interferir con el rendimiento de una CPU de 80 MIPS (que cuesta 50 000 dólares). Aquí están las suposiciones sobre el software:

- Cada transacción requiere 2 lecturas del disco más 2 escrituras del disco.
- El sistema operativo utiliza 15 000 instrucciones para cada lectura o escritura del disco.
- El software de la base de datos ejecuta 40 000 instrucciones para procesar una transacción.
- El tamaño de la transferencia es de 100 bytes.

Se ha de elegir uno de dos tipos diferentes de discos:

- Un disco de 2,5 pulgadas que almacena 100 MB y cuesta 500 dólares.
- Un disco de 3,5 pulgadas que almacena 250 MB y cuesta 1 250 dólares.
- Cualquier disco del sistema puede soportar una media de 30 escrituras o lecturas del disco por segundo.

Responder las preguntas siguientes utilizando el benchmark TP-1 de la Sección 9.3. Suponer que las peticiones se reparten uniformemente en todos los discos, que no hay tiempo de espera debido a discos ocupados, y que el archivo de cuentas debe ser suficientemente grande para manipular 1 000 TPS de acuerdo con las reglas del benchmark.

9.5 <9.3,9.4> [20] ¿Cuántas transacciones TP-1 por segundo son posibles con cada organización de discos, suponiendo que cada una utiliza el mínimo número de discos para mantener el archivo de contabilidad?

9.6 <9.3,9.4> [15] ¿Cuál es el coste del sistema por transacción por segundo de cada alternativa para TP-1?

9.7 <9.3,9.4> [15] ¿Con qué rapidez convierte una CPU al bus de E/S, de 100 MB por segundo, en un cuello de botella para TP-1? (Suponer que se puede continuar añadiendo discos).

9.8 <9.3,9.4> [15] Como gestor de MTP (Mega TP), usted está decidiendo si gastar su dinero de desarrollo en construir una CPU más rápida o mejorar el rendimiento del software. El grupo de base de datos dice que puede reducir una transacción a una lectura de un disco y una escritura en un disco y reducir las instrucciones de base de datos

por transacciones a 30 000. El grupo de hardware puede construir una CPU más rápida que se vende por la misma cantidad que la CPU más lenta con el mismo presupuesto de desarrollo. (Suponer que se pueden añadir tantos discos como sea necesario para conseguir mayor rendimiento.) ¿Qué rapidez debe tener la CPU para que coincida con la ganancia de rendimiento de la mejora del software?

9.9 <9.3,9.4> [15/15] El grupo de E/S de MTP estuvo escuchando en la puerta durante la presentación del software. Ellos argüían que los avances de la tecnología permitirán que las CPU sean más rápidas sin inversiones significativas, pero que el coste del sistema estará dominado por los discos si no desarrollan nuevos discos más rápidos de 2,5 pulgadas. Suponer que la siguiente CPU es 100 por 100 más rápida al mismo coste y que los nuevos discos tienen la misma capacidad que los antiguos.

a) [15] Dada la nueva CPU y el software antiguo, ¿cuál será el coste de un sistema con suficientes discos antiguos de 2,5 pulgadas para que no limiten el TPS del sistema?

b) [15] Suponer que ahora se tienen tantos discos nuevos como discos antiguos de 2,5 pulgadas se tenian en el diseño original. ¿Qué rapidez deben tener los nuevos discos (E/S por segundo) para lograr la misma velocidad TPS con la nueva CPU del sistema de la parte a)? ¿Cuál será el coste del sistema?

9.10 <9.4> [20/20/20] Suponer que tenemos las dos siguientes configuraciones de disco magnético: un solo disco y un array de cuatro discos. Cada disco tiene 20 superficies, 885 pistas por superficie con 16 sectores/pista, cada sector contiene 1 Kbyte, y gira a 3600 RPM. Utilizar la fórmula del tiempo de búsqueda, para el IBM 3380D de la Figura 9.39. El tiempo para conmutar entre superficies es el mismo que para desplazar el brazo una pista. En el array de discos todos los ejes están sincronizados —el sector 0 de cada disco gira bajo la cabeza al mismo tiempo— y los brazos de los cuatro discos están siempre sobre la misma pista. Los datos están «divididos» entre los cuatro discos; así, cuatro sectores consecutivos en un sistema de un solo disco se repartirán en un sector por disco en el array. El retardo del controlador del disco es 2 ms por transacción, tanto para un solo disco como para el array. Suponer que el rendimiento del sistema de E/S está limitado sólo por los discos y que en el array hay un camino a cada disco. Comparar el rendimiento de estas dos organizaciones de disco, en E/S por segundo y megabytes por segundo suponiendo los siguientes patrones:

a) [20] Lecturas aleatorias de 4 KB de sectores secuenciales. Suponer que los 4 KB están alineados bajo el mismo brazo en cada disco del array.

b) [20] Lecturas de 4 KB de sectores secuenciales donde la distancia media de búsqueda es 10 pistas. Suponer que los 4 KB están alineados bajo el mismo brazo de cada disco en el array.

c) [20] Lecturas aleatorias de 1 MB de sectores secuenciales. (Si interesa, suponer que el controlador del disco supone que los sectores llegan en cualquier orden.)

9.11 [20] <9.4> Suponer que tenemos un disco definido como en el Ejercicio 9.9. Suponer que leemos el siguiente sector después de cualquier lectura y que *todas* las peticiones de lectura tienen una longitud de un sector. Almacenamos los sectores extra que se leyeron antes en una *cache del disco.* Suponer que la probabilidad de recibir una petición para el sector que leímos anticipadamente en algún instante en el futuro (antes debe ser descartado porque llene el buffer de cache del disco) es 0,1. Suponer que debemos pagar todavía los gastos del controlador en un acierto de lectura de la cache de disco, y que el tiempo de transferencia para la cache del disco es de 250 ns por palabra. ¿Es la estrategia de lectura-anticipada más rápida? (Indicación: resolver el problema en

estado estacionario suponiendo que la cache del disco contiene la información apropiada y ha faltado una petición.

9.12-9.14 Suponer la siguiente información sobre nuestra máquina DLX:
Carga 2 ciclos
Almacena 2 ciclos

Todas las demás instrucciones son de 1 ciclo. Usar la información de la mezcla del resumen de instrucciones de la Figura C.4 del Apéndice C sobre DLX para GCC.

Aquí están las estadísticas de la cache para una cache de escritura directa:

- Cada bloque de la cache es de cuatro palabras, y en cualquier fallo se lee el bloque completo.
- Los fallos de la cache necesitan 13 ciclos.
- Las escrituras directas necesitan 6 ciclos para completarse, y no hay buffer de escritura.

Aquí están las estadísticas de la cache para una cache de postescritura:

- Cada bloque de la cache es de cuatro palabras, y en cualquier falta se lee el bloque entero.
- Los fallos de la cache necesitan 13 ciclos para un bloque sin modificar y 21 ciclos para un bloque modificado.
- Suponer que en un fallo, el 30 por 100 del tiempo del bloque está modificado.

Suponer que el bus

- está sólo ocupado durante las transferencias,
- transfiere en promedio 1 palabra/ciclo de reloj, y
- cada vez debe leer o escribir una simple palabra (esto no es más rápido que leer escribir dos de una vez).

9.12 [20/10/20/20] <9.4,9.5,9.6> Suponer que la E/S del DMA puede tener lugar simultáneamente con aciertos de la CPU en la cache. Suponer también que el sistema operativo puede garantizar que no habrá problema de datos obsoletos en la cache debido a las E/S. El tamaño del sector es de 1 KB.

a) [20] Suponer que la frecuencia de fallos de la cache es del 5 por 100. En promedio, ¿qué porcentaje del bus se utiliza para cada política de escritura de la cache? Esta medida se denomina la *relación de tráfico* en los estudios de la cache.

b) [10] Si el bus se puede cargar hasta el 80 por 100 de su capacidad sin sufrir severas penalizaciones de rendimiento, ¿cuánta anchura de banda de memoria está disponible para las E/S, para cada política de escritura de la cache? La frecuencia de fallos de la cache es todavía del 5 por 100.

c) [20] Suponer que una lectura de un sector del disco necesita 1 000 ciclos de reloj para iniciar una lectura, 100 000 ciclos de reloj para encontrar el dato en el disco, y 1 000 ciclos de reloj para que el DMA transfiera el dato a memoria. ¿Cuántas lecturas del disco pueden presentarse por millón de instrucciones ejecutadas para cada política de escritura? ¿Cómo se realiza este cambio si la frecuencia de fallos de la cache se reduce a la mitad?

d) [20] Ahora se puede disponer de cualquier número de discos. Suponiendo plani-

ficación ideal de los accesos al disco, ¿cuál es el máximo número de lecturas de sectores que se pueden presentar por millón de instrucciones ejecutadas?

9.13 [20/20] <9.4,9.5> Muchas máquinas actuales tienen un buffer de encuadre *(frame buffer)* separado para actualizar la pantalla con el fin de evitar la ralentización del sistema de memoria. Una característica interesante es el porcentaje de la anchura de banda de memoria que se podría utilizar si no hubiese buffer de encuadre. Suponer que todos los accesos a memoria son del tamaño de un bloque completo de cache y que necesitan el tiempo de un fallo de la cache. La frecuencia de refresco es de 60 Hz. Utilizando la información de la Sección 9.4, calcular el tráfico de memoria para los siguientes dispositivos gráficos:

1. Una pantalla en blanco y negro de 340 por 540.
2. Una pantalla de color de 1 280 por 1 024 con 24 bits de color.
3. Una pantalla de color de 1 280 por 1 024 utilizando un mapa de colores de 256 palabras.

Suponer que la frecuencia de reloj de la CPU es de 60 MHz.

a) [20] ¿Qué porcentaje de la anchura de banda de memoria/bus consume cada una de las tres pantallas?

b) [20] Suponer que el bus y la memoria principal son de 512 bits de ancho y no de 32 bits. ¿Cuánto tiempo debería emplear un acceso a memoria utilizando ahora el bus más ancho? ¿Qué porcentaje de anchura de banda de memoria se utiliza ahora para cada pantalla?

9.14 [20] <9.4,9.9> El director de almacenamiento del subsistema de E/S IBM 3990 puede tener una gran cache para lecturas y escrituras. Suponer que la cache cuesta igual que cuatro discos 3380D. ¿Qué frecuencia debe conseguir la cache para obtener el mismo rendimiento que cuatro discos más 3380D? (Ver la Figura 9.15 para el rendimiento del 3380.) Suponer que la cache puede soportar 5 000 E/S por segundo si todo son aciertos en la cache.

9.15 [50] <9.3,9.4> Tome su computador favorito y escriba tres programas que consigan lo siguiente:

1. Máximo ancho de banda hacia y desde los discos
2. Máximo ancho de banda para un buffer de encuadre
3. Máximo ancho de banda hacia y desde la red de área local

¿Cuál es el porcentaje de anchura de banda que se puede conseguir comparado con las afirmaciones de los fabricantes de dispositivos de E/S? Registrar también la utilización de la CPU, en cada caso, para los programas que se ejecutan separadamente. A continuación, ejecutar los tres juntos y ver qué porcentaje de máxima anchura de banda se consigue para tres dispositivos de E/S, así como para la utilización de la CPU. Intentar determinar por qué uno obtiene un mayor porcentaje que los demás.

9.16 [40] <9.2> Las fórmulas de velocidad del sistema están limitadas a uno o dos tipos de dispositivos. Obtener fórmulas para un número ilimitado de dispositivos, utilizando tantas suposiciones diferentes sobre solapamiento como se puedan manejar.

9.17 [Discusión] <9.2> ¿Cuáles son los argumentos para predecir el rendimiento del sistema utilizando máximo solapamiento, solapamiento escalado, y no solapamiento? Construir escenarios donde cada uno parezca más probable y otros escenarios donde cada interpretación sea absurda.

9.18 [Discusión] <9.11> ¿Cuáles son las ventajas y desventajas de un sistema de E/S de buffer mínimo que utilizó IBM frente a un sistema de E/S de buffer máximo sobre el coste/rendimiento del sistema de E/S?

El abandono de la organización convencional se produjo a mediados de los años 60, cuando la ley de rendimientos decrecientes comenzó a tener efecto en el esfuerzo de incrementar la velocidad de operación de un computador... Los circuitos electrónicos están en último lugar limitados en su velocidad de operación por la velocidad de la luz... y muchos de los circuitos operan ya en el rango del nanosegundo.

Bouknight y cols. [1972]

... los computadores secuenciales se aproximan a un límite físico fundamental sobre su potencial potencia de cálculo. Dicho límite es la velocidad de la luz...

A. L. DeCegama, *La Tecnología del Procesamiento Paralelo, Volumen I* (1989)

... las máquinas de hoy día... están cercanas a un callejón sin salida cuando la tecnología se aproxima a la velocidad de la luz. Aunque los componentes de un procesador secuencial pudieran trabajar a esta rapidez, lo mejor que se podría esperar no es más de unos pocos millones de instrucciones por segundo.

Mitchell [1989]

10.1 Introducción

10.2 Clasificación de Flynn de los computadores

10.3 Computadores SIMD. Flujo único de instrucciones, flujos múltiples de datos

10.4 Computadores MIMD. Flujos múltiples de instrucciones, flujos múltiples de datos

10.5 Las rutas a El Dorado

10.6 Procesadores de propósito especial

10.7 Direcciones futuras para los compiladores

10.8 Juntando todo: el multiprocesador Sequent Symmetry

10.9 Falacias y pifias

10.10 Observaciones finales. Evolución frente a revolución en arquitectura de computadores

10.11 Perspectiva histórica y referencias

Ejercicios

10 Tendencias futuras

10.1 Introducción

En los nueve capítulos primeros nos hemos limitado a las ideas cuya efectividad ha sido demostrada mediante realizaciones comerciales. Sin embargo, los principios en los que se basan estos capítulos pueden encontrarse en el primer artículo sobre computadores de programa almacenado. Las citas de la página anterior sugieren que los días de los computadores tradicionales están contados. ¡Para un modelo anticuado de cálculo han demostrado seguramente su viabilidad! Hoy está mejorándose en rendimiento a más rapidez que en cualquier momento de la historia, y la mejora en coste y rendimiento desde 1950 ha sido de cinco órdenes de magnitud. ¡Si la industria del transporte hubiese tenido estos avances, podríamos viajar desde San Francisco a Nueva York en un minuto por un dólar!

En este último capítulo abandonamos nuestra perspectiva conservadora y especulamos sobre el futuro de la arquitectura de computadores y de los compiladores. El objetivo de los diseños innovadores son las mejoras espectaculares en coste/rendimiento, o un rendimiento altamente escalable con buen coste/rendimiento. Muchas de las ideas cubiertas aquí han conducido a máquinas que están comenzando a competir en el mercado de computadores de hoy día. Algunas de ellas pueden no estar presentes en la siguiente edición de este libro, mientras que otras pueden necesitar sus propios capítulos.

10.2 Clasificación de Flynn de los computadores

Flynn [1966] propuso un sencillo modelo para clasificar todos los computadores. Observó el paralelismo de los flujos de instrucciones y datos exigidos por las instrucciones en los componentes más restringidos de la máquina, y colocó a todos los computadores en una de cuatro categorías:

1. *Flujo único de instrucciones, flujo único de datos* (SISD, el uniprocesador)
2. *Flujo único de instrucciones, flujos múltiples de datos* (SIMD)
3. *Flujos múltiples de instrucciones, flujo único de datos* (MISD)
4. *Flujos múltiples de instrucciones, flujos múltiples de datos* (MIMD)

Este es un modelo tosco, ya que algunas máquinas son híbridos de estas categorías. No obstante, en este capítulo seguimos este modelo clásico porque es sencillo, fácil de comprender, da una buena primera aproximación y —quizá debido a la facilidad de comprensión— también es el esquema más ampliamente utilizado.

Su primera pregunta sobre el modelo sería, «¿único o múltiple comparado con qué?». Una máquina que suma un número de 32 bits en un ciclo de reloj parecería tener múltiples flujos de datos cuando se compare con un computador de bits en serie que emplee 32 ciclos de reloj para la misma operación. Flynn seleccionó computadores populares de esa fecha, el IBM 704 y el IBM 7090, como modelo de SISD, aunque hoy día cualquiera de las máquinas del Capítulo 4 serviría como ejemplo.

Habiendo establecido entonces el punto de referencia para SISD, la siguiente clase es SIMD.

10.3 Computadores SIMD. Flujo único de instrucciones, flujos múltiples de datos

El coste de un multiprocesador general es, sin embargo, muy alto y se consideraron opciones de diseño adicionales que podían abaratar el coste sin degradar seriamente la potencia o eficiencia del sistema. Las opciones consisten en recentralizar uno de los tres componentes principales... Centralizando la [unidad de control] obtenemos a la organización básica de [un]... procesador en array, tal como el Illiac IV.

Bouknight y cols. [1972]

Ya hemos visto instrucciones típicas para una máquina SIMD, aunque la máquina no era SIMD. Las instrucciones vectoriales del Capítulo 7 operan sobre varios elementos de datos en una sola instrucción, ejecutándose de manera segmentada en una sola unidad funcional. De forma distinta a SIMD, una única instrucción no invoca a muchas unidades funcionales. Una verdadera

SIMD podría tener, por ejemplo, 64 flujos de datos, simultáneamente, hacia 64 ALU para formar 64 sumas en el mismo ciclo de reloj.

Las virtudes de SIMD son que todas las unidades de ejecución paralela están sincronizadas y que todas responden a una única instrucción de un único PC. Desde la perspectiva del programador, esto está próximo a las ya familiares SISD. La motivación original para SIMD fue amortizar el coste de la unidad de control mediante docenas de unidades de ejecución. Una ventaja observada más recientemente es el reducido tamaño de la memoria de programa —SIMD necesita sólo una copia del código que se está ejecutando simultáneamente, mientras que MIMD necesita una copia en cada procesador. Por consiguiente, el coste de memoria de programa para un gran número de unidades de ejecución es menor para SIMD.

Igual que las máquinas vectoriales, los computadores reales SIMD tienen una mezcla de instrucciones SISD y SIMD. Hay un computador SISD para realizar operaciones como bifurcaciones o cálculo de direcciones que no necesitan paralelismo masivo. Las instrucciones SIMD se difunden a todas las unidades de ejecución, cada una de las cuales tiene su propio conjunto de registros. También, como en las máquinas vectoriales, determinadas unidades de ejecución pueden ser inhabilitadas durante una instrucción SIMD. De forma distinta a las máquinas vectoriales, las máquinas SIMD masivamente paralelas cuentan con redes de interconexión o de comunicación para intercambiar datos entre los elementos de procesamiento.

Las máquinas SIMD son apropiadas en situaciones semejantes a las adecuadas para instrucciones vectoriales— tratamiento de arrays en bucles *for*. Por consiguiente, al tener oportunidad de paralelismo masivo en SIMD debe haber cantidades masivas de datos, o *paralelismo de datos*. SIMD tiene su mayor debilidad en las sentencias «case», donde cada unidad de ejecución debe realizar una operación diferente sobre su dato, dependiendo del dato que tenga. Las unidades de ejecución con el dato erróneo son inhabilitadas para que las unidades adecuadas puedan continuar. Estas situaciones, esencialmente, corren a $1/n$ del rendimiento, donde n es el número de «cases».

El compromiso básico en las máquinas SIMD es el rendimiento de un procesador frente al número de procesadores. Las máquinas del mercado actual hacen mayor énfasis en el elevado grado de paralelismo que en el rendimiento de cada procesador. La Connection Machine 2, por ejemplo, ofrece 65 536 procesadores de un bit de ancho mientras que la ILLIAC IV tenía 64 procesadores de 64 bits.

Aunque MISD completa la clasificación de Flynn, es difícil imaginarla. Un flujo único de instrucciones es más sencillo que flujos múltiples de instrucciones, pero flujos múltiples de instrucciones con flujos múltiples de datos son más fáciles de imaginar que múltiples instrucciones con un flujo único de datos. Algunas de las arquitecturas que hemos cubierto pueden considerarse MISD: las arquitecturas superescalar y VLIW del Capítulo 6 (Sección 6.8) tienen, con frecuencia, un flujo único de datos y múltiples instrucciones, aunque estas máquinas tengan un solo contador de programa. Quizá más próximos a este tipo están las arquitecturas desacopladas (págs. 344-345), que tienen dos flujos de instrucciones con contadores de programa independientes y un solo flujo de datos. Las arquitecturas sistólicas, cubiertas en la Sección 10.6, también pueden ser consideradas MISD.

Aunque podemos encontrar ejemplos de SIMD y MISD, su número es muy pequeño comparado con la multitud de máquinas MIMD.

10.4 Computadores MIMD. Flujos múltiples de instrucciones, flujos múltiples de datos

Multis son una nueva clase de computadores basados en múltiples microprocesadores. El pequeño tamaño, bajo coste y alto rendimiento de los microprocesadores permiten el diseño y construcción de estructuras de computadores que ofrecen ventajas significativas en fabricación, relación precio-rendimiento, y fiabilidad sobre las familias de computadores tradicionales... Multis son probablemente la base para la siguiente, la quinta, generación de computadores.

Bell [1985, 463]

Prácticamente, desde que funcionó el primer computador, los arquitectos han estado esforzándose por «El Dorado» del diseño de computadores. Construir un computador potente conectando sencillamente otros muchos más pequeños. El usuario pide tantas CPU como pueda comprar y obtiene una cantidad de rendimiento proporcional. Otras ventajas de MIMD pueden ser: un más alto rendimiento absoluto, más rapidez que el uniprocesador más grande, y la más alta fiabilidad/disponibilidad (pág. 560) vía redundancia.

Durante décadas, los diseñadores de computadores han estado buscando la pieza perdida del puzzle que permite que ocurra esta mejora, por arte de magia. La gente ha oído sentencias que comienzan «Ahora que los computadores han caído a tan bajo precio...» o «Este nuevo esquema de interconexión superará el problema del escalamiento, por ello...» o «Cuando este nuevo lenguaje de programación llegue a extenderse...», y acaban con «MIMD (finalmente) dominarán la computación».

Con tantos intentos de utilizar el paralelismo, hay algunos términos que son útiles conocer cuando se habla de MIMD. La principal división es la que define cómo se comparte la información. Los procesadores de *memoria compartida* ofrecen al programador una sola dirección de memoria a la que todos los procesadores pueden acceder; los múltiprocesadores con cache coherente son máquinas de memoria compartida (ver Secciones 8.8 y 10.8). Los procesos se comunican a través de variables compartidas en memoria, con cargas y almacenamientos capaces de acceder a cualquier posición de memoria. La sincronización debe ser posible para coordinar procesos. Un modelo alternativo para compartir datos se basa en la comunicación entre procesos por medio del envío de mensajes. Como ejemplo extremo, los procesos en diferentes estaciones de trabajo se comunican al enviar mensajes sobre una red de área local. Esta distinción de comunicación es tan fundamental que Bell sugiere que el término *multiprocesador* esté limitado a MIMD que puedan comunicarse vía memoria compartida, mientras que las MIMD que sólo puedan comunicarse vía paso explícito de mensajes deberían ser denominadas *multi-*

computadores. Como una parte de la memoria compartida se puede utilizar para mensajes, muchos multiprocesadores pueden ejecutar eficientemente software de paso de mensajes. Un multicomputador podría simular memoria compartida enviando un mensaje por cada carga o almacenamiento, pero, presumiblemente, esto se ejecutaría con mucha lentitud. Por tanto, la distinción de Bell está basada en el hardware fundamental y modelo de ejecución de programa, reflejado en el rendimiento de la comunicación de memoria compartida, en contraposición al software que puede correr en una máquina. Los partidarios de pasar mensajes cuestionan la *escalabilidad* de los multiprocesadores, mientras que los defensores de la memoria compartida cuestionan la programabilidad de los multicomputadores. La siguiente sección examina este debate.

Las buenas noticias son que después de muchos asaltos, MIMD han conseguido una posición consolidada. Hoy día hay, generalmente, acuerdo en que un multiprocesador puede ser más efectivo para una carga de trabajo en tiempo compartido que un SISD. Un determinado programa no emplea menos tiempo de CPU, pero se puede completar un mayor número de tareas independientes por hora —un argumento de productividad frente a latencia—. No sólo compañías que empiezan, como Encore y Sequent, están vendiendo multiprocesadores de pequeña escala, sino que las máquinas de altas prestaciones de IBM, DEC y Cray Research son multiprocesadores. Esto significa que los multiprocesadores involucran ahora un mercado significativo, responsable de la mayoría de los grandes computadores y virtualmente de todos los supercomputadores. El único punto de insatisfacción de los arquitectos de computadores es que la memoria compartida es prácticamente irrelevante para ejecutar programas de usuario en la máquina, siendo el sistema operativo el único benefactor. El desarrollo de sistemas operativos para multiprocesadores, particularmente su gestor de recursos, se simplifica con la memoria compartida.

Las malas noticias son que queda por ver cuántas aplicaciones importantes corren más rápidas en los MIMD. La dificultad no está en los precios de los SISD, en defectos las topologías de redes de interconexión, o en los lenguajes de programación; sino en la falta de aplicaciones software que han sido reprogramadas para aprovecharse de muchos procesadores y, así, completar más pronto las tareas importantes. Como aún ha sido más difícil encontrar aplicaciones que puedan aprovechar muchos procesadores, el desafío es mayor para los MIMD de gran escala. Cuando las ganancias positivas del tiempo compartido se combinan con la escasez de aplicaciones altamente paralelas, podemos apreciar la difícil situación que afrontan los arquitectos de computadores que diseñan MIMD de gran escala que no soporten tiempo compartido.

¿Pero por qué esto es así? ¿Por qué es mucho más difícil desarrollar programas MIMD que programas secuenciales? Una razón es que es difícil escribir programas MIMD que logren aproximarse a una aceleración lineal cuando se incrementa el número de procesadores dedicados a la tarea. Como analogía, pensar en los gastos de comunicación para una tarea realizada por una persona frente a los gastos para una tarea realizada por un comité, especialmente cuando aumenta el tamaño del grupo. Aunque n personas puedan tener el potencial de terminar cualquier trabajo con n veces más rapidez, los gastos de comunicación para el grupo pueden impedir lograr este objetivo; esto

llega a ser especialmente duro cuando aumenta *n*. (Imaginar el cambio en gastos de comunicación cuando se pasa de 10 personas a 1 000 personas o a 1 000 000 de personas.) Otra razón para la dificultad de escribir programas paralelos es lo que debe conocer el programador sobre el hardware. En un uniprocesador, el programador en lenguaje de alto nivel escribe su programa ignorando la organización fundamental de la máquina —ésta es la tarea del compilador—. Para un multiprocesador actual, el programador ha de conocer mejor el hardware y la organización fundamental si quiere escribir programas rápidos y escalables. Esta relación con el hardware también hace poco frecuentes los programas paralelos portables. Aunque este segundo obstáculo puede reducirse con el tiempo, en estos momentos es el mayor reto de la informática. Finalmente, en el Capítulo 1, la Ley de Amdahl (pág. 9) nos recuerda que, incluso las partes pequeñas de un programa, deben paralelizarse para conseguir un rendimiento alto. Por tanto, aproximarse a una aceleración lineal involucra inventar nuevos algoritmos que sean inherentemente paralelos.

Ejemplo

Suponer que se quiere lograr una aceleración lineal con 100 procesadores. ¿Qué fracción del cálculo original puede ser secuencial?

Respuesta

La Ley de Amdahl es

$$\text{Aceleración} = \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \dfrac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}}}$$

Sustituyendo el objetivo de aceleración lineal con 100 procesadores da:

$$100 = \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \dfrac{\text{Fracción}_{\text{mejorada}}}{100}}$$

Despejando el porcentaje mejorado:

$$100 - 100 \cdot \text{Fracción}_{\text{mejorada}} + 1 \cdot \text{Fracción}_{\text{mejorada}} = 1$$

$$-99 \cdot \text{Fracción}_{\text{mejorada}} = -99$$

$$\text{Fracción}_{\text{mejorada}} = 1$$

Por tanto, para conseguir una velocidad lineal con 100 procesadores, **ningún** cálculo original puede ser secuencial. Dicho de otra forma, para obtener una velocidad de 99 a partir de 100 procesadores significa que la fracción secuencial del programa original había de ser aproximadamente 0,0001.

El ejemplo anterior demuestra la necesidad de nuevos algoritmos. Esto respalda el pensamiento de los autores de que los éxitos importantes en utilizar máquinas paralelas a gran escala, en los años noventa, serán posibles para los que comprendan aplicaciones, algoritmos y arquitectura.

10.5 Las rutas a El Dorado

La Figura 10.1 muestra el estado de la industria, dibujando el número de procesadores frente a rendimiento de cada procesador. La cuestión sobre paralelismo masivo es si tomar la ruta superior o la ruta inferior de la Figura 10.1 para que nos lleve a El Dorado. Actualmente no sabemos lo suficiente sobre programación paralela y aplicaciones para poder medir cuantitativamente el compromiso entre número de procesadores y rendimiento por procesador para lograr el mejor coste/rendimiento.

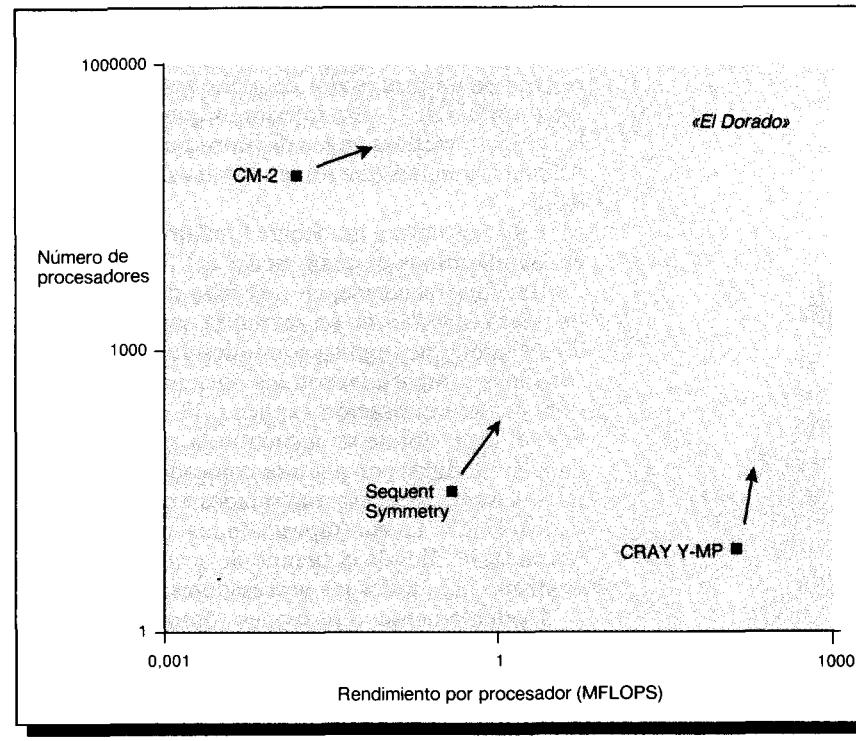


FIGURA 10.1 Danny Hillis, arquitecto de las Connection Machines, ha utilizado una figura similar a ésta para ilustrar la industria de multiprocesadores. (El eje x de Hillis era la anchura del procesador en lugar del rendimiento del procesador.) El rendimiento del procesador en este gráfico se aproxima por la velocidad en MFLOPS de un **solo** procesador para el procedimiento DAXPY del benchmark Linpack para una matriz 1 000 x 1 000. Generalmente, es más fácil para los programadores desplazarse a la derecha, mientras que al diseñador hardware es más fácil desplazarse hacia arriba porque hay más replicación hardware. La pregunta del paralelismo masivo es «¿Cuál es el camino más rápido hacia el extremo superior derecho?». La pregunta del diseñador de computadores es «¿Qué tiene mejor coste/rendimiento o es más escalable para el mismo coste/rendimiento?».

Es interesante observar que se necesitan cambios muy diferentes para mejorar el rendimiento, dependiendo si se toma la ruta inferior o la superior de esta figura. Como la mayoría de los programas se escriben en lenguajes de alto nivel, desplazarse a lo largo de la dirección horizontal (incrementar el rendimiento por procesador) es casi completamente un asunto de mejorar el hardware. Las aplicaciones permanecen inalteradas, adaptando los compiladores al procesador más potente. Por consiguiente, incrementar el rendimiento del procesador frente al número de procesadores es más fácil para las aplicaciones software. Mejorar el rendimiento desplazándose en la dirección vertical (incrementando el paralelismo), por otro lado, puede involucrar cambios significativos en las aplicaciones, ya que programar diez procesadores puede ser muy diferente de programar mil, y todavía diferente de programar un millón. (Pero ir de 100 a 101 probablemente no sea diferente.) Una ventaja del camino vertical para el rendimiento es que el hardware se puede replicar sencillamente —los procesadores en particular, pero también el hardware de interconexión. Por consiguiente, incrementar el número de procesadores frente al rendimiento de un procesador da como resultado mayor replicación del hardware. Una ventaja de la ruta inferior es que es mucho más probable que tenga realizaciones prácticas en los distintos puntos a lo largo del camino a El Dorado. Además, aquellos que tomen la ruta superior deben aferrarse a la Ley de Amdahl.

Esto nos lleva a un debate fundamental sobre la organización de memoria en las máquinas de gran escala del futuro. El debate, desgraciadamente, se centra, con frecuencia, en una falsa dicotomía: *memoria compartida* frente a *memoria distribuida.* La memoria compartida significa un simple espacio de direcciones, que implica comunicación implícita. La contraposición real a direcciones compartidas son los *espacios de direcciones privadas múltiples,* que implican comunicación explícita. El concepto de memoria distribuida se refiere al lugar donde se encuentra la memoria. Si la memoria física está dividida en módulos con algunos colocados cerca de cada procesador (lo que permite tiempos de acceso más rápidos a esa memoria), entonces la memoria física es distribuida. La contraposición real de la memoria distribuida es la *memoria centralizada,* donde el tiempo de acceso a una posición de memoria física es el mismo para todos los procesadores.

Evidentemente, direcciones compartidas frente a direcciones múltiples y memoria distribuida frente a memoria centralizada son cuestiones ortogonales: los SIMD o MIMD pueden tener direcciones compartidas y una memoria física distribuida o espacios de direcciones privadas múltiples y una memoria física centralizada (aunque esta última combinación sería inusual). La Figura 10.2 clasifica diversas máquinas según estos ejes. Los debates apropiados relativos al futuro están en los pros y contras de un único espacio de direcciones y los pros y contras de la memoria distribuida.

El debate sobre un único espacio de direcciones está íntimamente relacionado con el modelo de comunicación, ya que las máquinas de direcciones compartidas deben ofrecer comunicación implícita (posiblemente, como parte de cualquier acceso a memoria) y las máquinas de múltiples direcciones deben tener comunicaciones explícitas. (No es tan simple, ya que algunas máquinas de direcciones compartidas ofrecen también comunicaciones explícitas de varias formas.) Los «implicistas» critican a los «explicistas» de abogar

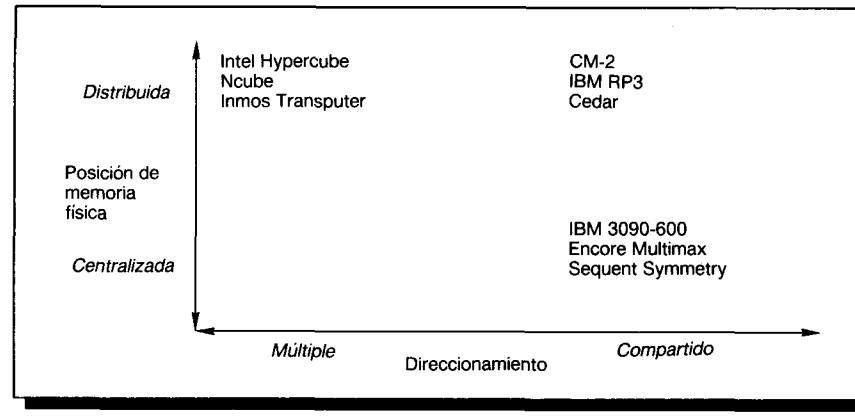


FIGURA 10.2 Procesadores paralelos colocados de acuerdo con memoria centralizada frente a distribuida y direccionamiento compartido frente a múltiple. En general es más fácil el software para las máquinas en el extremo compartido del eje de direccionamiento y es más fácil construir máquinas de gran escala en el extremo distribuido del acceso vertical. Estas máquinas del gráfico se describen en la Sección 10.11.

por máquinas que son más difíciles de programar cuando ya es bastante difícil encontrar aplicaciones: ¿por qué hacer la vida del programador más difícil cuando el software es el eje del paralelismo de gran escala? Una réplica es que, si la memoria es distribuida, cuando los procesadores se hacen más rápidos, el tiempo para la memoria remota será tan grande —digamos de 50 a 100 ciclos de reloj— que el compilador o el programador deben ser conscientes que están escribiendo para una máquina paralela de gran escala sin importar el esquema de comunicación que utilice. La comunicación explícita también ofrece la posibilidad de ocultar el coste de la comunicación solapándolo con el del cálculo. Los implicistas replican que utilizando hardware, en lugar de instrucciones explícitas, se reduce el gasto de comunicación. Además, un único espacio de direcciones significa que los procesos pueden utilizar punteros y comunicar datos sólo si el puntero es desreferenciado, mientras que la comunicación explícita significa que el dato se debe enviar en presencia de punteros, ya que el dato **puede** ser accedido. La refutación de los explicistas es que el propietario del dato puede enviar el dato, atravesando una red propiamente diseñada, sólo una vez, mientras que en las máquinas de memoria compartida un procesador pide el dato y, entonces, el propietario lo devuelve, requiriendo dos viajes sobre la red de comunicaciones.

Los defensores de la memoria distribuida arguyen que la memoria central tiene un ancho de banda limitado sin importar la cantidad de cache que se utilice y, por tanto, limita el número de procesadores. Los defensores de la memoria central argumentan la cuestión de la eficiencia: ¿si no hay suficiente paralelismo para utilizar muchos procesadores, entonces por qué distribuir memoria? Los centralistas también señalan que la memoria distribuida incrementa la dificultad de programación, ya que ahora el programador o el compilador deben decidir cómo organizar los datos en los módulos de la memoria

física para reducir la comunicación. Por consiguiente, la memoria distribuida introduce el concepto de elementos de datos próximos (el módulo que necesita menos tiempo de acceso) o lejanos (en otros módulos de memoria) a un procesador.

Ahora podemos explicar una dificultad de la dicotomía «distribuida frente centralizada». Cada procesador, probablemente, tendrá una cache, que es, en algún sentido, una memoria distribuida sin importar cómo esté organizada la memoria principal. Aun con las caches, la latencia de un fallo y el ancho de banda efectivo para satisfacer peticiones cache se pueden mejorar si los datos están ubicados en el módulo de memoria cerca de la cache adecuada. Por consiguiente, hay todavía una distinción entre memoria principal centralizada y distribuida en la presencia de caches.

Como se puede imaginar, estos debates continúan de una parte a otra, prácticamente de manera interminable. Afortunadamente, en arquitectura de computadores estos desacuerdos se saldan con medidas en lugar de con polémicas. Por tanto, el tiempo será el juez de estas cuestiones, pero sus autores serán el juez de una apuesta inspirada por estos debates (ver pág. 635 en 10.11).

Las cuestiones reales para las máquinas futuras son éstas: ¿existen problemas y algoritmos con suficiente paralelismo? Y ¿puede la gente ser entrenada o los compiladores ser escritos para explotar tal paralelismo?

10.6 Procesadores de propósito especial

Además de explorar el paralelismo, muchos diseñadores hoy día están explorando computadores de propósito especial. La creciente sofisticación del software de *diseño ayudado por computador* y la creciente capacidad por chip conlleva la oportunidad de construir rápidamente un chip que haga bien una cosa a bajo coste. El tratamiento de imágenes y el reconocimiento del habla en tiempo real son ejemplos. Estos dispositivos de propósito especial, o *coprocesadores,* frecuentemente actúan en unión de la CPU. Hay dos tipos en la tendencia de coprocesadores: procesadores de señales digitales y arrays sistólicos.

Los *procesadores de señales digitales* (o DSP) no derivan del modelo tradicional de cálculo, y tienden a parecer máquinas horizontales microprogramadas (ver pág. 226) o máquinas VLIW (ver págs. 346-348). Tienden a resolver problemas de tiempo real, que tienen esencialmente un flujo de datos de entrada infinito. Ha habido poco énfasis en la compilación a partir de lenguajes de programación como C, pero esto está empezando a cambiar. Cuando los DSP se dobleguen frente a las demandas de los lenguajes de programación, será interesante ver cómo difieren de los microprocesadores tradicionales.

Los arrays sistólicos evolucionaron de intentos de obtener un ancho de banda de cálculo más eficiente del silicio. Los arrays sistólicos se pueden considerar como un método para diseñar computadores de propósito especial para equilibrar recursos, ancho de banda de E/S y cálculo. Basándose en la segmentación, los datos fluyen en etapas desde memoria a un array de unidades de cálculo y vuelta a memoria, como sugiere la Figura 10.3. Recientemente, la investigación sobre arrays sistólicos se ha desplazado desde muchos chips

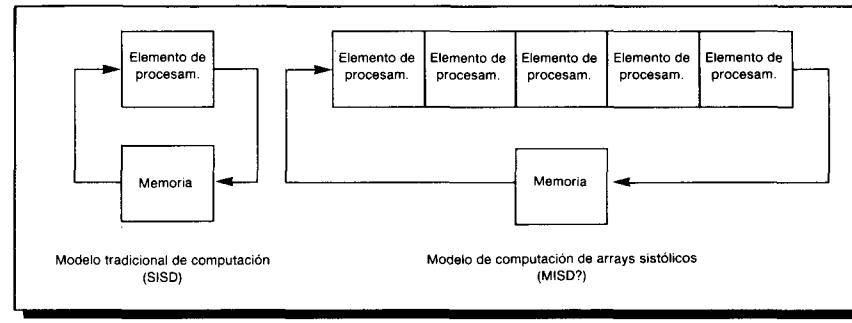


FIGURA 10.3 La arquitectura sistólica toma su nombre del corazón bombeando rítmicamente la sangre. Los datos llegan a un elemento de procesamiento a intervalos regulares, donde se modifica y pasa al siguiente elemento, y así sucesivamente, hasta que vuelve a circular a memoria. Algunos consideran los arrays sistólicos un ejemplo de MISD.

de propósito especial dedicados a menos chips, más potentes, que son programables.

Los autores esperan, en los años noventa, un creciente papel para los computadores de propósito especial, porque ofrecen mayor rendimiento y menor coste para funciones dedicadas, como tratamiento de imágenes y reconocimiento del habla en tiempo real. El mercado de los consumidores parece el candidato más probable, dado su alto volumen y sensibilidad al coste.

10.7 Direcciones futuras para los compiladores

Los compiladores del futuro tienen dos desafíos sobre las máquinas del futuro:

- Organización de los datos para reducir los gastos de jerarquía de memoria y de comunicación, y
- Explotación del paralelismo.

Los programas del futuro gastarán un porcentaje mayor de tiempo de ejecución esperando la jerarquía de memoria cuando crezca el desnivel entre la duración del ciclo de reloj de los procesadores y el tiempo de acceso a memoria principal (ver Fig. 8.18). Los compiladores que organizan códigos y datos para reducir los fallos de cache pueden conducir a mejoras mayores de rendimiento que las optimizaciones tradicionales de hoy día. Mejoras adicionales son posibles con la posibilidad de prebúsqueda de datos en una cache antes de que los necesite el programa. Una proposición interesante es que al extender los lenguajes de programación existentes con operaciones sobre arrays, un programador puede expresar el paralelismo con cálculo sobre arrays completos, dejando al compilador que organice los datos en los procesadores para reducir la cantidad de comunicaciones. Por ejemplo, la extensión propuesta a

FORTRAN 77, denominada FORTRAN 8X, incluye extensiones de arrays. La esperanza es que la tarea del programador pueda incluso ser más simple que con máquinas SISD, donde las operaciones de arrays deben ser especificadas con bucles. El rango de programas que dicho compilador puede manejar eficientemente, y el número de indicaciones que un programador debe suministrar sobre dónde colocar los datos, determinará el valor práctico de esta propuesta.

Además de reducir los costes de accesos a memoria y comunicación, los compiladores pueden cambiar el rendimiento en factores de dos o tres, al utilizar el paralelismo disponible en el procesador. La Figura 2.25 muestra que los benchmarks de Perfect Club operan sólo al 1 por 100 del rendimiento máximo, sugiriendo claramente muchas oportunidades para el software. Más específicamente, las máquinas superescalares del Capítulo 6, normalmente, consiguen una aceleración menor de 2 utilizando los compiladores actuales, aun cuando la mejora potencial de rendimiento al ejecutar cuatro instrucciones a la vez sea 4. En el Capítulo 7 vimos que las máquinas vectoriales normalmente consiguen una frecuencia de vectorización del 40 al 70 por 100, obteniendo una aceleración de 1,5 a 2,5, mientras que una frecuencia de vectorización del 90 por 100 podría conseguir una aceleración en torno a 5. Y compiladores actuales para multiprocesadores se consideran un éxito si consiguen una aceleración de 3 para un solo programa cuando el potencial de ocho procesadores es 8. La Figura 10.4 muestra la mejora potencial del ren-

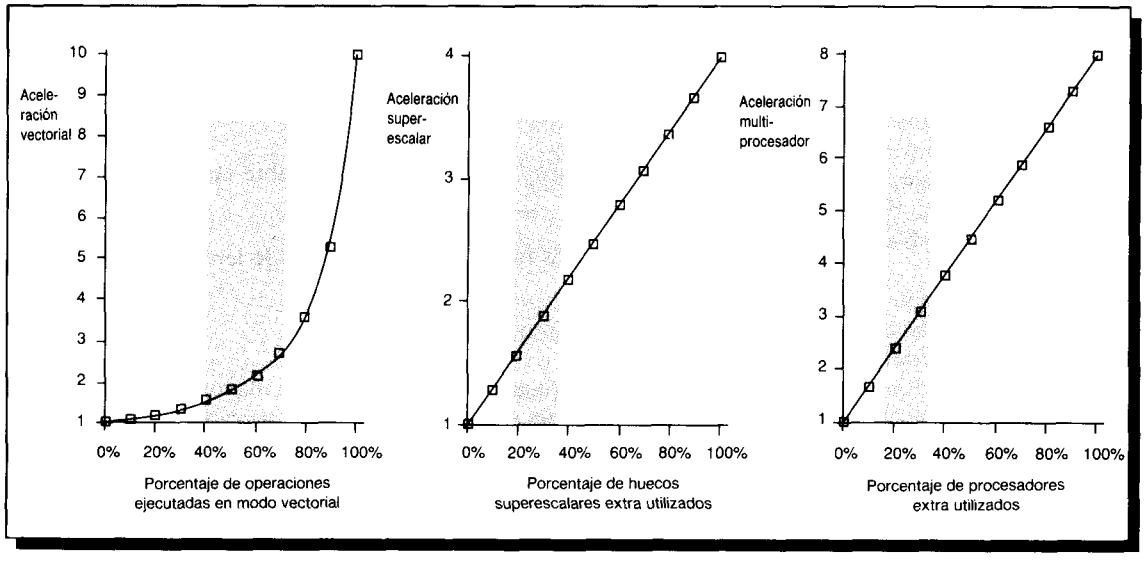


FIGURA 10.4 Potencial de mejora del rendimiento mediante compiladores que transformen una mayor parte de la computación al modo más rápido. El gráfico de más a la izquierda muestra el porcentaje de operaciones ejecutadas en modo vectorial, mientras que los otros gráficos muestran el porcentaje de la aceleración potencial en uso en promedio: porcentaje de cuatro instrucciones utilizadas por ciclo en el superescalar y porcentaje del tiempo de uso de los 8 procesadores en el multiprocesador. El área gris muestra el rango de utilización normalmente encontrado en programas que utilizan compiladores actuales.

dimiento debido a ejecutar un mayor porcentaje del trabajo en el modo del mayor rendimiento para cada una de estas categorías. Como podemos esperar, procesadores múltiples en máquinas donde cada procesador tenga características vectoriales o superescalares, la aceleración potencial de estos factores puede multiplicarse.

Aunque esta oportunidad existe para los compiladores, no queremos minimizar su dificultad. La paralelización de compiladores se ha desarrollado desde 1975, pero el progreso ha sido lento. Estos problemas son difíciles, especialmente para el reto de ejecutar programas existentes. Los éxitos han estado limitados a programas donde el paralelismo está disponible en el algoritmo y se expresa en el programa y a máquinas con un pequeño número de procesadores. ¡Progresos significativos pueden eventualmente requerir nuevos lenguajes de programación así como compiladores más inteligentes!

10.8 Juntando todo: el multiprocesador Sequent Symmetry

El alto rendimiento y bajo coste de los microprocesadores inspiraron un renovado interés en los multiprocesadores en los años ochenta. Varios microprocesadores se pueden colocar sobre un bus común porque:

son mucho más pequeños que los procesadores multichip,

las caches pueden disminuir el tráfico del bus y

los protocolos de coherencia pueden mantener caches y memoria consistente.

El tráfico por procesador y el ancho de banda del bus determinan el número de procesadores en dicho multiprocesador.

Algunos proyectos de investigación y compañías investigaron estos multiprocesadores de bus compartido. Un ejemplo es Sequent Corporation, fundada para construir multiprocesadores basados en microprocesadores estándares, y el sistema operativo UNIX. El primer sistema fue el Balance 8000, ofertado en 1984 con un número de microprocesadores National 32032 comprendido entre 2 y 12, un bus de transacciones de 32 bits que multiplexaba direcciones y datos, y una cache de escritura directa, asociativa por conjuntos, de 2 vías y 8 KB por procesador. Cada cache observa el bus para mantener la coherencia utilizando escritura directa con invalidación. (Ver Secciones 8.4, 8.8 y 9.4 para revisar estos términos.) El ancho de banda sostenida de la memoria principal y del bus es de 26,7 MB/s. Dos años más tarde Sequent se modernizó con el Balance 21000, ofreciendo hasta 30 microprocesadores National 32032 con el mismo bus y sistema de memoria.

En 1986, Sequent comenzó a diseñar el multiprocesador Symmetry, utilizando un microprocesador de 300 a 400 por 100 más rápido que el 32032. El objetivo fue soportar tantos procesadores como fuese posible utilizando los controladores de E/S desarrollados para el sistema de Balance. Esto significó que el bus tenía que permanecer compatible, aunque los nuevos sistemas de

bus y memoria debían proporcionar aproximadamente del 300 al 400 por 100 más ancho de banda que el sistema más antiguo.

El objetivo de mayor ancho de banda del sistema de memoria con un bus similar fue atacado en cuatro niveles. Primero, la cache se incrementó a 64 KB, aumentando la frecuencia de aciertos y, por tanto, el ancho de banda efectivo visto por el procesador. Segundo, la política de cache fue cambiada de escritura directa a postescritura para reducir el número de operaciones de escritura sobre el bus compartido. Para mantener la coherencia cache con postescritura, Symmetry utiliza un esquema de invalidación en escritura (ver págs. 504-506). El tercer cambio fue duplicar la anchura del bus a 64 bits, doblando casi el ancho de banda del bus a 53 MB/s. El cambio final fue que cada controlador de memoria entrelazaba la memoria de dos bancos (ver Sección 8.8), permitiendo que el sistema de memoria se adecuase al mayor ancho de banda del bus. El sistema de memoria puede tener hasta seis controladores, con un total de hasta 240 MB de memoria principal.

El uso de lenguajes de alto nivel y la portabilidad del sistema operativo UNIX permitió cambiar los repertorios de instrucciones al Intel 80386 más rápido. Corriendo a una frecuencia de reloj mayor, con el acelerador, más rápido, Weitek 1167 de punto flotante, y con el sistema de memoria mejorado, un único 80386 corría de 214 a 776 por 100 con más rapidez para benchmarks en punto flotante y, aproximadamente, el 375 por 100 con más rapidez para benchmarks enteros. La Figura 10.5 muestra la organización del Symmetry.

Otra de las restricciones de diseño fue que los nuevos circuitos impresos de Symmetry tenían que funcionar adecuadamente cuando se pusiesen en los antiguos sistemas de Balance. Como el nuevo sistema tenía que utilizar postescritura y el sistema antiguo utilizaba escritura directa, el equipo hardware resolvió el problema diseñanado las nuevas caches para que soportasen o escritura directa o postescritura. Lovett y Thakkar [1988] aprovecharon esa característica para ejecutar programas paralelos con ambas políticas. La Figura 10.6 muestra la utilización del bus frente al número de procesadores, para cuatro programas paralelos.

Como mencionamos antes, la utilización del bus está estrechamente ligada al número de procesadores que se pueden utilizar en los sistemas de bus único. Las caches de escritura directa deberían tener mayor utilización del bus para el mismo número de procesadores, ya que cada escritura debe ir sobre el bus; o desde una perspectiva diferente, el mismo bus debería poder soportar más procesadores si utilizan caches de postescritura. La Figura 10.6 confirma completamente nuestras expectativas; los buses se saturan con menos de 16 procesadores con escritura directa, pero la postescritura parece ser escable hasta el tamaño total.

Hay dos componentes para el tráfico de bus: los fallos normales y el soporte de coherencia. Los fallos del uniprocesador (forzoso, capacidad y conflicto) pueden reducirse con caches mayores y con mejores políticas de escritura, pero el tráfico de coherencia es una función del programa paralelo. El beneficio principal de la postescritura para los programas de la Figura 10.6 fue simplemente reducir el número de escrituras en el bus debido a la política de postescritura, ya que hubía pocas escrituras a datos compartidos en estos programas.

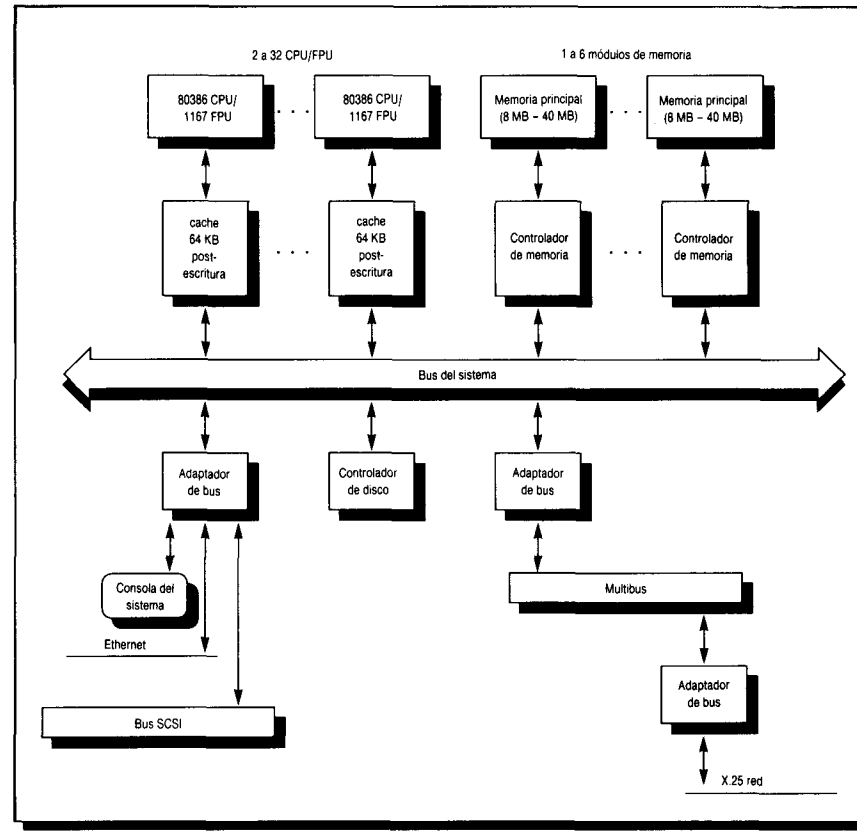


FIGURA 10.5 El multiprocesador Sequent Symmetry tiene hasta 30 microprocesadores, cada uno con 64 KB de cache de postescritura, asociativa por conjuntos, de 2 vías conectada al bus del sistema compartido. Hasta seis controladores de memoria también se conectan a este bus de 64 bits, más algunas interfaces para E/S. Además de un controlador de disco de propósito especial, hay una interfaz para la consola del sistema, red Ethernet, y bus SCSI de E/S (ver Cap. 9), así como otra interfaz para Multibus. Los dispositivos de E/S se pueden conectar al SCSI o al Multibus, según desee el cliente. (Aunque todas las interfaces están etiquetadas «Adaptador de Bus», cada una tiene un diseño único.)

Otro experimento evaluó al Symmetry como un multiprocesador de tiempo compartido (multiprogramado), ejecutando 10 programas independientes. El experimento ejecutaba *n* copias del programa sobre *n* procesadores. Este estudio determinó que, aproximadamente, la mitad de los programas comenzaban a desviarse de un incremento linealmente de la productividad a partir de 6 a 8 procesadores con escritura directa. Sin embargo, con postescritura permanecía casi lineal para todos, excepto uno, de los 10 programas para 28 procesadores. (El fallo fue debido a los puntos calientes *(hot spots)* del sistema operativo en lugar de al protocolo de coherencia de postescritura.)

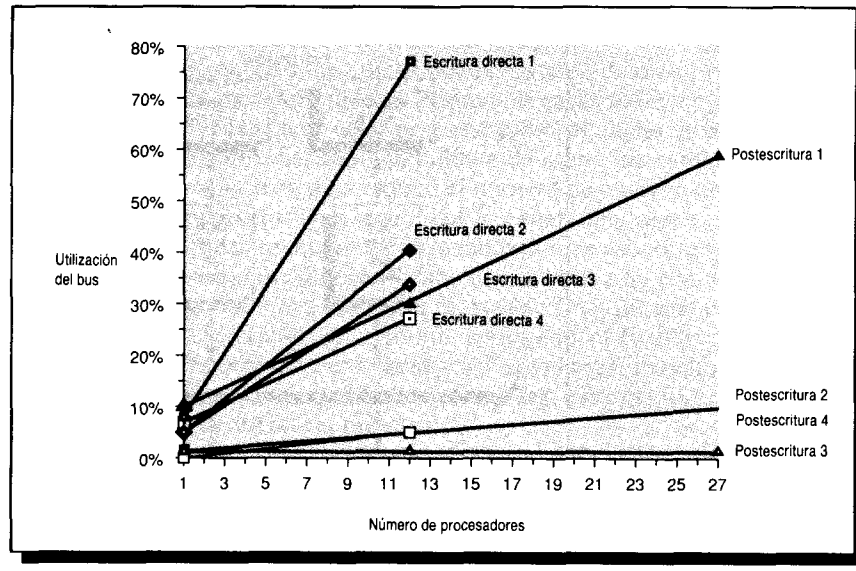


FIGURA 10.6 Comparación del impacto de coherencia cache de escritura directa frente a postescritura en la utilización del bus del multiprocesador Simmetry de Sequent para cuatro benchmarks: (1) Simulador Butterfly Switch, (2) Simulación 2D Monte Carlo, (3) Ray Tracing y (4) Parallel Linpack Benchmark. Lovett y Thakkar [1988] obtuvieron estos datos con un monitor de rendimiento hardware.

10.9 Falacias y pifias

Dada la naturaleza especulativa de este capítulo, parece que esta sección no sería necesaria. Sin embargo, en buena conciencia, planteamos dos avisos.

Pifia: Medir el rendimiento de multiprocesadores mediante aceleración lineal frente al tiempo de ejecución.

Las gráficas «disparo de mortero» —dibujan el rendimiento frente al número de procesadores mostrando un crecimiento lineal, una meseta y después una caída— se han utilizado largamente para juzgar el éxito de los procesadores paralelos. Aunque la escalabilidad es una faceta de un programa paralelo, no es una medida directa del rendimiento. La primera pregunta es la potencia de los procesadores que se están escalando: un programa que mejora linealmente el rendimiento, hasta 100 Intels 8080, puede ser más lento que la versión secuencial en una estación de trabajo. Ser especialmente cuidadoso en los programas intensivos de punto flotante, como elementos de procesamiento que, sin asistencia hardware, pueden escalar estupendamente pero tienen poco rendimiento colectivo.

Comparar tiempos de ejecución es sólo justo si se están comparando los mejores algoritmos en cada máquina. (Por supuesto, no se puede restar tiempo de procesadores inactivos cuando se evalúa un multiprocesador; así, el tiempo de CPU es inapropiado para los multiprocesadores.) Comparar el código idéntico en dos máquinas puede parecer justo, pero no lo es; el programa paralelo puede ser más lento en un uniprocesador que una versión secuencial. A veces, desarrollar un programa paralelo conducirá a mejoras algorítmicas; así que comparar el programa secuencial previamente mejor conocido con el código paralelo —lo que parece justo— no comparará algoritmos equivalentes. Para reflejar este hecho, a veces se utilizan los términos de *velocidad relativa* (mismo programa) y *velocidad verdadera* (los mejores programas). Los resultados que sugieren rendimiento *superlineal,* cuando un programa en n procesadores es más de n veces más rápido que el equivalente uniprocesador, da un indicio a comparaciones injustas.

Falacia: La Ley de Amdahl no se aplica a los computadores paralelos.

En 1987, el responsable de una organización de investigación afirmaba que la Ley de Amdahl (ver Sección 1.3) había sido quebrantada por una máquina MIMD. Sin embargo, esto no significó que la ley haya sido revocada para los computadores paralelos. La parte despreciada del programa limitará todavía el rendimiento. Para tratar de comprender la base de los informes de los medios, veamos lo que decía originalmente la Ley de Amdahl [1967]:

Una conclusión obvia que puede deducirse en este punto, es que el esfuerzo empleado en lograr elevadas velocidades de procesamiento paralelo es inútil, a menos que esté acompañado por logros en velocidades de procesamiento secuencial de, aproximadamente, la misma magnitud. [pág. 483]

Una interpretación de la ley fue que, como hay porciones de cada programa que deben ser secuenciales, hay un límite al número económico de procesadores —digamos 100—. Al mostrar aceleración lineal con 1 000 procesadores, esta interpretación de la Ley de Amdahl fue refutada.

La aproximación de los investigadores fue cambiar la entrada al benchmark, para que, en lugar de ir 1 000 veces más rápido, esencialmente, calculase 1 000 veces más trabajo en tiempo comparable. Para su algoritmo la parte secuencial del programa fue constante, independiente del tamaño de la entrada, y el resto fue completamente paralelo —de aquí, la aceleración lineal con 1 000 procesadores.

El Capítulo 2 (ver Sección 2.2) describe los peligros de dejar a cada experimentador seleccionar su propia entrada para los benchmarks. No vemos ninguna razón por la que, variar la entrada, sea digno de confianza para evaluar el rendimiento de los multiprocesadores ni por la que no pueda aplicarse la Ley de Amdahl. Lo que esta investigación destaca es la importancia de tener benchmarks que sean suficientemente grandes para demostrar el rendimiento de procesadores paralelos de gran escala.

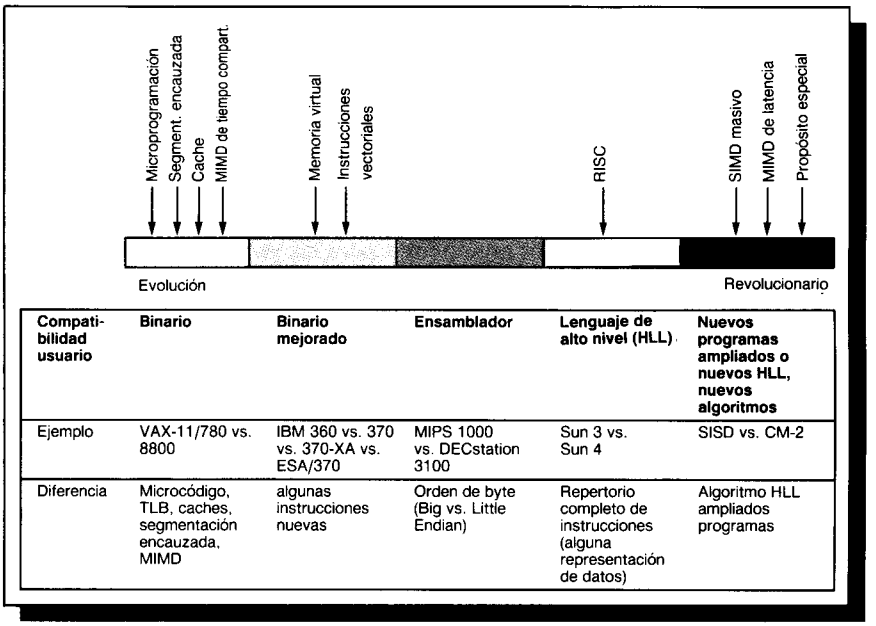
10.10 Observaciones finales. Evolución frente a revolución en arquitectura de computadores

La lectura de conferencias y artículos de revistas desde los últimos veinte años puede dejar a uno desalentado; se ha gastado demasiado esfuerzo para tan poco impacto. Hablando con optimismo, estos artículos actúan como puzzle y, cuando se colocan juntos, lógicamente, forman los fundamentos de la siguiente generación de computadores. Desde un punto de vista más pesimista, si el 90 por 100 de las ideas desaparecieran no se notaría.

Una razón para que esto ocurra se denomina el «síndrome de von Neumann». Esperando inventar un nuevo modelo de computación que revolucione el cálculo, los investigadores están luchando para llegar a ser conocidos como el von Neumann del siglo XXI. Otra razón es de gusto: los investigadores, con frecuencia, seleccionan problemas que no tienen interés para los demás. Incluso cuando se seleccionan problemas importantes, hay frecuentemente una falta de evidencia experimental para demostrar convincentemente el valor de la solución. Además, cuando se seleccionan problemas importantes y se demuestran las soluciones, las soluciones propuestas pueden ser muy caras en relación con sus beneficios. A veces este gasto se mide como coste/rendimiento —la mejora del rendimiento no es digna del coste añadido—. Con más frecuencia, los gastos de innovación son demasiado perjudiciales para los usuarios de los computadores. La Figura 10.7 muestra lo que significamos por *espectro de evolución-revolución* de la innovación en la arquitectura de computadores. A la izquierda están las ideas que son invisibles al usuario (presumiblemente orientadas a obtener mejor coste, mejor rendimiento o ambas cosas). Este es el extremo evolutivo del espectro. En el otro extremo están las ideas revolucionarias de arquitectura. Aquellas son las ideas que requieren nuevas aplicaciones de los programadores, que deben aprender nuevos lenguajes de programación y modelos de computación, y deben inventar nuevas estructuras de datos y algoritmos.

Las ideas revolucionarias son más fáciles de publicar que las ideas evolutivas, pero para que se adopten deben tener recompensas mucho mayores. Las caches son un ejemplo de una mejora evolutiva. Cinco años después de la primera publicación sobre las caches, casi todas las compañías de computadores estaban diseñando una máquina con cache. Las ideas RISC están más cerca de la mitad del espectro; para ellas se necesitaron casi diez años para que muchas compañías tuviesen un producto RISC. Un ejemplo de arquitectura de computador revolucionaria es la Connection Machine. Cada programa que se ejecuta eficientemente en esa máquina era sustancialmente modificado o escrito especialmente para ella, y los programadores necesitaban aprender un nuevo estilo de programación para ella. Thinking Machines fue fundada en el año de 1983, pero tan sólo unas pocas compañías ofrecían ese estilo de máquina.

Hay valor en proyectos que no afectan a la industria de computadores a causa de lecciones que documentan esfuerzos futuros. El pecado no es tener una arquitectura nueva que no sea un éxito comercial; el pecado está en no evaluar cuantitativamente las fortalezas y debilidades de las nuevas ideas. La



Compatibilidad usuario	Binario	Binario mejorado	Ensamblador	Lenguaje de alto nivel (HLL)	Nuevos programas ampliados o nuevos HLL, nuevos algoritmos
Ejemplo	VAX-11/780 vs. 8800	IBM 360 vs. 370 vs. 370-XA vs. ESA/370	MIPS 1000 vs. DECstation 3100	Sun 3 vs. Sun 4	SISD vs. CM-2
Diferencia	Microcódigo, TLB, caches, segmentación encauzada, MIMD	algunas instrucciones nuevas	Orden de byte (Big vs. Little Endian)	Repertorio completo de instrucciones (alguna representación de datos)	Algoritmo HLL ampliados programas

FIGURA 10.7 El espectro evolución-revolución de la arquitectura de computadores. Las cuatro primeras columnas se distinguen de la última columna en que las aplicaciones y sistemas operativos pueden ser transportados desde otros computadores en lugar de escritos de nuevo. Por ejemplo, RISC está listado en el medio del espectro porque la compatibilidad del usuario está sólo a nivel de lenguajes de alto nivel, mientras que la microprogramación permite compatibilidad binaria, y MIMDs orientado a latencia requiere cambios para los algoritmos y extensión de los HLL (lenguajes de alto nivel). MIMD de tiempo compartido significa MIMD justiticado al ejecutar a la vez muchos programas independientes. Mientras que MIMD de latencia significa MIMD pensados para ejecutar con más rapidez un único programa.

siguiente sección menciona varias máquinas cuya principal contribución es la documentación de la máquina y la experiencia de utilizarla.

Cuando se contemple el futuro —y cuando se inventen sus propias contribuciones— recordar el espectro evolución-revolución. Tener también en cuenta las leyes y principios de la arquitectura de computadores que aparecían en los primeros capítulos; éstas, seguramente, guiarán a los computadores del futuro, como han guiado a los computadores del pasado.

10.11 Perspectiva histórica y referencias

Durante una década los profetas han predicado el argumento de que la organización de un solo computador ha alcanzado sus límites y que avances verdaderamente significativos sólo pueden hacerse interconectando una

multiplicidad de computadores de tal forma que permitan soluciones cooperativas... La demostración consiste en la validez continuada del enfoque de un procesador...

Amdahl [1967, 483]

Las citas, en la apertura del capítulo, dan los argumentos clásicos para abandonar la forma actual de computación, y Amdahl [1967] da la réplica clásica. ¡Los argumentos para las ventajas de la ejecución paralela pueden remontarse al siglo XIX [Menabrea, 1842]! Con todo, la efectividad del multiprocesador para reducir la latencia de programas importantes individuales está siendo determinanda todavía.

Las primeras ideas sobre computadores estilo SIMD son de Unger [1958] y Slotnick, Borck y McReynolds [1962]. El diseño Solomon de Slotnick formó las bases del Illiac IV, quizás el más infame de los proyectos de supercomputadores. Aunque con éxito al introducir varias tecnologías útiles en proyectos posteriores, falló como computador. Los costes aumentaron desde los ocho millones de dólares estimados en 1966 a 31 millones de dólares en 1972, a pesar de construir solamente una cuarta parte de la máquina planeada. El rendimiento real fue como máximo de 15 MFLOPS frente a las predicciones iniciales de 1 000 MFLOPS para el sistema completo (ver Hord [1982]). Suministrado a NASA Ames Research en 1972, el computador necesitó tres años más de ingeniería antes de que fuese utilizable. Estos eventos ralentizaron la investigación de los SIMD, con Danny Hillis [1985] resucitando este estilo en la Connection Machine: el coste de una memoria de programa para cada uno de los 65 636 procesadores de 1 bit era prohibitivo, y SIMD fue la solución.

Es difícil señalar el primer multiprocesador. El primer computador de Eckert-Mauchly Corporation, por ejemplo, tenía unidades duplicadas para mejorar la disponibilidad. Holland [1959] dio los primeros argumentos para procesadores múltiples. Después de varios intentos de laboratorio en multiprocesadores, los años 1980 vieron con éxito los primeros multiprocesadores comerciales. Bell [1985] sugirió que la clave era que el tamaño más pequeño de los microprocesadores permitía al bus de memoria sustituir al hardware de la red de interconexión, y que los sistemas operativos portables significaban proyectos de multiprocesadores que no requerían la invención de un nuevo sistema operativo. Este es el artículo en el que se definen los términos «multiprocesador» y «multicomputador». Dos de los proyectos de multiprocesadores mejor documentados son el C.mmp [Wulf y Bell, 1972, y Wulf y Harbison, 1978] y Cm* [Swan y cols., 1977, y Gehringer, Siewiorek y Segall, 1987]. Multiprocesadores comerciales recientes incluyen el Encore Multimax [Wilson, 1987] y el Symmetry de Sequent [Lovett y Thakkar, 1988]. El Cosmic Cube es un multicomputador de la primera época [Seitz, 1985]. Multicomputadores comerciales recientes son el Intel Hypercube y las máquinas basadas en Transputer [Whitby-Strevens, 1985]. Intentos en construir un multiprocesador escalable de memoria compartida incluyen el IBM RP3 [Pfister, Brantley, George, Harvey, Kleinfekder, McAuliffe, Melton, Norton y Weiss, 1985], el NYU Ultracomputer [Schwartz, 1980, y Elder, Gottlieb, Kruskal, McAuliffe, Randolph, Snir, Teller y Wilson, 1985] y el proyecto Cedar de la Universidad de Illinois [Gajksi, Kuck, Lawrie y Sameh, 1983].

Hay información ilimitada sobre multiprocesadores y multicomputadores: conferencias, artículos de revistas e incluso libros parece ser que aparecen con más rapidez de la que cualquier persona pueda asimilar las ideas. Una buena fuente es la International Conference on Parallel Processing, que se celebra anualmente desde 1972. Dos libros recientes sobre cálculo paralelo han sido escritos por Almasi y Gottlieb [1989], y Hockney y Jesshope [1988]. Eugene Miya de NASA Ames ha recopilado una bibliografía «on-line» de artículos de procesamiento paralelo que contiene más de 10 000 entradas. Para destacar algunos artículos, envía peticiones electrónicas cada mes de enero para preguntar qué artículos debe leer todo estudiante serio de este campo. Después de reunir los votos, escoge los 10 artículos recomendados con más frecuencia y publica esa lista. Aquí se da una lista alfabética de los ganadores: Andrews y Schneider [1983]; Batcher [1974]; Dewitt, Finkel y Solomon [1984]; Kuhn y Padua [1981]; Lipovsky y Tripathi [1977]; Russell [1978]; Seitz [1985]; Swan, Fuller y Siewiorek [1977]; Treleaven, Brownbridge y Hopkins [1982], y Wulf y Bell [1972].

Los computadores de propósito especial son anteriores a los computadores de programa almacenado. Brodersen [1989] da una historia del procesamiento de señal y su evolución para dispositivos programables. H. T. Kung [1982] acuñó el término «arrays sistólicos» y ha sido uno de los principales proponentes de este estilo de diseño de computadores. Investigaciones recientes han ido en la dirección de hacer programables los elementos de los arrays sistólicos y proporcionar un entorno de programación para simplificar la tarea de programar.

Es difícil predecir el futuro; con todo Gordon Bell ha hecho dos predicciones para 1995. La primera es que un computador capaz de sostener un TeraFLOPS —un millón de MFLOPS— se construirá alrededor de 1995, bien utilizando un multicomputador de 4 a 32 K nodos o una Connection Machine con varios millones de elementos de procesamiento [Bell, 1989]. Para poner esta predicción en perspectiva, cada año el Gordon Bell Prize es un reconocimiento a los avances en paralelismo, incluyendo el programa real más rápido (más MFLOPS). En 1988, el ganador logró 400 MFLOPS utilizando una CRAY X-MP con cuatro procesadores y 16 megapalabras y, en 1989, el ganador utilizó una CRAY Y-MP de ocho procesadores para conseguir a 1 680 MFLOPS. Máquinas y programas tendrán que mejorar en un factor de tres cada año para que el programa más rápido consiga 1 TFLOPS en 1995.

La segunda predicción de Bell está relacionada con el número del flujo de datos en los supercomputadores construidos en 1995. Danny Hillis piensa que aunque los supercomputadores con un pequeño número de flujo de datos puedan ser los que tengan una mayor demanda, las máquinas más grandes serán aquéllas con muchos flujos de datos, y éstas realizarán la mayor parte de los cálculos. Bell apuesta a Hillis que en el último trimestre del año 1995 se contruirán más MFLOPS sostenidos en máquinas utilizando pocos flujos de datos ($\leqslant 100$) en lugar de con muchos flujos de datos ($\geqslant 1\,000$). Esta apuesta concierne solamente a supercomputadores, definidos como máquinas que cuestan más de 1 000 000 de dólares y son utilizadas para aplicaciones científicas. Los MFLOPS sostenidos se definen para esta apuesta como el número de operaciones en punto flotante por **mes**; así, la disponibilidad de las máquinas afecta a su velocidad. El perdedor debe escribir y publicar un artículo ex-

plicando por qué falló su predicción; los autores actuarán como jueces y jurado.

Referencias

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing,* Benjamin/Cummings, Redwood City, Calif.

AMDAHL, G. M. [1967]. «Validity of the single processor approach to achieving large scale computing capabilities», *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N.J. (April) 483-485.

ANDREWS, G. R. AND B. SCHNEIDER [1983]. «Concept and notations for concurrent programming», *Computing Surveys* 15:1 (March) 3-43.

BATCHER, K. E. [1974]. «STARAN parallel processor system hardware», *Proc. AFIPS National Computer Conference,* 405-410.

BELL, C. G. [1985]. «Multis: A new class of multiprocessor computers», *Science* 228 (April 26) 462-567.

BELL, C. G. [1989]. «The future of high performance computers in science and engineering», *Comm. ACM* 32:9 (September) 1091-1101.

BOUKNIGHT, W. J., S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. «The ILLIAC IV system», *Proc. IEEE* 60:4, 369-379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), 306-316.

BRODERSEN, R. W. [1989]. «Evolution of VLSI signal-processing circuits», *Proc. Decennial Caltech Conf. on VLSI* (March) 43-46, The MIT Press, Pasadena, Calif.

DEWITT, D. J., R. FINKEL, AND M. SOLOMON [1984]. «The CRYSTAL multicomputer: Design and implementation experience, Computer Sciences Tech. Rep. No. 553, University of Wisconsin-Madison, September.

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL, K. P. CCAULIFFE, L. RANDOLPH, M. SNIR, P. TELLER, AND J. WILSON [1985]. «Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach», *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 126-135.

FLYNN, M. J. [1966]. «Very high-speed computing systems», *Proc. IEEE* 54:12 (December) 1901-1909.

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. «CEDAR—A large scale multiprocessor», *Proc. Int'l Conf. on Parallel Processing* (August) 524-529.

GEHRINGER, E. F., D. P. SIEVIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience,* Digital Press, Bedford, Mass.

HILLIS, W. D. [1985]. *The Connection Machine,* The MIT Press, Cambridge, Mass.

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms,* Adam Hilger Ltd., Bristol, England and Philadelphia.

HOLLAND, J. H. [1959]. «A universal computer capable of executing an arbitrary number of subprograms simultaneously», *Proc. East Joint Computer Conf.* 16, 108-113.

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer,* Computer Science Press, Rockville, Md.

KUHN, R. H. AND D. A. PADUA, EDS. [1981]. *Tutorial on Parallel Processing,* IEEE.

KUNG, H. T. [1982]. «Why systolic architectures?», *IEEE Computer* 15:1, 37-46.

LIPOVSKI, A. G. AND A. TRIPATHI [1977]. «A reconfigurable varistructure array processor», *Proc. 1977 Int'l Conf. of Parallel Processing* (August), 165-174.

LOVETT, T. AND S. THAKKAR [1988]. «The Symmetry multiprocessor system», *Proc. 1988 Int'l Conf. of Parallel Processing,* University Park, Pennsylvania, 303-310.

MENABREA, L. F. [1842]. «Sketch of the analytical engine invented by Charles Babbage», Bibliothèque Universelle de Genève (October).

MITCHELL, D. [1989]. «The Transputer: The time is now», *Computer Design,* RISC supplement, 40-41 (November).

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. «The IBM research parallel processor prototype (RP3): Introduction and architecture», *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 764-771.

RUSSELL, R. M. [1978]. «The Cray-1 computer system», *Comm. ACM* 21:1 (January) 63-72.
SEITZ, C. [1985]. «The Cosmic Cube», *Comm. ACM* 28:1 (January) 22-31.
SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. «The Solomon computer», *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97-107.
SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. «The implementation of the Cm* multi-microprocesador», *Proc. AFIPS National Computing Conf.*, 645-654.
SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. «Cm*—A modular, multimicroprocesor», *Proc. AFIPS National Computer Conf.* 46, 637-644.
SWARTZ, J. T. [1980]. «Ultracomputers», *ACM Transactions on Programming Languages and Systems* 4:2, 484-521.
TRELEAVEN, P. C., D. R. BROWNBRIDGE, AND R. P. HOPKINS [1982]. «Data-driven and demand-driven computer architectures», *Computing Surveys* 14:1 (March) 93-143.
UNGER, S. H. [1958]. «A computer oriented towards spatial problems», *Proc. Institute of Radio Engineers* 46:10 (October) 1744-1750.
VON NEUMANN, J. [1945]. «First draft of a report on the EDVAC». Reprinted in W. Aspray and A. Burks, eds., *Papers of John von Neumann on Computing and Computer Theory* (1987), 17-82, The MIT Press, Cambridge, Mass.
WHITBY-STREVENS, C. [1985]. «The transputer», *Proc. 12th Int'l Symposium on Computer Architecture,* Boston, Mass. (June) 292-300.
WILSON, A. W., JR. [1987]. «Hierarchical cache/bus architecture for shared memory multiprocesors», *Proc. 14th Int'l Symposium on Computer Architecture* (June), Pittsburg, Penn., 244-252.
WULF, W. AND C. G. BELL [1972]. «C.mmp—A multi-mini-processor», *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765-777.
WULF, W. AND S. P. HARBISON [1978]. «Reflections in a pool of processors—An experience report on C.mmp/Hydra», *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif. 939-951.

EJERCICIOS

10.1 [Discusión] <10.4> La debilidad de los SIMD para las sentencias «case», así como el fallo de la primera máquina al popularizar los SIMD, impidió la exploración de diseños SIMD mientras los MIMD eran todavía una frontera abierta. Los MIMD tienen, además, la ventaja de aprovechar la onda de mejoras en los procesadores SISD. Ahora que la programación MIMD no ha sucumbido fácilmente a los asaltos de los científicos de los computadores, la cuestión surge si el modelo de programación más simple de SIMD podría conducir a la victoria sobre MIMD para un gran número de procesadores. Parece como si los programas MIMD para miles de procesadores constaran de miles de copias de un programa en lugar de miles de diferentes programas. Por tanto, la dirección es hacia un único **programa** con múltiples flujos de datos, independientemente que la máquina sea SIMD o MIMD. ¿Qué tendencias favorecen MIMD sobre SIMD, y viceversa? Asegurarse de considerar la utilización de memoria y procesadores (incluyendo comunicación y sincronización).

10.2 [Discusión] <10.3,10.5> Se podrían emplear aproximadamente 100 ciclos para una comunicación en una máquina masivamente paralela SIMD o MIMD. ¿Qué técnicas hardware podrían emplearse esta vez? ¿Cómo se cambiaría la arquitectura o el modelo de programación para hacer un computador más inmune a dichos retardos?

10.3 [Discusión] <10.4,10.8> ¿Qué debe ocurrir antes que las máquinas MIMD orientadas a latencia llegasen a ser frecuentes?

10.4 [Discusión] <10.6> ¿Cuándo tiene sentido económicamente hacer procesadores de propósito especial?

10.5 [Discusión] <10.8> Construir un escenario donde una arquitectura completamente revolucionaria —escoja su candidata favorita— juegue un papel importante. Importante se define como el 10 por 100 de computadores vendidos, 10 por 100 de usuarios, 10 por 100 del dinero gastado en computadores o 10 por 100 de alguna otra figura de mérito.

10.6 [30] <10.2> El CM-2 utiliza 64K procesadores de 1 bit en modo SIMD. 32 operaciones sobre bits se pueden simular fácilmente mediante un paso de un SISD de 32 bits, al menos para las operaciones lógicas. El CM-2 emplea aproximadamente 500 ns para dichas operaciones. Si se tiene acceso a un SISD rápido, calcular cuánto tiempo necesitan la suma y la AND lógica sobre 64K números de 1 bit.

10.7 [30] <10.2> Análogo a la pregunta anterior, un uso popular del CM-2 es operar sobre datos de 32 bits, utilizando múltiples pasos con 64K procesadores de 1 bit. El CM-2 necesita, aproximadamente, 16 microsegundos para una suma o AND de 32 bits. Simular esta actividad sobre un SISD rápido; calcular cuánto tiempo necesita para la suma y AND lógica sobre 64K números de 32 bits.

10.8-10.12 <2.2,10.4> **Si se tiene acceso a algunos multiprocesadores o multicomputadores diferentes, la comparación de rendimientos es la base de algunos proyectos.**

10.8 [50] <2.2-10.4> Un argumento para aceleración superlineal (págs. 630-631) es que el tiempo empleado en servir interrupciones o cambiar de contexto se reduce cuando se tienen muchos procesadores, ya que sólo uno necesita servir interrupciones y hay más procesadores para ser compartidos por los usuarios. Medir el tiempo empleado en una carga de trabajo manipulando interrupciones o cambiando de contexto sobre un uniprocesador frente a un multiprocesador. Esta carga de trabajo puede ser una mezcla de tareas independientes para un entorno de multiprogramación o un solo trabajo largo. ¿Se cumple el argumento?

10.9 [50] <2.2,10.4> Un multiprocesador o multicomputador es comercializado normalmente utilizando programas que pueden escalar linealmente el rendimiento con el número de procesadores. El proyecto consistiría en llevar programas escritos para una máquina a las demás y medir su rendimiento absoluto y cómo cambia cuando se cambia el número de procesadores. ¿Qué cambios es necesario hacer para mejorar el rendimiento de los programas llevados a cada máquina? ¿Cuál es la relación del rendimiento de los procesadores para cada programa?

10.10 [50] <2.2,10.4> En lugar de intentar crear benchmarks adecuados, inventar programas que hagan que un multiprocesador o multicomputador parezca horrible comparado con los demás, y también programas que siempre hagan que uno parezca mejor que los demás. Sería un resultado interesante si no se pudiese encontrar un programa que haga que un multiprocesador o multicomputador parezca peor que los demás. ¿Cuáles son las características clave del rendimiento de cada organización?

10.11 [50] <2.2,10.4> Los multiprocesadores y multicomputadores muestran, habitualmente, que el rendimiento aumenta cuando se incrementa el número de procesadores, siendo a la aceleración ideal para n procesadores. El objetivo de este benchmark polarizado es hacer que un programa obtenga peor rendimiento cuando se añadan procesadores. Por ejemplo, esto significa que 1 procesador en el multiprocesador o multicomputador ejecuta el programa el que más rápido, 2 con más lentitud, 4 con más lentitud que 2, y así sucesivamente. ¿Cuáles son las características clave del rendimiento, para cada organización, que den una aceleración lineal inversa?

10.12 [50] <10.4> Las estaciones de trabajo conectadas en red se pueden considerar multicomputadores, aunque con comunicación lenta relativa a la computación. Adapte benchmarks de multicomputadores para ejecutarlos en una red utilizando llamadas remotas a procedimientos para comunicación. ¿Cómo se escalan los benchmarks en la red frente al multicomputador? ¿Cuáles son las diferencias prácticas entre estaciones de trabajo en red y un multicomputador comercial?

Lo rápido expulsa a lo lento aun cuando lo rápido sea erróneo.

W. Kahan

por David Goldberg
(Xerox Palo Alto Research Center)

A.1 Introducción

A.2 Técnicas básicas de la aritmética entera

A.3 Punto flotante

A.4 Suma en punto flotante

A.5 Multiplicación en punto flotante

A.6 División y resto

A.7 Precisiones y tratamiento de excepciones

A.8 Aceleración de la suma entera

A.9 Aceleración de la multiplicación y división enteras

A.10 Juntando todo

A.11 Falacias y pifias

A.12 Perspectiva histórica y referencias

Ejercicios

A Aritmética de computadores

A.1 Introducción

Se ha propuesto una gran variedad de algoritmos para utilizarlos en los aceleradores de punto flotante. Sin embargo, los chips actuales de punto flotante están habitualmente, basados en refinamientos y variaciones de algunos algoritmos básicos. En este apéndice, nos centraremos en esos algoritmos. Además de escoger algoritmos para la suma, resta, multiplicación y división, el arquitecto de computadores debe decidir si va más allá de lo básico. ¿Se implementará la raíz cuadrada en hardware o software? ¿Se deberá implementar la precisión extendida? Este apéndice le dará el conocimiento básico para tomar estas y otras decisiones.

Nuestra discusión sobre el punto flotante se centrará casi exclusivamente en el estándar de punto flotante del IEEE (IEEE 754) a causa de su gran aceptación. Aunque la aritmética de punto flotante involucra la manipulación de exponentes y desplazamiento de fracciones, la mayor cantidad del tiempo en las operaciones de punto flotante se emplea en operar sobre fracciones utilizando algoritmos de números enteros (pero no necesariamente utilizando el hardware de enteros). Por tanto, después de nuestra discusión sobre el punto flotante, daremos una visión más detallada de los algoritmos de enteros.

Unas buenas referencias en aritmética de computadores, en orden de las menos a las más detalladas, están en el Capítulo 7 de Hamacher, Vranesic y Zaky [1984], Gosling [1980] y Scott [1985].

A.2 Técnicas básicas de la aritmética entera

Los lectores que hayan estudiado aritmética de computadores anteriormente, encontrarán que la mayor parte de esta sección es una revisión.

Suma con transmisión de acarreo

Los componentes de un sumador que pueda calcular la suma de números de n bits $a_{n-1}...a_1a_0$ y $b_{n-1}...b_1b_0$ *son los semisumadores* y *sumadores completos.* El semisumador toma dos bits a_i y b_i como entrada y produce como salida un bit de suma s_i y un bit de acarreo c_{i+1}. Matemáticamente, $s_i = (a_i + b_i) \bmod 2$, y $c_{i+1} = \lfloor (a_i + b_i)/2 \rfloor$, donde $\lfloor\ \rfloor$ es la función parte entera por abajo. Como ecuaciones lógicas, $s_i = a_i\bar{b}_i + \bar{a}_ib_i$, y $c_{i+1} = a_ib_i$, donde a_ib_i significa $a_i \wedge b_i$ y $a_i + b_i$ significa $a_i \vee b_i$. El semisumador también se denomina sumador (2,2), ya que toma dos entradas y produce dos salidas. El sumador completo es un sumador (3,2) y se define por las ecuaciones lógicas

A.2.1
$$s_i = a_i\bar{b}_i\bar{c}_i + \bar{a}_ib_i\bar{c}_i + \bar{a}_i\bar{b}_ic_i + a_ib_ic_i$$

A.2.2
$$c_{i+1} = a_ib_i + a_ic_i + b_ic_i$$

La entrada c_i se denomina el *acarreo de entrada*, mientras que c_{i+1} es el *acarreo de salida.* El problema principal al construir un sumador para números de n bits es propagar los acarreos. La forma más obvia de resolver esto es con un *sumador con transmisión de acarreo (ripple-carry adder)*, que consta de n sumadores completos, como se ilustra en la Figura A.1. (En las figuras de este apéndice los bits menos significativos están siempre a la derecha.) La salida c_{i+1} del iésimo sumador alimenta a la entrada c_{i+1} del siguiente sumador (el sumador $(i + 1)$-ésimo); como el acarreo de orden inferior es cero, el sumador de orden inferior será un semisumador. Sin embargo, más tarde veremos que, inicializar el bit del acarreo de entrada de orden inferior a 1, es útil para realizar la resta o sustracción.

A partir de la Ecuación A.2.2, hay dos niveles de lógica involucrados en calcular c_{i+1} a partir de c_i. Por tanto, si el bit menos significativo genera un acarreo, y ese acarreo se propaga hasta el último sumador, la señal a_0 pasará a través de $2n$ niveles de lógica antes que la puerta final pueda determinar si hay un acarreo de salida de la posición más significativa. En general, el tiempo que un circuito emplea en producir una salida es proporcional al máximo número de niveles lógicos a través de los cuales viaja una señal. Sin embargo, determinar la relación exacta entre los niveles lógicos y el tiempo depende enormemente de la tecnología. Por tanto, cuando se comparan sumadores, sencillamente comparamos el número de niveles lógicos de cada uno. Para un sumador con transmisión de acarreo que opera sobre n bits, hay $2n$ niveles lógicos. Valores típicos de n son 32 para aritmética entera y 53 para punto flotante de doble precisión. El sumador con transmisión de acarreo es el su-

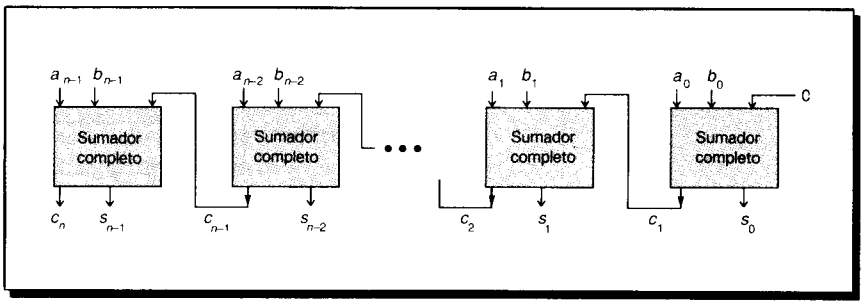


FIGURA A.1 Sumador con transmisión de acarreo, que consta de *n* sumadores completos. El acarreo de salida de un sumador completo se conecta al acarreo de entrada del sumador del siguiente bit más significativo. Los acarreos se transmiten desde el bit menos significativo (a la derecha) al bit más significativo (a la izquierda).

mador más lento, pero también el más barato. Se puede construir con sólo n celdas, conectadas de forma regular y sencilla.

Debido a que este sumador es relativamente lento comparado con los diseños explicados en la Sección A.8, nos podríamos preguntar por qué se utiliza. En las tecnologías como CMOS, aunque los sumadores con propagación emplean un tiempo $O(n)$, el factor constante es muy pequeño. En tales casos se utilizan, con frecuencia, pequeños sumadores con transmisión como componentes de sumadores mayores.

Multiplicación y división en base 2

El multiplicador más sencillo opera sobre dos números sin signo, produciendo cada vez un bit, como se ilustra en la Figura A.2(a). Los números que se van a multiplicar son $a_{n-1}a_{n-2}...a_0$ y $b_{n-1}b_{n-2}...b_0$, y se colocan en los registros A y B, respectivamente. El registro P está inicialmente a cero. Hay dos partes en cada paso de la multiplicación.

1. Si el bit menos significativo de A es 1, entonces el registro B, que contiene $b_{n-1}b_{n-2}...b_0$, se suma a P; en cualquier otro caso 00...00 se suma a P. La suma se vuelve a colocar en P.
2. Los registros P y A se desplazan a la derecha, el bit de orden inferior de P se desplaza al registro A y el bit más a la derecha de A, que no se utiliza en el resto del algoritmo, se desplaza fuera.

Después de n pasos, el producto aparece en los registros P y A, conteniendo A los bits menos significativos.

El divisor más sencillo también opera sobre números sin signo y produce cada vez un bit. Un divisor hardware se muestra en la Figura A.2(b). Para calcular a/b, se pone a en el registro A, b en el registro B, 0 en el registro P, y después se procede como sigue:

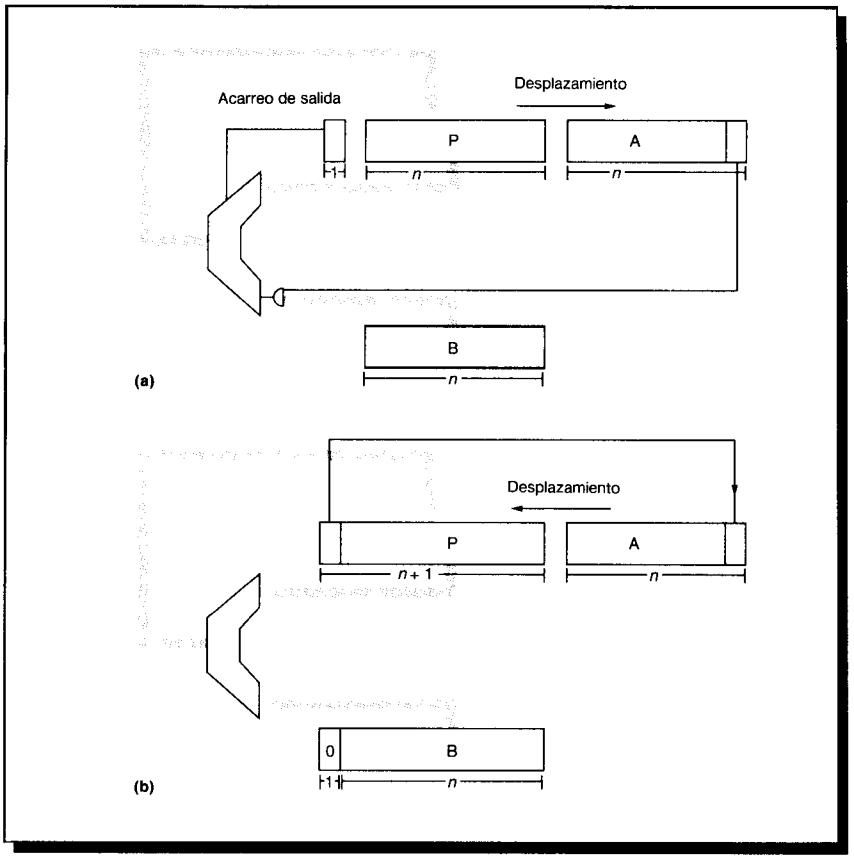


FIGURA A.2 Diagrama de bloques de multiplicador (a) y divisor (b) sencillos para enteros de *n* bits sin signo. Cada paso de la multiplicación consiste en sumar el contenido de P bien a B o a 0 (dependiendo del bit de orden inferior de A); sustituir P por la suma, y después desplazar P y A un bit a la derecha. Cada paso de la división involucra primero desplazar P y A un bit a la izquierda, restar B de P, y si la diferencia es no negativa, ponerla en P. Si la diferencia es no negativa, el bit de orden inferior de A se pone a 1.

1. Desplazar el par de registros (P,A) un bit a la izquierda.
2. Restar el contenido del registro B (que es $b_{n-1}b_{n-2}...b_0$) del registro P.
3. Si el resultado del paso 2 es negativo, poner el bit de orden inferior de A a 0, en cualquier otro caso a 1.
4. Si el resultado del paso 2 es negativo, restaurar el valor antiguo de P sumando el contenido del registro B de nuevo a P.

Después de repetir esto n veces, el registro A contendrá el cociente, y el registro P contendrá el resto. Este algoritmo es la versión binaria del método de papel-y-lápiz; un ejemplo numérico se ilustra en la Figura A.3(a).

Observar que los dos diagramas de bloques de la Figura A.2 son muy similares. La diferencia principal es que el par de registros (P,A) se desplaza a la derecha cuando se multiplica y a la izquierda cuando se divide. Al permitir que estos registros se desplacen bidireccionalmente, se puede compartir el mismo hardware entre multiplicación y división.

El algoritmo de división ilustrado en la Figura A.3(a) se denomina de *restauración*, porque si la resta de b da un resultado negativo, el registro P se restaura, volviéndole a sumar b. El paso de restauración (4 anterior) puede eliminarse fácilmente. Para ver por qué, sea r el contenido del par de registros (P,A), con un punto binario entre el bit de orden inferior de P y el bit de orden superior de A. Después de cada paso del algoritmo, se calcula $2r - b$, estando la palabra de orden superior de esta diferencia en P, y la de orden inferior en A. Suponer que el resultado de un paso es negativo. Normalmente, volveríamos a sumar b (obteniendo $2r$), desplazar (obteniendo $4r$), y después restar (obteniendo $4r - b$). Suponer que no restauramos, pero continuamos con el algoritmo. Primero, desplazar lo no restaurado $2r - b$, obteniendo $4r - 2b$, después sumamos b, obteniendo $4r - b$. ¡Esto es exactamente lo que habríamos obtenido si hubiésemos restaurado! Por tanto, el algoritmo de *no restauración* es

Si P es negativo,

1a. Desplazar el par de registros (P,A) un bit a la izquierda.

2a. Sumar el contenido del registro B al P.

Si no,

1b. Desplazar el par de registros (P,A) un bit a la izquierda.

2b. Restar el contenido del registro B del P.

Finalmente,

3. Si P es negativo, poner el bit de orden inferior de A a 0, en otro caso a 1.

Después de repetir esto n veces, el cociente está en A. Si P es no negativo, es el resto. En cualquier otro caso, necesita ser restaurado (p. e., sumar b), y después será el resto. Un ejemplo numérico se da en la Figura A.3(b). Observar que el signo de P se debe comprobar antes de desplazar, ya que el bit de signo puede perderse con el desplazamiento. Sin embargo, debido a la aritmética en complemento a dos (explicada en la última sección), el resultado neto del desplazamiento seguido por la operación adecuada de suma/resta será el valor correcto. Esto sucede porque el resultado de cada caso es un número r tal que $|r| \leqslant b$.

Si a y b son números sin signo en el rango $0 \leqslant a,b \leqslant 2^n - 1$, entonces el multiplicador de la Figura A.2 funcionará si el registro P tiene n bits de longitud. Sin embargo, para la división, P debe ampliarse a $n + 1$ bits con el fin de detectar el signo de P. Por tanto, el sumador también debe tener $n + 1$ bits.

(a)

P	A	
00000	1110	Divide 14 = 1110 por 3 = 11.B siempre contiene 0011
00001	110	paso (1): desplaza
-00011		paso (2):resta
-00010	1100	paso (3): resultado es negativo, pone bit de cociente a 0
00001	1100	paso (4): restaura
00011	100	paso (1): desplaza
-00011		paso (2): resta
00000	1001	paso (3): resultado es no negativo, pone bit de cociente a 1
00001	001	paso (1): desplaza
-00011		paso (2): resta
-00010	0010	paso (3): resultado es negativo, pone bit de cociente a 0
00001	0010	paso (4): restaura
00010	010	paso (1): desplaza
-00011		paso (2): resta
-00001	0100	paso (3): resultado es negativo, pone bit de cociente a 0
00010	0100	paso (4): restaura. El cociente es 0100 y el resto es 00010.

(b)

00000	1110	Divide 14 = 1110 por 3 = 11.B siempre contiene 0011
00001	110	paso (1b): desplaza
+11101		paso (2b): resta *b* (suma complemento a 2)
11110	1100	paso (3): P es negativo, así pone bit de cociente a 0
11101	100	paso (1a): desplaza
+00011		paso (2a): suma *b*
00000	1001	paso (3): P es no negativo, así pone bit de cociente a 1
00001	001	paso (1b): desplaza
+11101		paso (2b): resta *b*
11110	0010	paso (3): P es negativo, así pone bit de cociente a 0
11100	010	paso (1a): desplaza
+00011		paso (2a): suma b
11111	0100	paso (3): P es negativo, así pone bit de cociente a 0
+00011		resto es negativo, así hacer paso final de restauración
00010		El cociente es 0100 y el resto es 00010

FIGURA A.3 Ejemplo numérico de (a) división con restauración y (b) división sin restauración.

¿Por qué nadie implementa la división con restauración, que utiliza el mismo hardware que la división sin restauración (el control es ligeramente diferente), pero involucra una suma extra? En efecto, la implementación habitual de la división con restauración no realiza literalmente una suma en el paso 4. En su lugar se examinan el signo resultante de la resta, y sólo si la suma es no negativa se vuelve a cargar en el registro P.

Como punto final, antes de comenzar a dividir, el hardware debe comprobar si el divisor es cero.

Números con signo

Hay cuatro métodos normalmente utilizados para representar números de n bits con signo: *signo y magnitud, complemento a dos, complemento a uno* y *polarizado* (*biased*). En el sistema de signo-magnitud, el bit de orden superior es el bit de signo, y los $n - 1$ bits de orden inferior son la magnitud del número. En el sistema de complemento a dos, un número y su negativo suman 2^n. En el complemento a uno, el negativo de un número se obtiene complementando cada bit. En un sistema polarizado, se toma una polarización fija para que la suma de la polarización y el número que se está representando sea siempre no negativa. Un número se representa primero sumándolo a la polarización y, después, codificando la suma con un número ordinario sin signo.

Ejemplo

¿Cuánto es −3 expresado en cada uno de estos formatos?

Respuesta

La representación binaria de 3 es 0011_2. En magnitud y signo, $-0011 = 1011$. En complemento a dos $0011_2 + 1101_2 = 8$, así $-0011 = 1101$. En complemento a uno, $-0011 = 1100$. Utilizando una polarización de 8, 3 se representa por 1011, y −3 por 0101.

El sistema más ampliamente utilizado para representar enteros, el complemento a dos, es el sistema que utilizaremos aquí; el complemento a uno se explica en los Ejercicios. Una razón de la popularidad del complemento a dos es que la suma es extremadamente sencilla: sencillamente se descarta el acarreo de salida del bit de orden superior. Para sumar $5 + -2$, por ejemplo, sumamos 0101 y 1110 para obtener 0011, dando como resultado el valor correcto de 3. Una fórmula útil para el valor de un número en complemento a dos $a_{n-1}a_{n-2}...a_1a_0$ es

A.2.3

$$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_12^1 + a_0$$

El *desbordamiento* (*overflow*) se presenta cuando el resultado de la operación no cabe en la representación que se está utilizando. Por ejemplo, si los números sin signo se representan utilizando cuatro bits, entonces $6 = 0110_2$, y $11 = 1011_2$. Su suma (17) desborda porque su equivalente binario (10001_2) no cabe en cuatro bits. Para números sin signo, detectar el «desbordamiento» es fácil; se presenta justo cuando hay un acarreo de salida del bit más significativo. Para el complemento a dos, las cosas son más complicadas: el desbor-

damiento se presenta, exactamente, cuando el acarreo de entrada del bit de orden superior es diferente del acarreo de salida (que es descartado) del bit de orden superior. En el ejemplo anterior de 5 + −2, se acarrea un 1 dentro y fuera del bit de más a la izquierda, evitando el «desbordamiento».

La negación de un número en complemento a dos, involucra complementar cada bit y después sumar 1. Por ejemplo, para negar 0011, el complemento da 1100 y, después, al sumar 1 se obtiene 1101. Por tanto, para implementar $a - b$ utilizando un sumador, basta conectar a y $\overline{b}$ (donde $\overline{b}$ es el número obtenido al complementar cada bit de b) al sumador, y poner a 1 el bit de acarreo de entrada de orden inferior. Esto explica por qué el sumador de más a la derecha en la Figura A.1 es un sumador completo.

Multiplicar números en complemento a dos no es tan simple como sumarlos. La aproximación obvia es convertir ambos operandos a no negativos, hacer una multiplicación sin signo, y después (si los operandos originales tenían signos opuestos) negar el resultado. Aunque esto es conceptualmente simple, requiere tiempo y hardware extra. Hay una mejor aproximación: suponer que estamos multiplicando a por b usando el hardware mostrado en la Figura A.2(a). El registro A se carga con el número a; el B se carga con b. Como el contenido del registro B es siembre b, utilizaremos B y b de manera indistinta. La primera cosa a hacer, cuando se multiplican números en complemento a dos, es asegurar que cuando se desplace P, se desplace aritméticamente; es decir, el bit desplazado en el bit de orden superior de P, debería ser el bit de signo de P. Observar que nuestro sumador de n bits de ancho ahora sumará números de n bits en complemento a dos entre -2^{n-1} y $2^{n-1} - 1$.

A continuación, supongamos que a es negativo. El método para manipular este caso se denomina *recodificación de Booth*. Este método es una técnica muy básica en aritmética de computadores y juega un papel clave en la Sección A.9. Observar que multiplicar por 0111_2 es lo mismo que multiplicar por $1000_2 - 1$. Para realizar esta multiplicación, restar b del registro P en el primer ciclo de la multiplicación. Sumar cero en el segundo y tercer ciclos. En el cuarto ciclo, sumar b. Para aplicar esta técnica a un multiplicador negativo como $-4 = 1100_2$, pensar como si fuese un número sin signo y escribirlo como $10000_2 - 0100_2$. Si el algoritmo de la multiplicación sólo involucra n pasos ($n = 4$ en este caso), se ignora el término 10000_2, y finalizamos restando $0100_2 = 4$ veces el multiplicador —exactamente la respuesta correcta. La ventaja de la recodificación de Booth es que funciona igualmente bien para multiplicadores positivos y negativos. Para tratar con valores negativos de a, entonces, todo lo que se requiere es restar «a» veces b de P, en lugar de sumar b o 0 a P. Aquí se dan las reglas precisas: si el contenido inicial de A es $a_{n-1}...a_0$, entonces en el paso iésimo de la multiplicación, el bit de orden inferior del registro A, es a_i, y

1. Si $a_i = 0$ y $a_{i-1} = 0$ entonces suma 0
2. Si $a_i = 0$ y $a_{i-1} = 1$ entonces suma B
3. Si $a_i = 1$ y $a_{i-1} = 0$ entonces resta B
4. Si $a_i = 1$ y $a_{i-1} = 1$ entonces suma 0

Para el primer paso, cuando $i = 0$, tomar a_{i-1} igual a 0.

Ejemplo

Cuando se multiplica −6 por −5, ¿cuál es la secuencia de valores en el par de registros (P,A)?

Respuesta

Inicialmente, P es cero y A contiene $-6 = 1010_2$. De la Figura A.4, en el primer paso, se suma 0 a P dando (P,A) = 0000 1010. Después, desplazar (P,A) = 0000 0101. En el siguiente paso, la Figura A.4 muestra que 0101 se suma a P, dando (P,A) = 0101 0101. Continuando (P,A) = 0010 1010, 1101 1010, 1110 1101, 0011 1101, y finalmente 0001 1110.

Los cuatro casos anteriores se pueden redefinir diciendo que en el paso iésimo se debe sumar $(a_{i-1} - a_i)$B a P. Con esta observación, es fácil verificar que estas reglas funcionan, porque el resultado de toda la suma es

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0)$$

De la Ecuación A.2.3, la cantidad entre paréntesis es el valor de A interpretado como un número en complemento a dos.

La forma más simple de implementar las reglas para la recodificación de Booth es ampliar el registro A un bit a la derecha para que este nuevo bit contenga a_{i-1}. De forma distinta al método sencillo de invertir cualquier operando negativo, esta técnica no requiere pasos extra o ningún caso especial para operandos negativos. Tiene solamente una lógica de control, ligeramente más complicada. Si el multiplicador se está compartiendo con un divisor, existirá ya la capacidad de restar *b*, en lugar de sumarlo. Resumiendo, un sencillo método para manipular la multiplicación en complemento a dos es prestar atención al signo de P cuando se desplaza a la derecha, y guardar el bit de A más recientemente desplazado para utilizarlo, al decidir si sumar o restar *b* de P.

La razón para el término «recodificación» *(recoding)* es la siguiente. Considerar la representación de números utilizando 1, 0 y $\bar{1}$, donde $\bar{1}$ representa −1; como ejemplo, esto nos permite también representar (recodificar) 0111

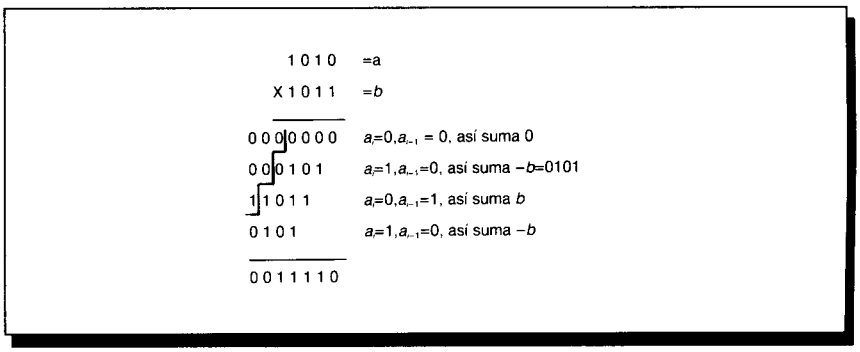


FIGURA A.4 Multiplicación de *a* = −6 por *b* = −5 para obtener 30 utilizando la recodificación de Booth. Los dígitos a la izquierda de la línea dentada son los dígitos de extensión de signo.

como $100\bar{1}$. Imaginar un algoritmo de multiplicación que funcione como sigue: poner un número recodificado en el registro A. Si el bit de orden inferior de A es 1, entonces sumar B. Si es $\bar{1}$, entonces restar B. Si el bit de orden inferior es 0, entonces sumar 0. Este algoritmo imaginario tiene exactamente el mismo efecto que el método de recodificación de Booth dado anteriormente.

La recodificación de Booth es habitualmente el mejor método para diseñar hardware que opere sobre números con signo. Sin embargo, para una implementación no directamente en hardware, realizar el método de recodificación de Booth en software o microcódigo es, habitualmente, más lento, debido a los test y saltos. Si el hardware soporta desplazamientos aritméticos (para que b negativo sea manipulado correctamente), entonces se puede utilizar el siguiente método. Considerar el multiplicador a como si fuese un número sin signo, y realizar $n - 1$ pasos de multiplicación. Si $a < 0$ (en cuyo caso habrá un 1 en el bit de orden inferior del registro A en este punto), entonces restar b de P; si no ($a \geq 0$), no sumar ni restar nada. En cualquiera de los dos casos, hacer un desplazamiento final (para un total de n desplazamientos) para colocar el bit de orden inferior del producto, en la posición de orden inferior de A. Esto funciona porque el resultado será el producto de b por $-a_{n-1}2^{n-1} + ... + a_1 2 + a_0$, que es el valor de $a_{n-1}...a_0$ como un número en complemento a dos según la Ecuación A.2.3. Si el hardware no soporta desplazamientos aritméticos, entonces el mejor enfoque quizá sea convertir los operandos para que sean no negativos.

Dos observaciones finales: una buena forma para examinar una rutina de multiplicación con signo es intentar $-2^{n-1} \times -2^{n-1}$, ya que éste es el único caso que produce un resultado de $2n - 1$ bits. De forma distinta a la multiplicación, la división habitualmente se realiza en hardware convirtiendo los operandos a no negativos y después haciendo una división sin signo; como la división es sustancialmente más lenta (y menos frecuente) que la multiplicación, el tiempo extra utilizado para manipular los signos tiene menos impacto que en la multiplicación.

Aspectos del sistema

Cuando se diseña un repertorio de instrucciones, hay una serie de cuestiones relativas a la aritmética entera que es necesario resolver. Algunas de ellas se explican aquí.

Primero, ¿qué debe hacerse con el desbordamiento de enteros? Esta situación se complica por el hecho de que detectar el desbordamiento es diferente dependiendo que los operandos sean enteros con signo o sin signo. Consideremos primero la aritmética con signo. Hay tres enfoques: poner a 1 un bit de desbordamiento, causar un trap en caso de desbordamiento, o no hacer nada con el desbordamiento. En el último caso, el software tiene que comprobar si se ha presentado o no desbordamiento. La solución más conveniente para el programador es tener un bit de habilitación. Si este bit se activa, entonces el desbordamiento produce un trap. Si se desactiva, entonces el desbordamiento pone a 1 un bit. La ventaja de este enfoque es que ambas operaciones con trap o sin trap requieren solamente una instrucción. Además, como veremos en la Sección A.7, esto es análogo a la forma en que el estándar de punto flotante

del IEEE maneja el desbordamiento de punto flotante. La Figura A.5 muestra cómo algunas máquinas comunes tratan el desbordamiento.

¿Qué ocurre con la suma sin signo? Observar que ninguna de las arquitecturas de la Figura A.5 causa un trap en caso de desbordamiento sin signo. Es conveniente poder restar, de una dirección sin signo, sumando. Por ejemplo, cuando $n = 4$, podemos restar 2 de la dirección sin signo $10 = 1010_2$ sumando $14 = 1110_2$. Aun cuando $1010_2 + 1110_2$ suma la respuesta que queremos ($1000_2 = 8$), esta operación tiene un desbordamiento sin signo. En otras palabras, las direcciones se tratan como números con signo y sin signo, haciendo un trap de desbordamiento inservible para el cálculo con direcciones.

Una segunda cuestión está relacionada con la multiplicación. El resultado de la multiplicación de dos números de n bits ¿deberá ser de $2n$ bits, o deberá devolver los n bits de orden inferior, señalando desbordamiento si el resultado sobrepasa esos n bits? El argumento en favor de un resultado de n bits es que, virtualmente en todos los lenguajes de alto nivel, la multiplicación es una operación cuyos argumentos son variables enteras y cuyo resultado es una variable entera del mismo tipo. Por tanto, no hay forma de generar código que utilice un resultado en doble precisión. El argumento en favor de un resultado de $2n$ bits es que lo puede utilizar una rutina, en lenguaje ensamblador, para acelerar sustancialmente la multiplicación de enteros en múltiple precisión (en aproximadamente un factor de 3).

Una tercera cuestión concierne a las máquinas que en cada ciclo quieren ejecutar una instrucción. Raramente es práctico realizar una multiplicación o división en la misma cantidad de tiempo que una suma o una transferencia registro a registro. Hay tres posibles aproximaciones a este problema. La pri-

Máquina	¿Trap en desbordamiento con signo?	Trap en desbordamiento sin signo?	¿Pone a 1 bit en desbordamiento con signo?	¿Pone a 1 bit en desbordamiento sin signo?
VAX	Si habilitación está activa	No	Sí. ADD pone a 1 bit V	Sí. ADD pone a 1 bit C
IBM 370	Si habilitación está activa	No	Sí. ADD pone a 1 código de cond	Sí. ADD lógica pone a 1 código de cond
Intel 8086	No	No	Sí. ADD pone a 1 bit V	Sí. ADD pone a 1 bit C
MIPS R3000	Hay 2 instrucciones ADD: una siempre interrumpe, la otra nunca	No	No. El software debe deducirlo del signo de los operandos y resultado	
SPARC	No	No	ADDCC pone a 1 bit V ADD no	ADDCC pone a 1 bit C ADD no

FIGURA A.5 Resumen de cómo varias máquinas manipulan el desbordamiento entero. Tanto el 8086 como el SPARC tienen una instrucción que interrumpe si el bit V está a 1; así, el coste de interrumpir en caso de desbordamiento es una instrucción extra.

mera es tener una instrucción de *paso de multiplicación* de un solo ciclo. Esta podría realizar un paso del algoritmo de Booth. La segunda aproximación es hacer multiplicaciones enteras en la unidad de punto flotante y hacer que ésta sea parte del repertorio de instrucciones de punto flotante. (Esto es lo que hace DLX.) La tercera aproximación es tener una unidad autónoma en la CPU para hacer la multiplicación. En este caso se puede garantizar que el resultado estará disponible en un número fijo de ciclos —y el compilador se encarga de esperar el tiempo adecuado— o puede haber un interbloqueo. Los mismos comentarios se aplican también a la división. Como ejemplos, la SPARC tiene una instrucción de paso de multiplicación, pero no tiene instrucción de paso de división, y el MIPS R3000 tiene una unidad autónoma que hace multiplicaciones y divisiones (ver Sección E-6 para nuevas ampliaciones para la aritmética en SPARC). Los diseñadores de HP Precisión Architecture hicieron una trabajo especialmente minucioso de análisis de la frecuencia de los operandos para multiplicación y división, y diseñaron sus pasos de multiplicación y división adecuadamente. (Ver Magenheimer y cols. [1988] para detalles.)

Una pifia potencial que merece la pena mencionar está relacionada con la suma en múltiple precisión. Muchos repertorios de instrucciones ofrecen una variante de la instrucción ADD que suma tres operandos: dos números de n bits junto con un tercer número de un solo bit. Este tercer número es el acarreo de la suma anterior. Como el número en múltiple precisión, normalmente, se almacenará en un array, es importante poder incrementar el puntero del array sin destruir el bit de acarreo.

A.3 Punto flotante

Introducción

Muchas aplicaciones requieren números que no son enteros. Hay una serie de formas en las que pueden representarse los no enteros. Una es utilizar *punto fijo*; es decir, utilizar aritmética entera e imaginar simplemente el punto binario en algún sitio distinto a la derecha del dígito menos significativo. Sumar dos de tales números puede hacerse mediante una suma entera, mientras que la multiplicación requiere algún desplazamiento extra. Otras representaciones que se han propuesto involucran el almacenamiento del logaritmo de un número y realizan la multiplicación sumando los logaritmos, o utilizan un par de enteros (a,b) para representar la fracción a/b. Sin embargo, sólo hay una representación no entera cuyo uso se ha extendido ampliamente, y es la representación en punto flotante. En este sistema, una palabra del computador se divide en dos partes, un exponente y una mantisa. Como ejemplo, un exponente de -2 y una mantisa de 1,5 puede representar el número $1{,}5 \times 2^{-2} = 0{,}375$. Las ventajas de estandarizar una representación determinada son obvias. Los analistas numéricos pueden construir bibliotecas de software de alta calidad, los diseñadores de computadores pueden desarrollar técnicas para implementaciones hardware de alto rendimiento, y los vendedores de hardware pueden construir aceleradores estándares. Dado lo predominante

de la representación en punto flotante, parece poco probable que cualquier otra representación tenga un uso tan extendido.

Un hecho clave con respecto a las instrucciones de punto flotante, es que su semántica no es tan clara como la semántica del resto del repertorio de instrucciones, y en el pasado el comportamiento de las operaciones de punto flotante variaba considerablemente de una familia de computadores a la siguiente. Las variaciones involucraban cosas tales como: el número de bits ubicados en el exponente y mantisa, el rango de los exponentes, cómo se realizaba el redondeo, y las acciones tomadas en condiciones excepcionales como desbordamiento a cero *(underflow)* y desbordamiento *(overflow)*. Los libros de arquitectura de computadores solían contener información sobre cómo tratar todos estos detalles, pero afortunadamente esto no es necesario por más tiempo. Ello se debe a que la industria de computadores está convergiendo rápidamente hacia el formato especificado por el estándar 754-1985 del IEEE. Las ventajas de utilizar una variante estándar del punto flotante son similares a las de utilizar punto flotante sobre otras representaciones no enteras. En este capítulo explicaremos sólo la versión de punto flotante del IEEE. Para lecturas adicionales ver IEEE [1985], Cody y cols. [1984], Cody [1988] y Goldberg [1989].

Visión general del estándar del IEEE

Probablemente, la característica más notable del estándar sea que requiere que el cálculo continúe en caso de condiciones excepcionales, tal como dividir por cero o determinar la raíz cuadrada de un número negativo. El resultado de calcular la raíz cuadrada de un número negativo es *NaN* (*N*o un *N*úmero), un patrón de bits que no representa un número ordinario. Como ejemplo de cómo los NaN pueden ser útiles, considerar el código para un determinador de ceros que tome una función F como argumento y evalúe F en distintos puntos para determinar un cero de la función. Si el determinador de ceros indaga, accidentalmente, fuera de los valores válidos para F, F puede provocar una excepción. Escribir un determinador de ceros que trate este caso es altamente dependiente del lenguaje y del sistema operativo, porque se basa en cómo reacciona el sistema operativo ante excepciones y cómo esta reacción se corresponde con el lenguaje de programación. En aritmética del IEEE es fácil sintetizar un determinador de ceros que manipule esta situación y corra en muchos sistemas diferentes. Después de cada evaluación de F, simplemente comprueba si F ha devuelto un NAN; si es así, sabe que ha indagado fuera del dominio de F.

Debido a las reglas para realizar aritmética con NaN, escribir subrutinas en punto flotante que puedan aceptar NaN como un argumento raramente requiere comprobaciones de casos especiales. Suponer que «arccos» se calcula en función de «arctan», utilizando la fórmula $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$. Si «arctan» manipula un argumento de NaN adecuadamente, «arccos» automáticamente lo hará también. Esto es porque el estándar del IEEE especifica que cuando un argumento de una operación es un NaN, el resultado debe ser un NaN. Por tanto si x es un NaN, $1 + x$, $1 - x$, $(1 + x)/(1 - x)$ y $\sqrt{(1-x)/(1+x)}$ también serán NaN. No se requiere comprobar los NaN.

Mientras que el resultado de $\sqrt{-1}$ es un NaN, el resultado de 1/0 no es un NaN, sino $+\infty$, que es otro valor especial. El estándar define aritmética en el infinito (incluyendo $-\infty$) utilizando reglas tal como $1/\infty = 0$. La fórmula $\arccos x = 2 \arctan(\sqrt{(1 - x)/(1 + x)}$ ilustra cómo puede utilizarse la aritmética del infinito. Como $\arctan x$ se aproxima asintóticamente a $\pi/2$ cuando x se aproxima a ∞, es natural definir $\arctan(\infty) = \pi/2$, en cuyo caso $\arccos(-1)$, automáticamente, se calculará, de forma correcta, como $2 \arctan(\infty) = \pi$.

Otra característica del estándar del IEEE con implicaciones para el hardware es la regla del redondeo. Cuando se opera sobre dos números en punto flotante, el resultado es habitualmente un número que no se puede representar exactamente como otro número en punto flotante. Por ejemplo, en un sistema de punto flotante que utiliza base 10 y dos dígitos significativos, $2{,}1 \times 0{,}5 = 1{,}05$. Esto necesita redondearse a dos dígitos. ¿Deberá redondearse a 1,0 ó 1,1? En el estándar del IEEE, estos casos en los que el resultado está justo a medio camino entre dos números representables se redondean al número cuyo dígito de orden inferior es par. Es decir, 1,05 se redondea a 1,0 y no a 1,1. El estándar realmente tiene cuatro *modos de redondeo*. El implícito es el *redondeo al más próximo*, que redondea a un número par en el caso de empate. Los otros modos son redondeo hacia cero, redondeo hacia $+\infty$ y redondeo hacia $-\infty$.

El estándar especifica cuatro precisiones: *simple, extendida simple, doble* y *extendida doble*. Las propiedades de estas precisiones se resumen en la Figura A.6. No se requiere que las implementaciones tengan las cuatro precisiones, pero se recomienda que soporten o bien la combinación de simple y simple extendida o la simple, doble y doble extendida. Consideremos la precisión simple con más detalle. Los números en simple precisión se representan utilizando 32 bits: 1 para el signo, 8 para el exponente y 23 para la fracción. El exponente es un número con signo que se representa, utilizando el método de la polarización (como se explicó en la Sección A.2 anterior), con una polarización de 127. Siempre utilizaremos el término *campo de exponente* para indicar el número sin signo contenido en los bits uno a nueve y *exponente* para indicar la potencia a la cual debe elevarse dos. (En el estándar estos se denominan «exponente polarizado» y «exponente no polarizado», respectivamente.) La fracción representa un número menor que uno, pero la *mantisa* del número, en punto flotante, es uno más la parte fraccionaria. En otras palabras, si e es el valor del campo de exponente y f es el valor del campo fraccionario, el número se representa como $1.f \times 2^{e-127}$.

Ejemplo

¿Qué número en simple precisión representa la siguiente palabra de 32 bits?

```
1  10000001  01000000000000000000000
```

Respuesta

Considerado como un número sin signo, el campo del exponente es 129, calculando el valor del exponente $129 - 127 = 2$. La parte fraccionaria es $0{,}01_2 = 0{,}25$, siendo la mantisa 1,25. Por tanto, este patrón de bits representa el número $-1{,}25 \times 2^2 = -5$.

	Simple	Simple extendido	Doble	Doble extendido
p (bits de precisión)	24	$\geqslant 32$	53	$\geqslant 64$
$E_{máx}$	127	$\geqslant 1\,023$	1 023	$\geqslant 16\,383$
$E_{mín}$	−126	$\leqslant -1\,022$	−1 022	$\leqslant -16\,382$
Polarización de exponente	127		1 023	

FIGURA A.6 Parámetros de formato para el estándar de punto flotante del IEEE 754. La primera fila da el número de bits en la mantisa. Los espacios en blanco son parámetros no especificados.

La parte fraccionaria de un número en punto flotante (0,25 en el ejemplo anterior) no debe ser confundida con la mantisa, que es uno más la parte fraccionaria. El 1 delantero en la mantisa $1.f$ no aparece en la representación; es decir, el bit del comienzo es implícito. Cuando se realiza aritmética en números con formato IEEE, la parte fraccionaria normalmente necesita estar *desempaquetada* (*unpacked*), lo que significa que el uno implícito necesita hacerse explícito.

En la Figura A.6, el rango de los exponentes para simple precisión varía de −126 a 127; el campo de exponente varía de 1 a 254. Los campos de exponente de 0 y 255 se utilizan para representar valores especiales. Cuando el campo de exponente es 255, un campo fraccionario de cero representa infinito, y un campo fraccionario de no cero representa un NaN. Por tanto, hay una familia completa de NaN. Cuando los campos exponente y fraccionario son cero, entonces el número representado es cero. Como los números ordinarios siempre tienen una mantisa mayor o igual que 1 —y por tanto nunca son cero— se requiere un convenio especial, como éste, para representar el cero.

Un campo de exponente cero y una parte fraccionaria no cero representan un número *denormal*, aunque a veces se denomina número *subnormal*. Estos números son la parte más controvertida del estándar. Más tarde, en la discusión sobre la multiplicación, veremos por qué son difíciles de implementar en hardware. En muchos sistemas de punto flotante si $E_{mín}$ es el exponente más pequeño, un número menor que $1{,}0 \times 2^{E_{mín}}$ no se puede representar, y una operación de punto flotante que dé como consecuencia un número menor que éste, simplemente se pone a cero. Por otro lado, en el estándar del IEEE los números menores que $1{,}0 \times 2^{E_{mín}}$ son representados desplazando su parte fraccionaria a la derecha. Esto se denomina *desbordamiento a cero gradual*. Por tanto, cuando los números decrecen en magnitud por debajo de $2^{E_{mín}}$, gradualmente pierden su significando y sólo se representan por cero cuando su significado se ha desplazado fuera. Por ejemplo, en base 10 con 4 números significativos, sea $x = 1{,}234 \times 2^{E_{mín}}$. Entonces $x/10 = 0{,}123 \times 10^{E_{mín}}$, habiendo perdido un dígito de precisión; $x/100$ y $x/1000$ tiene aún menos precisión, mientras que $x/10000$ es finalmente suficientemente pequeño para que se redondee a cero. Los números denormalizados se implementan haciendo que una palabra con campo de exponente cero represente el núme-

ro $0.f \times 2^{E_{min}}$. Una de las ventajas del desbordamiento gradual es que cuando se utiliza, si $x \neq y$, entonces $x - y \neq 0$. En un sistema de redondeo a cero, esto no es siempre cierto.

La razón principal por la que el estándar del IEEE, igual que muchos otros formatos de punto flotante, utiliza exponentes polarizados es para significar que los números no negativos están ordenados en la misma forma que los enteros. Es decir, la magnitud de los números en punto flotante se puede comparar utilizando un comparador de enteros. Otra ventaja (relacionada) es que el cero se representa por una palabra llena de ceros. La otra cara de los exponentes polarizados es que sumarlos es ligeramente difícil, porque requiere que la polarización se reste de su suma.

Cuando el estándar del IEEE esté más extendido, será más fácil portar software y escribir bibliotecas portables que traten las excepciones de punto flotante. Pero el estándar también tiene algunos inconvenientes:

1. Originalmente, fue pensado para microprocesadores, por tanto a los requerimientos de las implementaciones de alto rendimiento no se les dio elevada prioridad.

2. El estándar contiene partes opcionales. Esto da como resultado difíciles decisiones para los implementadores —¿qué partes deben implementarse?— y para los escritores de software portable —¿deberían evitar utilizar alguna de las partes opcionales del estándar?

3. El desbordamiento a cero gradual, habitualmente, se ha implementado en una forma que es órdenes de magnitud más lento que el redondeo a cero, así que los usuarios lo inhabilitan con frecuencia.

4. Todavía no hay series de test de punto flotante del IEEE de dominio público y de peso en el mundo industrial.

Aunque el estándar pueda acabar mejorando la calidad de las bibliotecas de punto flotante, esto está por ocurrir, todavía, debido a la gran base de VAX, IBM/370 y Cray, así como al hecho de que no hay estándar correspondiente sobre cómo acceder a sus características en software. Por otro lado, tanto DEC como IBM han introducido, recientemente, máquinas que usan aritmética del IEEE.

Algunos comentarios finales sobre el estándar:

1. De forma distinta a muchos estándares, el IEEE 754 no ratifica ni refina ningún sistema existente. Aunque la mayoría de las características del estándar aparecen como mínimo en uno de los anteriores sistemas de computadores, es sustancialmente diferente de lo que fue práctica actual en la época.

2. El estándar no dice nada sobre la aritmética entera ni sobre las funciones transcendentes (sen, cos, exp, etc.). En particular, no dice nada sobre la precisión que deberían tener las funciones transcendentes, y no dice nada sobre los valores excepcionales de estas funciones, tal como 0^0.

3. Usualmente, un **sistema** computador —es decir, una combinación de hardware y software— implementará el estándar. Por tanto, no hay nada

erróneo con diseñar hardware que no implemente completamente el estándar, mientras haya alguna forma de suministrar por software lo que el hardware no hace. En efecto, el mejor diseño puede hacer que los casos menos frecuentes sean manipulados por software.

A.4 Suma en punto flotante

Hay dos diferencias entre la aritmética en punto flotante y la aritmética entera: debe manejarse un campo de exponente, además del campo fraccionario, y el resultado de una operación en punto flotante, habitualmente, se ha de redondear con el fin de que sea representado por otro número en punto flotante de la misma precisión.

Redondeo

El estándar del IEEE especifica que el resultado de una operación aritmética sería el mismo que si se calculase exactamente y después se redondease utilizando el modo actual de redondeo. El modo más difícil de implementar es el modo implícito —redondear hacia el valor más cercano (y redondear los casos intermedios al par). La aproximación sencilla para cumplir con el estándar del IEEE es calcular exactamente la suma y después redondear. Esto sería bastante caro, ya que requeriría un gran sumador. Para ver cómo satisfacer el estándar con menos hardware, consideremos algunos ejemplos.

Hay dos formas de redondeo que se pueden presentar durante la suma. Con fines de ilustración utilizaremos la base 10, que es más natural para los humanos, y tres dígitos significativos. El primer caso requiere redondeo debido al acarreo de salida a la izquierda, como se ilustra en la Figura A.7(a). El segundo caso requiere redondeo debido a exponentes desiguales, como en la Figura A.7(b). La Figura A.7(c) muestra que es posible que ambas situaciones ocurran simultáneamente. En cada uno de estos casos, la suma se debe calcular con más de tres posiciones, con el fin de realizar el redondeo. En un caso —cuando se restan números próximos, como en la Figura A.7(d)— la suma se debe calcular con más de tres posiciones, aun cuando no se realice redondeo. Ignorando temporalmente el requerimiento de redondeo al par, cada uno de estos ejemplos se puede implementar con un sumador de cuatro dígitos (es decir, utilizando un dígito adicional). Por tanto, en la Figura A.7(b) el 6 más a la derecha de 2,56, sencillamente se puede despreciar antes de sumar. Pero hay un caso, mostrado en la Figura A.7(e), en el cual cuatro dígitos no son suficientes. Si se desplazase el dígito de orden inferior de .0376, la respuesta habría sido 0,973 en lugar de 0,972. Sin embargo, es fácil comprobar (sin hacer caso al redondeo al par) que dos dígitos extra son siempre suficientes. Estos dígitos extra se denominan *dígitos de guarda y redondeo.*

La regla de redondeo al par introduce una complicación extra. La Figura A.7(f) muestra un ejemplo con cinco dígitos significativos. Puede parecer, en principio, que se necesita mantener doble número de dígitos para

realizar un redondeo al par, ya que el 1 más a la derecha de 2,5001 determina si el resultado es 4,5676 ó 4,5677.

Después de una pequeña reflexión se puede ver que sólo es necesario saber si hay o no hay dígitos distintos de cero pasadas las posiciones de guarda y redondeo. Esta información se puede almacenar en un solo bit denominado, habitualmente, *bit retenedor* (*sticky bit*), que se implementa examinando cada dígito que es despreciado debido a un desplazamiento. Tan pronto como aparece un dígito distinto de cero, el bit retenedor se pone a 1 y permanece con ese valor. Para implementar el redondeo al par, añadir, sencillamente, el bit retenedor a la derecha del dígito de redondeo justo antes de redondear.

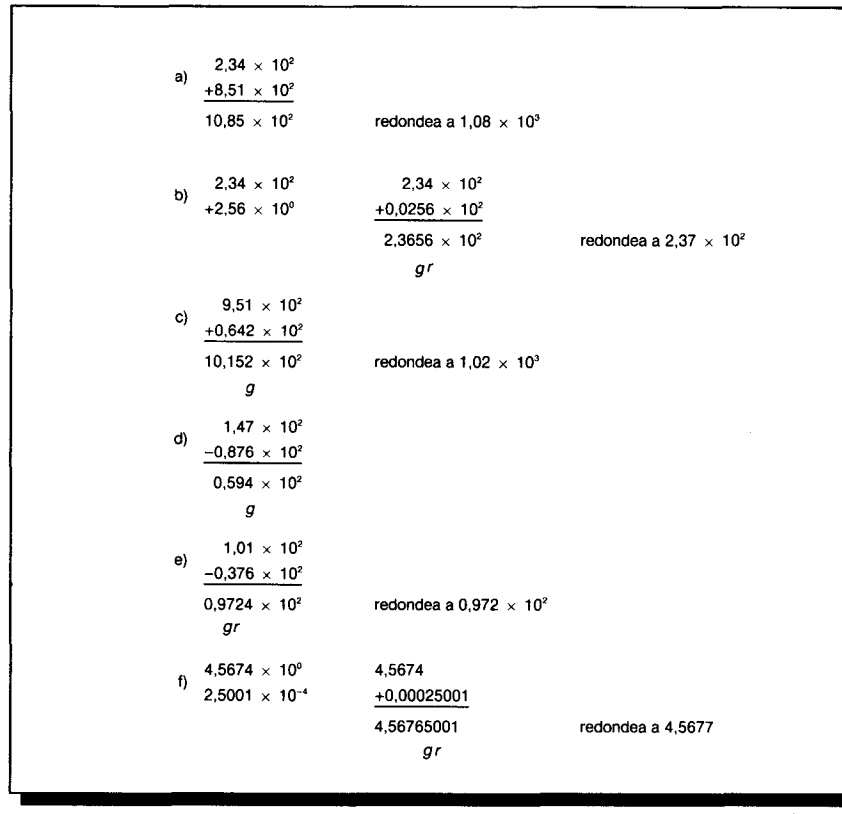


FIGURA A.7 Ejemplos de redondeo. En (a) hay redondeo debido al acarreo de salida a la izquierda y en (b) debido a los exponentes desiguales, mientras que en (c) ocurren ambas cosas. El ejemplo (d) muestra que una posición extra debe encontrarse aun cuando no haya redondeo, mientras que (e) muestra la situación en la cual se necesitan dos dígitos extra. Finalmente (f), donde $p = 5$, ilustra por qué es necesario un bit retenedor para realizar redondeo al par. Las letras g y r se colocan bajo los dígitos de guarda y redondeo.

El algoritmo de la suma

Las notaciones e_i y s_i se utilizan aquí para los campos del exponente y mantisa del número en punto flotante a_i. Esto significa que el número a_i se ha desempaquetado y que s_i tiene un bit explícito delante. El procedimiento básico para sumar dos números en punto flotante a_1 y a_2 es sencillo y consta de cinco pasos.

1. Si $e_1 < e_2$, intercambiar los operandos para que la diferencia de los exponentes satisfaga $d = e_1 - e_2 \geqslant 0$. Provisionalmente, poner el exponente del resultado en e_1.

2. Desplazar s_2 $d = e_1 - e_2$ lugares a la derecha. Dicho con más precisión, poner s_2 en un registro de p bits y después extender el registro MIN(2,d) bits a la derecha. Desplazar s_2 d lugares a la derecha. Si $d > 2$, el bit retenedor es la OR lógica de los $d - 2$ bits que son desplazados fuera del registro extendido. De los dos bits extendidos, el más significativo es el bit de guarda; el menos significativo es el bit de redondeo.

3. Añadir el bit retenedor a s_2, y después sumar los dos campos de fracción en signo y magnitud en un sumador de $p + 3$ bits. Llamar a esta suma preliminar S.

4. Si hubiese acarreo del lugar más significativo, en el paso anterior, desplazar una posición a la derecha la magnitud de S. En otro caso, desplazarla a la izquierda hasta que se normalice. Ajustar adecuadamente el exponente del resultado. El bit de redondeo ahora se pone como el bit $(p + 1)$ésimo de la magnitud de S, y el bit retenedor como la OR lógica de todos los bits a la derecha del bit de redondeo.

5. Redondear el resultado utilizando la Figura A.8. Si una entrada de la tabla es no vacía, sumar 1 a la magnitud de S. Así, si $S \geqslant 0$, se deberá calcular $S + 1$; en cualquier otro caso $S - 1$.

Los bits de guarda y redondeo, antes del desplazamiento, están marcados en cada uno de los ejemplos de la Figura A.7.

Ejemplo

Mostrar cómo procede el algoritmo de suma sobre los operandos de la Figura A.7(f) cuando tiene lugar el redondeo al más próximo.

Respuesta

En el paso 1, $e_1 = 0 > e_2 = -3$, así $d = 3$ y no es necesario intercambio. En el paso 2, $g = 5$, $r = 0$, y el retenedor *(sticky)* es la OR de 0, 0 y 1; por consiguiente, es 1. En el paso 3 los números que se van a sumar son 4,5674 y 0,0002501, por tanto, la suma preliminarmente es $S = 4{,}5676501$. En el paso 4 no hay acarreo de salida, por tanto, d todavía es 3. El bit de redondeo es 5, y el retenedor es $1 = 0 \vee 1$. En el paso 5, al consultar la tabla nos indica que como los bits de redondeo y retenedor no son cero, debemos sumar 1 al quinto dígito de S, cambiando S de 45676 a $45676 + 1 = 45677$.

Modo de redondeo	$S \geqslant 0$	$S < 0$
$-\infty$		$+1$ si $r \vee s$
$+\infty$	$+1$ si $r \vee s$	
0		
Más próximo	$+1$ si $r \wedge \bar{s} \wedge p_0$ o $r \wedge s$	$+1$ si $r \wedge \bar{s} \wedge p_0$ o $r \wedge s$

FIGURA A.8 Reglas para implementar los modos de redondeo del IEEE. Los espacios en blanco significan que los p bits más significativos de la suma preliminar S son los bits de suma real. Si la condición especificada es cierta, sumar 1 al p-ésimo bit más significativo de S. Los símbolos r y s representan los bits de redondeo y retenedor, mientras que p_0 es el p-ésimo bit más significativo de S.

El paso 3 involucra la suma de números con signo y magnitud, y en sí mismo tiene tres pasos:

3a. Convertir cualquier número negativo a su complemento dos.

3b. Realizar una suma de $(p + 4)$ bits en complemento a dos ($p + 3$ bits de magnitud, 1 bit para el signo).

3c. Si el resultado es negativo, realizar otra complementación a dos para volver a poner el resultado en la forma de magnitud y signo.

Como se puede ver, la suma es una operación bastante complicada. Aquí se da un truco que puede acelerarla. Sólo será necesario desplazar variablemente una vez un par de números, bien en el paso 2 o en el paso 4, pero no en ambos. La razón es sencilla: si $|e_1 - e_2| > 1$, entonces el paso 4 puede necesitar un desplazamiento como mucho de una posición. Y si $|e_1 - e_2| \leqslant 1$, entonces el paso 2 obviamente requiere, como máximo, un paso de desplazamiento. Un sumador no segmentado puede explotar esto y reducir el número de pasos de cinco a cuatro. Un sumador que utilice cada uno de los pasos anteriores, como una etapa en modo segmentado, también puede utilizar esta reducción, aunque requiera duplicar desplazador y sumador.

El paso 3 puede consumir tiempo, porque puede involucrar hasta cuatro sumas: dos para negar ambos operandos (complementación a dos realizada complementando los bits, seguida por una suma de 1), una tercera para la misma suma, y después una cuarta para negar el resultado. Hay diversas formas de acelerar este paso. Ya hemos visto que se puede añadir 1 a una suma poniendo a 1 el bit de acarreo de entrada, el de orden inferior del sumador. Si ambos operandos son negativos, podemos poner sus bits de signo a cero, no olvidando negar el resultado. La suma requerida, cuando se niegue el resultado, puede combinarse con el paso de redondeo (que debe estar preparado, de todos modos, para hacer nuevamente una suma).

El paso de redondeo requiere una segunda suma de completa precisión, además de la del paso 3. Es posible combinarlas en una sola suma. Observar que al final del paso 2, se conocen los bits g, r y s; por tanto, también se sabe

si se redondea o no, sumando 1 al bit p-ésimo más significativo. Lo que no se sabe es la posición del p-ésimo bit más significativo, ya que su posición depende del resultado de la suma del paso 3; cuando se suman números del mismo signo, esa posición está determinada por el acarreo del bit más significativo. Por tanto, la forma de eliminar el paso 5 es sumar el bit de redondeo (si es necesario), como parte del paso 3. Como se desconoce la posición, deben realizarse dos versiones del paso 3 utilizando dos sumadores en paralelo. Cada sumador supone una de las dos posibilidades según la posición donde vaya el bit de redondeo. Esta técnica para reducir el número de pasos de la suma se utiliza en el Intel 860 [Kohn, 1989]. Cuando se redondea, puede surgir una complicación: la suma de 1 puede producir un acarreo de salida en el bit de orden superior. Este caso solamente ocurre cuando el valor de S es 11...11.

Números denormalizados

Hay que hacer muy pocos cambios, en la descripción anterior, si una de las entradas es un número denormal. Debe hacerse un test para ver si el campo de exponente es 0. Si es así, entonces al desempaquetar el significando no aparecerá un 1 al principio. Poniendo a 1 el campo de exponente, cuando se desempaquete un denormal, las reglas de desplazamiento en los pasos 1-5 todavía son correctas.

Con el fin de tratar con salidas denormalizadas, se debe modificar ligeramente el paso 4. El valor del registro P se desplaza a la izquierda hasta que se normalice P, o hasta que el exponente alcanza el valor $E_{\text{mín}}$ (es decir, el campo de exponente se haga 1). Si el exponente es $E_{\text{mín}}$, y si después de redondear, el bit de orden superior de P es 1, entonces el resultado es un número normalizado y se debe empaquetar de la forma usual, omitiendo el 1. Si, por el contrario, el bit de orden superior es 0, el resultado es denormal, y cuando se empaquete el resultado, el campo de exponente debe ponerse a 0.

Incidentalmente, es muy fácil detectar desbordamiento. Sólo puede ocurrir si el paso 4 involucra un desplazamiento a la derecha, y si el campo de exponente, en ese punto, aumenta hasta 255 en simple precisión (o 2047 para doble precisión), o si esto ocurre después del redondeo.

Detectar desbordamiento a cero es complicado, ya que depende de que haya manipulador de traps del usuario. El estándar del IEEE especifica que si están habilitados los manipuladres de traps del usuario, el sistema debe generar un trap si el resultado es denormal. Por otro lado, si están inhabilitados los manipuladores de traps, el señalizador de desbordamiento a cero se pone a 1 sólo si hay una pérdida de precisión —es decir, si se debe redondear el resultado. La razón para esto es que si no se pierde precisión en un desbordamiento a cero, no tiene sentido poner a 1 ningún señalizador de aviso. Pero si está habilitado un manipulador de traps, el usuario puede intentar simular el redondo a cero y, por tanto, se deberá notificar cuando un resultado caiga por debajo de $1{,}0 \times 2^{E\text{mín}}$. Esta discusión es relevante para la suma, ya que el resultado de una suma o resta de un número denormal siempre será exacto; como no se puede perder precisión en un desbordamiento a cero, no hay necesidad de poner a 1 el señalizador de desbordamiento a cero.

A.5 Multiplicación en punto flotante

La multiplicación en punto flotante es muy parecida a la multiplicación entera. Debido a que los números en punto flotante se almacenan en la forma de signo-magnitud, el multiplicador sólo necesita tratar con números sin signo (aunque hemos visto que la recodificación de Booth manipula números con signo en complemento a dos sin causar dificultades). Si las fracciones son números sin signo de p bits, entonces el producto puede tener hasta $2p$ bits y se debe redondear a un número de p bits. Además de multiplicar las partes fraccionarias, se deben sumar los campos de los exponentes, y después restar de su suma la polarización.

Aquí se da un método correcto de tratar el redondeo utilizando el multiplicador de la Figura A.2: multiplicar las dos partes fraccionarias para obtener un producto de $2p$ bits en los registros (P,A). Durante la multiplicación, las $p - 2$ primeras veces se desplaza un bit en el registro A, haciendo la operación OR con el bit retenedor. Cuando finalizan todos los pasos de la multiplicación, el bit de orden superior a A es el bit de guarda, y el bit siguiente es el de redondeo. Hay dos casos:

1. El bit de orden superior de P es 0. Desplazar P a la izquierda 1 bit, desplazando el bit g de A. Desplazar el resto de A no es necesario.

2. El bit de orden superior de P es 1. Poner $s := s \vee r$ y $r := g$, y sumar 1 al exponente.

Ahora, usar las reglas de la Figura A.8 para redondear el resultado, sumando 1 (si es necesario) al bit de orden inferior de P. La parte fraccionaria (en forma desempaquetada) está en el registro P. Recordar que la operación de redondeo puede provocar un acarreo de salida del bit más significativo. Una buena discusión de las formas más eficientes de implementar el redondeo está en Santoro, Bewick y Horowitz [1989].

Detectar desbordamiento y desbordamiento a cero es ligeramente complicado. Considerar el caso de simple precisión. Los campos de exponente se deben sumar, junto con −127. Si la suma se hace en un sumador de 10 bits, $-127 = 1110000001_2$, y se presenta desbordamiento cuando los bits de orden superior de la suma son 01 o cuando la suma es 0011111111. El desbordamiento a cero se presenta cuando los bits de orden superior son 11 o la suma es 0000000000. Alternativamente, la suma se puede hacer utilizando sólo un sumador de 8 bits. Sumar, sencillamente, ambos exponentes y $-127 = 10000001_2$. Si los bits de orden superior de los campos de exponente son diferentes, no es posible desbordamiento/desbordamiento a cero. Si los bits de orden superior son ambos 1, el resultado presenta desbordamiento si hay un 0 en el bit de orden superior o si es 1111111. Si ambos exponentes tienen cero en los bits de orden superior se presenta desbordamiento si la suma tiene el bit de orden superior igual a 1, o si la suma es 00000000.

Denormales

A partir de la descripción del algoritmo de la multiplicación, se puede ver que después de hacer una multiplicación entera sobre las fracciones, el resultado final se obtiene a lo sumo con un desplazamiento. Con los denormales, la situación cambia completamente. Suponer que la entrada está normalizada, pero que la salida es denormal, ya que en simple precisión el producto tiene un exponente e que es $e < -126$. Entonces, el resultado se debe desplazar a la derecha $-e - 126$ posiciones. Esto requiere hardware extra (un desplazador que no sería necesario en otro caso) y tiempo extra. La situación con entradas denormales no es mejor, porque incluso si el resultado final es un número normalizado, se requiere todavía un desplazamiento variable. Por tanto, los multiplicadores de punto flotante de elevado rendimiento, no manejan, con frecuencia, números denormalizados, pero en cambio generan un trap que permite al software manejarlos. Hay algunos códigos prácticos que generan muchos desbordamiento a cero, incluso cuando se trabaja adecuadamente, y estos programas, habitualmente, van un poco más lentos en sistemas que requieren que los denormales sean procesados por un manipulador de traps.

Un procedimiento seguido por algunas unidades de punto flotante es que el multiplicador tenga salidas denormalizadas en forma aproximada (*wrapped*). Es decir, la parte fraccionaria está normalizada, y el exponente se aproxima alrededor de *(wrapped around)* un gran número positivo. Este es exactamente el resultado cuando se sigue el algoritmo de la multiplicación, dado anteriormente, para números normalizados. Como la unidad de suma debe tener un desplazador, es habitualmente sencillo proporcionar una forma de convertir los números aproximados *(wrapped)* en su forma denormalizada correcta pasándolos a través del sumador. Sin embargo, si tiene que intervenir un manipulador de traps con el fin de enviar números aproximados *(wrapped)* al sumador, la multiplicación será sustancialmente retardada.

Se presentan algunos puntos interesantes cuando una multiplicación produce un número denormal. Considerar el caso de un sistema de punto flotante en base 2, con 3 bits significtativos (por consiguiente dos bits fraccionarios). El resultado exacto de $1{,}11 \times 2^{-2}$ multiplicado por $1{,}11 \times 2^{E_{\text{mín}}}$ es $0{,}110001 \times 2^{E_{\text{mín}}}$. Si el modo de redondeo es redondear hacia más infinito, el resultado redondeado es el número normal $1{,}00 \times 2^{E_{\text{mín}}}$ ¿Se debería indicar desbordamiento a cero? Señalar desbordamiento a cero significa que se está utilizando la regla de *redondear antes*, porque el resultado era denormal antes de redondear. No señalar desbordamiento a cero significa que se está utilizando la regla de *redondear después*, porque el resultado está normalizado después del redondeo. El estándar del IEEE proporciona la posibilidad de escoger una de las dos reglas; sin embargo, la escogida se debe utilizar, consistentemente, en todas las operaciones.

Como se mencionó en la sección de la suma, el manipulador de traps, si existe, se debe llamar siempre que el resultado sea denormal. Si no hay manipulador de traps, la excepción de desbordamiento a cero se señala sólo cuando el resultado sea denormal e inexacto. Normalmente, inexacto significa que había un resultado que no se podía representar exactamente y tuvo que ser redondeado. Considerar de nuevo el ejemplo de $(1{,}11 \times 2^{-2}) \times (1{,}11 \times 2^{E_{\text{mín}}}) = 0{,}110001 \times 2^{E_{\text{mín}}}$, con el redondeo al más

cercano. El resultado suministrado es $0{,}11 \times 2^{E_{min}}$, que tuvo que ser redondeado, haciendo que se señale inexacto. Pero ¿es también correcto señalar desbordamiento a cero? El desbordamiento a cero gradual pierde significado porque está limitado el rango del exponente. Si el rango del exponente fuese ilimitado, el resultado suministrado sería $1{,}10 \times 2^{E_{min}-1}$, exactamente la misma respuesta que con el desbordamiento a cero gradual. El hecho de que los números denormalizados tengan menos bits en su mantisa que los números normalizados, no significa ninguna diferencia en este caso. El comentario al estándar [Cody y cols., 1984] fomenta esto como el criterio para inicializar (poner a 1) el señalizador de desbordamiento a cero. Es decir, se debería inicializar siempre que el resultado suministrado sea diferente del que se obtendría en un sistema con el mismo tamaño fraccionario, pero con un rango de exponente muy grande. Sin embargo, debido a la dificultad de implementar este esquema, el estándar permite inicializar el señalizador de desbordamiento a cero siempre que el resultado sea denormal y diferente infinitésimamente del resultado preciso.

Precisión de la multiplicación

En la discusión de la multiplicación entera, mencionamos que los diseñadores deben decidir si suministrar la palabra de orden inferior del producto o el producto completo. Una pregunta similar surge en la multiplicación en punto flotante, donde el producto exacto se puede redondear a la precisión de los operandos o a la precisión del orden superior siguiente. En el caso de la multiplicación entera, ninguno de los lenguajes estándares de alto nivel contiene una construcción que pueda generar una instrucción «simple por simple da doble». La situación es diferente para punto fotante. No sólo muchos lenguajes permiten asignar el producto de dos variables en simple precisión a una de doble precisión, sino que la construcción también la pueden aprovechar los algoritmos numéricos. El caso mejor conocido es utilizar un refinamiento iterativo para resolver sistemas de ecuaciones lineales.

A.6 División y resto

División iterativa

Antes, explicamos un algoritmo para la división entera. Convertirlo en un algoritmo de división de punto flotante es similar a convertir el algoritmo de la multiplicación entera en uno de punto flotante. Si los números que se van a dividir son $s_1 2^{e_1}$ y $s_2 2^{e_2}$ entonces el divisor calculará s_1/s_2, y la respuesta final será este cociente multiplicado por $2^{e_1-e_2}$. Refiriéndonos a la Figura A.2(b), el alineamiento de los operandos es ligeramente diferente de la división entera. Cargar s_2 en b y $s_1/2$ en P para que s_1 se desplace un bit a la derecha. Entonces se puede utilizar el algoritmo para la división entera, y el resultado será de la forma $q_0.q_1...$ Para la división en punto flotante, no es necesario que el registro A contenga operandos. Para redondear, sencillamente, calcular dos bits

adicionales de cociente (guarda y redondeo) y utilizar el resto como el bit retenedor. El dígito de guarda es necesario porque el primer bit del cociente puede ser cero. Sin embargo, como numerador y denominador están ambos normalizados, no es posible que los dos bits más significativos del cociente sean cero.

Hay una aproximación diferente a la división, basada en la iteración. Una máquina real que utiliza este algoritmo se explicará en la Sección A.10. Primero, describiremos los dos principales algoritmos iterativos y después descutiremos los pros y contras de la iteración comparada con los algoritmos directos. Hay una técnica general para construir algoritmos iterativos, denominada *iteración de Newton*, mostrada en la Figura A.9. Primero, plantear el problema en la forma de calcular el cero de una función. Después, comenzando con una estimación para el cero, aproximar la función por su tangente en esa estimación y formar una nueva estimación basándose donde la tangente tiene un cero. Si x_i es una aproximación de un cero, entonces la línea tangente tiene la ecuación

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

Esta ecuación tiene un cero en

$$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad \textbf{A.6.1}$$

Para replantear la división como la determinación del cero de una función, considerar $f(x) = 1/x - b$. Como el cero de esta función está en $1/b$, aplicando la iteración de Newton se obtendrá un método iterativo de calcular $1/b$ a partir de b. Utilizando $f'(x) = -1/x^2$, la Ecuación A.6.1 se convierte en

$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b) \qquad \textbf{A.6.2}$$

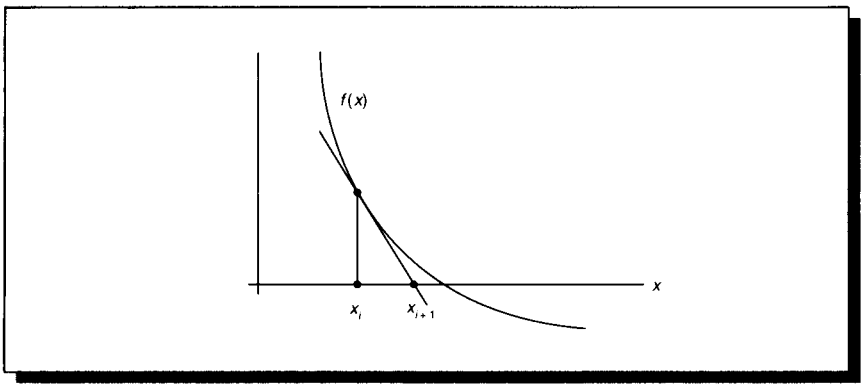


FIGURA A.9 Iteración de Newton para calcular ceros. Si x_i es una estimación de un cero de f, entonces x_{i+1} es una estimación mejor. Para calcular x_{i+1}, calcular la intersección del eje x con la línea tangente a f en x_i.

Por tanto, podemos implementar el cálculo a/b utilizando el siguiente método:

1. Escalar b para que esté en el rango $1 \leqslant b < 2$ y obtener un valor aproximado de $1/b$ (llamado x_0) utilizando una tabla de consulta.
2. Iterar $x_{i+1} = x_i(2 - x_i b)$ hasta conseguir un x_n que sea suficientemente preciso.
3. Calcular ax_n e invertir el escalamiento realizado en el paso 1.

Aquí hay más detalles. ¿Cuántas veces habrá que iterar el paso 2? Decir que x_i tiene una precisión de p bits significa que $(x_i - 1/b)/(1/b) = 2^{-p}$, y una manipulación algebraica muestra $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Por tanto, en cada caso, se duplica el número de bits correctos. La iteración de Newton es **autocorrectora** en el sentido de que no importa realmente cometer un error en x_i. Es decir, x_i se trata como una aproximación en $1/b$ y devuelve x_{i+1} como una mejora de ella (doblando aproximadamente los dígitos). Lo que podría hacer que x_i sea erróneo es el error de redondeo. Sin embargo, es más importante, que en las primeras iteraciones se pueda aprovechar el hecho de que no se esperan muchos bits correctos para realizar la multiplicación en precisión reducida, ganando por tanto velocidad sin sacrificar precisión. Otras aplicaciones de la iteración de Newton se explican en los Ejercicios.

El segundo método iterativo de división, a veces, se denomina *algoritmo de Goldschmidt*. Está basado en la idea de que para calcular a/b, se debe multiplicar el numerador y denominador por un número r siendo $rb \approx 1$. Con más detalle, sea $x_0 = a$ e $y_0 = b$. En cada paso calcular $x_{i+1} = r_i x_i$ e $y_{i+1} = r_i y_i$. Entonces el cociente $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ es constante. Si escogemos r_i para que $y_i \to 1$, entonces $x_i \to a/b$, es decir, x_i converge a la respuesta que queremos. Esta misma idea se puede utilizar para calcular otras funciones. Por ejemplo, para calcular la raíz cuadrada de a, sea $x_0 = a$ e $y_0 = a$, y en cada paso calcular $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Entonces $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, así si r_i se escoge para lograr $x_i \to 1$, entonces $y_i \to \sqrt{a}$. Esta técnica se utiliza para calcular raíces cuadradas en el TI 8847.

Volviendo al algoritmo de división de Goldschmidt, pongamos $x_0 = a$ e $y_0 = b$, y escribamos $b = 1 - \delta$, donde $|\delta| < 1$. Si escogemos $r_0 = 1 + \delta$, entonces $y_1 = r_0 y_0 = 1 - \delta^2$. A continuación escogemos $r_1 = 1 + \delta^2$, para que $y_2 = r_1 \cdot y_1 = 1 - \delta^4$, y así sucesivamente. Como $|\delta| < 1$, $y_i \to 1$. Con esta elección de r_i, x_i se calculará como $x_{i+1} = r_i x_i = (1 + \sigma^{2^i})x_i = (1 + (1 - b)^{2^i})x_i$, o

A.6.3

$$x_{i+1} = a[1 + (1 - b)][1 + (1 - b)^2][1 + (1 - b)^4]...[1 + (1 - b)^{2^i}]$$

Aparecen dos problemas con este algoritmo. Primero, la convergencia es lenta cuando b no es próximo a 1 (es decir, δ no es próximo a 0); y segundo, la fórmula no es autocorrectora —ya que el cociente se está calculando como un producto de términos independientes, un error en uno de ellos no se corregirá. Para tratar con convergencia lenta, si se quiere calcular a/b, buscar un inverso aproximado a b (denominado b') y ejecutar el algoritmo sobre ab'/bb'. Esto convergerá rápidamente ya que $bb' \approx 1$.

Para tratar con el problema de la autocorrección, el cálculo debe ejecutarse con algunos bits de precisión extra para compensar los errores de redondeo. Sin embargo, el algoritmo de Goldschmidt tiene una forma débil de autocorrección, ya que no importa el valor preciso de r_i. Por ello, en las primeras iteraciones (pocas), se puede escoger r_i para que sea una truncación de $1 + \delta^{2^i}$, lo cual puede hacer que estas iteraciones se ejecuten con mucha más rapidez, sin afectar a la velocidad de la convergencia. Si se trunca r_i, entonces y_i no es exactamente igual a $1 - \delta^{2^i}$; así, la Ecuación A.6.3 no se puede utilizar, pero es fácil de organizar el cálculo, para que no dependa del valor preciso de r_i. Con estos cambios, el algoritmo de Goldschmidt es como sigue (las notas entre corchetes muestran la conexión con otras fórmulas anteriores).

1. Escalar a y b para que $1 \leqslant b < 2$.
2. Buscar una aproximación a $1/b$ (denominada b') en un tabla.
3. Poner $x_0 = ab'$ e $y_0 = bb'$.
4. Iterar hasta que x_i se aproxime suficientemente a a/b.

$r \approx 2 - y$ [si $y_i = 1 + \delta_i$, entonces $r \approx 1 - \delta_i$]

$y = y \times r$ [$y_{i+1} = y_i \times r \approx 1 - \delta_i^2$]

$x = x \times r$ [$x_{i+1} = x_i \times r$]

Los dos métodos de iteración están relacionados. Supongamos en el método de Newton que desenrollamos la iteración y calculamos cada término x_{i+1} directamente en función de b, en lugar de recursivamente en función de x_i. Realizando este cálculo, descubrimos que

$$x_{i+1} = x_0(2 - x_0b)(1 + (x_0b - 1)^2)(1 + (x_0b + 1)^4)...(1 + (x_0b - 1)^{2^i})$$

Esta fórmula es muy similar a la Ecuación A.6.3 cuando $a = 1$. En efecto, si las iteraciones se hiciesen con precisión infinita, los dos métodos darían exactamente la misma secuencia x_i.

La ventaja de la iteración es que no requiere hardware especial de división; en cambio puede utilizar el multiplicador (que, sin embargo, requiere control extra). Además, en cada paso, se genera el doble de dígitos que en el paso anterior —de forma distinta a la división ordinaria, que en cada paso produce un número de dígitos fijo. Hay dos desventajas de la división iterativa. La primera es que el estándar del IEEE requiere que la división se redondee correctamente, pero la iteración solamente suministra un resultado que está próximo a la respuesta redondeada correctamente. En el caso de la iteración de Newton, que calcula $1/b$ en lugar de a/b directamente, hay un problema adicional. Aunque $1/b$ se redondee correctamente, no hay garantía que a/b lo sea. Tomemos como ejemplo 5/7: con dos dígitos de precisión 1/7 es 0,14, y $5 \times 0{,}14$ es 0,70, pero 5/7 es 0,71. La segunda desventaja es que la iteración no da resto. Esto es especialmente problemático si el hardware de división de punto flotante se utiliza para realizar una división entera, ya que una operación de resto está presente en casi todos los lenguajes de alto nivel.

El folklore tradicional ha mantenido que la forma de obtener un resultado de la iteración redondeado correctamente es calcular 1/b para que tenga ligeramente más de $2p$ bits, calcular a/b para que tenga ligeramente más de $2p$ bits, y después redondear a p bits. Sin embargo, hay una forma más rápida, que, aparentemente, se implementó por primera vez en el TI 8847. En este método, a/b se calcula con aproximadamente seis bits extra de precisión, dando un cociente preliminar q. Comparando qb con a (de nuevo con sólo seis bits extra), es posible decidir rápidamente si q está redondeado correctamente o si es necesario aumentar o disminuir en 1 la posición menos significativa. Este algoritmo se analiza en los Ejercicios.

Otro factor, a tener en cuenta, cuando se decide sobre los algoritmos de la división es la velocidad relativa de la división y de la multiplicación. Como la división es más compleja que la multiplicación, se realizará más lentamente. Como regla empírica general, los algoritmos de división deberían intentar conseguir una velocidad que sea aproximadamente un tercio la de la multiplicación. Un argumento en favor de esta regla es que hay programas reales (como algunas versiones de Spice) donde la relación de la división a la multiplicación es 1:3. Otro lugar, donde se presenta un factor de tres, es en el método iterativo estándar para calcular la raíz cuadrada. Este método involucra una división por iteración, pero se puede sustituir por otro que utilice tres multiplicaciones. Esto se explica en los ejercicios.

Resto en punto flotante

Para enteros no negativos, la división entera y resto satisfacen

$$a = (a \text{ DIV } b)b + a \text{ REM } b, 0 \leqslant a \text{ REM } b < b$$

Un resto en punto flotante x REM y se puede definir análogamente como $x = \text{INT}(x/y)y + x \text{ REM } y$. ¿Cómo deberá convertirse en entero x/y? La función resto del IEEE utiliza la regla de redondeo al par. Esto es, toma $n = \text{INT}(x/y)$ para que $|x/y - n| \leqslant 1/2$. Si dos n diferentes satisfacen esta relación, tomar el par. Entonces REM se define como $x - yn$. De forma distinta a los enteros donde $0 \leqslant a \text{ REM } b < b$, para números en punto flotante $|x \text{ REM } y| \leqslant y/2$. Aunque esto define precisamente a REM, no es una definición operativa práctica, porque n puede ser enorme. En simple precisión, n puede ser tan grande como $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

Hay una forma natural de calcular REM si se utiliza un algoritmo de división directa. Proceder como si se estuviese calculando x/y. Si $x = s_1 2^{e_1}$ e $y = s_2 2^{e_2}$ y el divisor es como el de la Figura A.2(b), entonces cargar s_1 en P y s_2 en B. Después de $e_1 - e_2$ pasos de división, el registro P contendrá un número r de la forma $x - yn$ que satisface $0 \leqslant r < y$. El resto del IEEE es entonces r o $r - y$. Sólo es necesario recordar el último bit del cociente producido, que es necesario con el fin de resolver casos a mitad de camino. Desgraciadamente, $e_1 - e_2$ puede constar de muchos pasos, y las unidades de punto flotante normalmente tienen una cantidad máxima de tiempo para emplear en una instrucción. Por tanto, habitualmente no es posible implementar REM directamente. Ninguno de los chips explicados en la Sección A.10

implementan REM, pero podrían proporcionar una instrucción de paso-de-resto *(step-remainder)* —esto es lo que se hace en la familia Intel 8087—. Un paso de resto toma como argumentos dos números x e y, y realiza pasos de división hasta que el resto está en P, o han realizado n pasos, donde n es un número pequeño, tal como el número de pasos requeridos para la división en la más alta precisión soportada. El gestor de REM llama $\lfloor (e_1 - e_2)/n \rfloor$ veces, a la instrucción de paso REM utilizando inicialmente x como numerador, pero sustituyéndolo por el resto del paso previo REM en sucesivos pasos. Es útil que la instrucción de paso REM *(step-REM)* devuelva los tres bits de orden inferior del cociente, ya que cuando se realizan reducciones de argumentos trigonométricos en el intervalo $(0, \pi/4)$, se necesita conocer el valor de n mod 8 con el fin de saber en qué cuadrante se está.

Actualmente, la mayoría de los chips más rápidos de punto flotante no implementan el resto, aún cuando sea una parte requerida del estándar del IEEE. Como el estándar permite implementaciones que sean una combinación hardware y software, la operación REM se puede implementar completamente en software. Sin embargo, la disponibilidad de la instrucción de paso-REM haría el cálculo de REM mucho más simple. ¿Vale la pena traer una instrucción de paso-REM? Por dos razones, esta situación es difícil de decidir en base a la frecuencia de datos. Primero, como REM es peculiar al estándar del IEEE, poca gente lo está utilizando actualmente. Examinar la demanda para REM es parecido a estimar la demanda de un nuevo producto. Segundo, el beneficio principal de REM no es un incremento del rendimiento, sino un incremento en la precisión, y no es fácil cuantificar el valor de la precisión. Lo que haremos aquí será presentar, sencillamente, la aplicación principal de REM, que es la reducción de argumentos para funciones periódicas, como seno y coseno.

Hay algunas cuestiones sutiles involucradas en la reducción de argumentos. Para simplificar las cosas, imaginar que se está trabajando en base 10 con 5 cifras significativas, y considerar el cálculo de sen x. Supongamos que $x = 7$. Entonces reducimos por $\pi = 3{,}1416$ y calculamos $\text{sen}(7) = \text{sen}(7 - 2 \times 3{,}1416) = \text{sen}(0{,}7168)$. Pero supongamos que queremos calcular $\text{sen}(2{,}0 \times 10^5)$. Entonces $2 \times 10^5/3{,}1416 = 63661{,}8$ que, en nuestro sistema de 5 posiciones, acaba siendo 63662. Como multiplicar 3,1416 por 63662 da 200000,5392, que se redondea a $2{,}0000 \times 10^5$, la reducción de argumentos reduce 2×10^5 a 0, que está lejos de ser correcto. El problema es que nuestro sistema de 5 posiciones no tiene precisión para realizar la reducción correcta del argumento. Suponer que tenemos el operador REM. Entonces podremos calcular 2×10^5 REM 3,1416 y obtener −0,5392. Sin embargo, esto no es correcto todavía, porque utilizamos 3,1416, que es una aproximación de π. El valor de 2×10^5 REM es −0,071513. La dificultad es que restamos dos números próximos, 2×10^5 y $63662 \times 3{,}1416$, donde $63662 \times 3{,}1416$ era ligeramente erróneo debido a la aproximación de π. Aun cuando REM tenga el efecto de realizar fácilmente la sustracción, todas las cifras significativas de $63662 \times 3{,}1416$ se cancelaron, dejando detrás sólo el error de redondeo.

Tradicionalmente ha habido dos aproximaciones para calcular funciones periódicas con grandes argumentos. La primera es devolver un error a su valor cuando x es grande. La segunda es almacenar π con un gran número de posiciones y hacer una reducción exacta del argumento. El operador REM no

ayuda mucho en ninguna de estas situaciones. Hay una tercera aproximación que se ha utilizado en algunas bibliotecas matemáticas, como la versión UNIX 4.3bsd de Berkeley. En estas bibliotecas, π se calcula como el número de punto flotante más próximo. Llamémosle π de la máquina, y lo denotamos por π'. Entonces, cuando se calcule sen x, reducir x utilizando x REM π'. Como vimos en el ejemplo anterior, x REM π' es bastante diferente de x REM π; así que, calcular sen x como sen (x REM π'), no dará el valor exacto de sen x. Sin embargo, calcular funciones trigonométricas de esta forma tiene la propiedad de que todas las identidades familiares (como $\text{sen}^2x + \cos^2x = 1$), son ciertas con pocos errores de redondeo. Por tanto, utilizar REM junto con el π de la máquina proporciona un método sencillo para calcular funciones trigonométricas, que es preciso para pequeños argumentos y todavía útil para grandes argumentos en muchas aplicaciones.

A.7 Precisiones y tratamiento de excepciones

Precisiones

Las implementaciones del estándar del IEEE solamente exigen soportar simple precisión. Por tanto, el diseñador de computadores debe elegir qué otras precisiones va a soportar. Debido al amplio uso de la doble precisión en el cálculo científico, ésta se implementa casi siempre.

La doble precisión extendida es más problemática. Aunque los procesadores Motorola 68882 e Intel 387 implementan precisión extendida, la mayoría de los chips de punto flotante y alto rendimiento, más recientemente diseñadas, no implementan la precisión extendida. Entre otras razones están: que los 80 bits de la precisión extendida son problemáticos para los buses y registros de 64 bits, y que muchos lenguajes de alto nivel no dan al usuario accesos a precisión extendida. Sin embargo, la precisión extendida es muy útil para los escritores de software matemático. A título de ejemplo, considerar escribir una rutina de biblioteca para calcular la longitud de un vector en el plano $\sqrt{x^2 + y^2}$. Si x es mayor que $2^{E_{máx}/2}$, entonces realizar este cálculo de la forma obvia producirá desbordamiento. Esto significa que o bien el rango permisible del exponente para esta subrutina se reducirá a la mitad, o se tendrá que emplear un algoritmo más complejo que utilice escalamiento. Pero si se dispone de precisión extendida, entonces funcionará el algoritmo sencillo. Calcular la longitud de un vector es una tarea simple, y no es difícil realizar un algoritmo que no produzca desbordamiento. Sin embargo, hay problemas más complejos para los cuales la precisión extendida significa la diferencia entre un algoritmo rápido y sencillo y otro mucho más complejo. Uno de los mejores ejemplos es la conversión binario/decimal. Un algoritmo eficiente para la conversión binario/decimal que haga uso esencial de la precisión extendida se presenta de forma muy legible en Coonen [1984]. Este algoritmo también está brevemente explicado en Goldberg [1989]. Calcular valores precisos para funciones transcendentes es otro ejemplo de problema que se hace mucho más fácil si se dispone de precisión extendida.

Un hecho muy importante sobre precisión concierne al *doble redondeo.* Para ilustrarlo en decimal, suponer que queremos calcular 1,9 × 0,66, y que la simple precisión es de dos dígitos, mientras que la precisión extendida es de tres dígitos. El resultado exacto del producto es 1,254. Redondeado a la precisión extendida, el resultado es 1,25. Cuando, a continuación, se redondea a simple precisión, obtenemos 1,2. Sin embargo, el resultado de 1,9 × 0,66 redondeado correctamente en simple precisión es 1,3. Por tanto, redondear dos veces puede no producir el mismo resultado que redondear una vez. Suponer que se quiere construir hardware que sólo haga aritmética en doble precisión. ¿Se puede simular simple precisión calculando primero en doble precisión y después redondeando a simple? El ejemplo anterior sugiere que no se puede. Sin embargo, el doble redondeo no es siempre peligroso. En efecto, la siguiente regla es cierta (aunque no sea fácil de probar).

> *Si x e y tienen mantisas de p bits, y x + y se calcula exactamente y después se redondea a q posiciones, un segundo redondeo a p posiciones no cambiará la respuesta si* $p \leqslant (q - 1)/2$. *Esto es cierto no sólo para la suma, sino también para la multiplicación, división y raíz cuadrada.*

En nuestro ejemplo anterior, $q = 3$, y $p = 2$, así $2 \leqslant (3 - 1)/2$ no es cierto. Por otro lado, para la aritmética del IEEE, la doble precisión tiene $p = 53$, y la simple precisión tiene $p = 24 \leqslant (q - 1)/2 = 26$. Por tanto, la simple precisión se puede implementar calculando en doble precisión (es decir, calcular exactamente la solución y después redondear a doble) y después redondear a simple precisión.

El estándar requiere implementaciones que proporcionen versiones de la suma, resta, multiplicación, división y resto que tengan dos operandos de la misma precisión y produzcan un resultado de esa precisión. También recomienda que las implementaciones permitan operaciones que tengan operandos de dos precisiones diferentes y devuelvan un resultado cuya precisión sea como mínimo la del operando que la tenga mayor. El estándar permite implementaciones que combinen dos operandos y devuelvan un resultado en una precisión mayor. Recordar que el resultado de una operación es el resultado exacto redondeado a la precisión de destino. Lo que el estándar no permite es combinar dos operandos y devolver un resultado en una precisión inferior. Aunque, en principio, esto pueda parecer una restricción menor, considerar de nuevo el problema de calcular $\sqrt{x^2 + y^2}$. Si x e y están en doble precisión, entonces se puede calcular $x^2 + y^2$ en precisión extendida y después calcular una raíz cuadrada que tome un argumento en precisión extendida y devuelva una respuesta en doble precisión. Pero esto no está permitido por el estándar.

Hay otra cuestión relacionada. El estándar permite combinar dos variables extendidas para producir un resultado que se almacene en formato extendido, pero redondeado a doble precisión. Sin embargo, esto no ayuda en el ejemplo de la raíz cuadrada, ya que el resultado de la raíz cuadrada se debe convertir todavía explícitamente desde un formato extendido a un formato de doble precisión.

Excepciones

El estándar del IEEE define cinco excepciones: desbordamiento a cero, desbordamiento, división por cero, inexactitud e invalidez. Implícitamente, cuando se presentan estas excepciones, meramente inicializan un señalizador y continúa el cálculo. Los señalizadores son *de retención* (*sticky*), significando que una vez puesto a 1, permanece a 1 hasta que se borren explícitamente. El estándar recomienda implementaciones que proporcionen un bit de habilitación de trap para cada excepción. Cuando se presenta una excepción con un manipulador de trap habilitado se llama un manipulador de trap de usuario, y el valor del señalizador de excepción asociado es indefinido.

Las excepciones de desbordamiento a cero, desbordamiento y división por cero se encuentran en muchos otros sistemas. La *excepción de inexactitud* es peculiar a la aritmética del IEEE y se presenta o cuando el resultado de una operación se debe redondear o cuando hay desbordamiento. En efecto, como 1/0 y una operación que tenga desbordamiento dan ∞, se deben consultar los señalizadores de excepción para distinguirlas. La excepción de inexactitud es una «excepción» inusual, ya que no es realmente una condición excepcional porque se presenta frecuentemente. Por tanto, habilitar un manipulador de interrupciones para «inexactitudes» probablemente tendrá un severo impacto en el rendimiento. La *excepción de invalidez* es para cosas como $\sqrt{-1}$, 0/0 o $\infty - \infty$, que no tienen ningún valor natural como números en punto flotante o como $\pm\infty$. Por tanto, 1/0 provoca una excepción de divide por cero y proporciona ∞, mientras que 0/0 provoca una excepción de invalidez y proporciona un NaN. Hay una peculiaridad en el desbordamiento a cero del IEEE, porque no se señala siempre que los números caen debajo de $1{,}0 \times 2^{E_{\min}}$. Si no está instalado un manipulador de traps de usuario, entonces el desbordamiento a cero se señala sólo si el resultado de una operación está por debajo de $2^{E_{\min}}$ y es inexacto.

El estándar del IEEE supone que cuando se presenta un trap, es posible identificar la operación y sus operandos. En máquinas con segmentación, o máquinas con múltiples unidades aritméticas, cuando se presenta una excepción puede no ser suficiente hacer que el manipulador de traps examine el contador de programa. Soporte hardware puede ser necesario con el fin de identificar exactamente qué operación causó el trap. Otro problema se ilustra con el siguiente fragmento de programa.

```
X = Y * Z;
Z = A + B;
```

Estas dos instrucciones se pueden ejecutar en paralelo. Si la multiplicación genera un trap, su argumento `z` puede haber sido sobreescrito por la suma, ya que la suma es habitualmente más rápida que la multiplicación. Los sistemas de computadores que soportan traps en el estándar del IEEE deben suministrar alguna forma de guardar el valor de `z`, bien mediante hardware o haciendo que el compilador evite dicha situación.

Una aproximación a este problema, utilizada en la MIPS R3010, es tratar las excepciones de punto flotante de forma análoga a las excepciones de fallo de página. Si una instrucción que asigna una posición de memoria a un regis-

tro provoca un fallo de página, la ejecución de la instrucción debe detenerse antes de que malfuncione el registro porque (por ejemplo) ese mismo registro se puede utilizar para referenciar la memoria que causó el fallo. La clave para hacer este trabajo es calcular la dirección de memoria al principio del ciclo de instrucción, antes de que la instrucción realmente escriba cualquier cosa. Un truco similar puede hacerse con las operaciones en punto flotante. Una instrucción que pueda provocar una excepción se puede identificar al principio del ciclo de instrucción. Por ejemplo, una suma puede producir desbordamiento sólo si uno de los operandos tiene un exponente $E_{máx}$, y así sucesivamente. Esta primera comprobación es conservadora: puede señalizar una operación que realmente no provoque una excepción. Sin embargo, si estos falsos avisos son raros, entonces esta técnica tendrá un rendimiento excelente. Cuando una instrucción se señala como posiblemente excepcional, un código especial en un manipulador de traps puede calcularla sin destruir ningún estado. Recordar que todos estos problemas se presentan sólo cuando están habilitados los manipuladores de traps. En cualquier otro caso, inicializar los señalizadores de excepción durante el tratamiento normal es sencillo.

Hay una sutileza que debe ser mencionada y que involucra al trap de desbordamiento a cero. Cuando no hay manipulador de traps de desbordamiento a cero, el resultado de una operación que involucre un desbordamiento a cero es un número denormal. Cuando hay un manipulador de traps, se proporciona el resultado de la operación con el exponente aproximado *(wrapped around)*. Ahora hay un potencial problema de doble redondeo. Si el modo de redondeo es redondear hacia el más próximo, cuando hay un manipulador de traps el resultado se redondea correctamente a p bits significativos. Si no hay manipulador de traps, el resultado se redondea a menos de p bits, dependiendo del número de ceros que encabecen el número denormal. Si el manipulador de traps quiere devolver el resultado denormal, no puede redondear exactamente su argumento, porque eso puede conducir a un error de doble redondeo. Por tanto, el manipulador de traps debe recibir como mínimo un bit extra de información si se quiere que suministre correctamente el resultado redondeado.

A.8 Aceleración de la suma entera

La sección anterior mostraba que hay muchos pasos que realizar en la implementación de operaciones en punto flotante. Sin embargo, cada operación de punto flotante, eventualmente, se reduce a una operación entera. Por tanto, incrementar la velocidad de las operaciones enteras también conducirá a un punto flotante más rápido.

La suma entera es la operación más sencilla y la más importante. Incluso para los programas que no hacen aritmética explícita, la suma se debe realizar para incrementar el contador de programa y para realizar cálculo de direcciones. A pesar de la simplicidad de la suma, no hay una única forma que sea la mejor para realizar sumas de alta velocidad. Explicaremos tres técnicas que son de uso actual: anticipación de acarreo, salto de acarreo y selección de acarreo.

Anticipación de acarreo

Un sumador de n bits es, exactamente, un circuito combinacional. Por tanto se puede escribir por una fórmula lógica cuya forma sea una suma de productos y se pueda calcular por un circuito con dos niveles de lógica. ¿Cómo se determina el circuito que haga esto? Recordar de la Ecuación A.2.1 que la fórmula para el bit i-ésimo de la suma es

A.8.1 $$s_i = a_i\bar{b}_i\bar{c}_i + \bar{a}_i b_i\bar{c}_i + \bar{a}_i\bar{b}_i c_i + a_i b_i c_i$$

El problema de esta fórmula es que aunque conozcamos los valores de a_i y b_i —son entradas al circuito— no conocemos c_i. Así, nuestro objetivo es escribir c_i en función de a_i y b_i. Para hacer esto, primero reescribimos la Ecuación A.2.2 como

A.8.2 $$c_{i+1} = g_i + p_i c_i, \quad g_i = a_i b_i, \quad p_i = a_i + b_i$$

Aquí está la razón de los símbolos p y g: si g_i es cierto, entonces c_{i+1} es verdadero, por tanto se *genera* un acarreo. Por tanto, g significa generar. Si p_i es cierto, entonces si c_i es cierto, se *propaga* a c_{i+1}. Empezar con la Ecuación A.8.1 y utilizar la Ecuación A.8.2 para sustituir c_i por $g_{i-1} + p_{i-1}c_{i-1}$. Entonces, utilizar la Ecuación A.8.2 con $i - 1$ en lugar de i, para sutituir c_{i-1} por c_{i-2}, y así sucesivamente. Esto da el resultado

A.8.3 $$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_1 g_0 + p_i p_{i-1} \ldots p_1 p_0 c_0$$

Un sumador que calcule acarreos utilizando la Ecuación A.8.3 se denomina *sumador con anticipación de acarreo*, o sumador CLA *(Carry Lookhead Adder)*. Un sumador CLA requiere un nivel de lógica para formar p y g, dos niveles para formar los acarreos, y dos para la suma, dando un total de cinco niveles lógicos. Esto es una mejora enorme sobre los $2n$ niveles requeridos por el sumador con transmisión de acarreo.

Desgraciadamente, como es evidente de la Ecuación A.8.3 o de la Figura A.10, un sumador con anticipación de acarreo de n bits requiere un «fan-in» (abanico de entrada) de $n + 1$ en la puerta OR, así como en la puerta AND de más a la derecha. Además, la señal p_{n-1} debe atacar n puertas AND. Además, la estructura irregular y muchas conexiones largas de la Figura A.10 hacen que no sea práctico construir un sumador completo con anticipación de acarreo cuando n es grande.

Sin embargo, podemos utilizar la idea de anticipación de acarreo para construir un sumador que tenga aproximadamente $\log_2 n$ niveles lógicos (sustancialmente menor que $2n$ requeridos por un sumador con transmisión de acarreo), y todavía tiene una estructura regular y simple. La idea es construir las p y las g en pasos. Hemos visto ya que

$$c_1 = g_0 + c_0 p_0$$

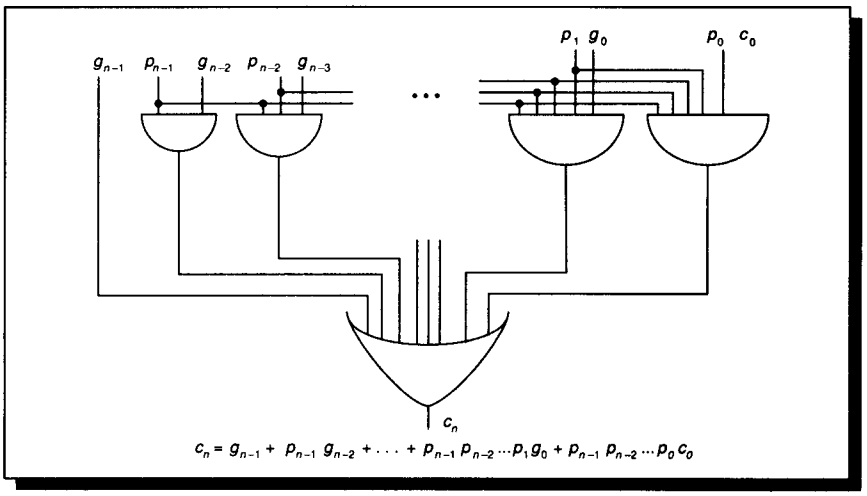


FIGURA A.10 Circuito anticipador de acarreo para el cálculo del acarreo de salida c_n de un sumador de n bits.

Esto indica que hay un acarreo que sale de la posición 0 (c_1) si se genera un acarreo en la posición 0, o si hay un acarreo en la posición 0 y se propaga. Análogamente,

$$c_2 = G_{01} + P_{01}c_0$$

G_{01} significa que se genera un acarreo fuera del bloque que consta de los dos primeros bits. P_{01} significa que se propaga un acarreo a través de este bloque. P y G tienen las siguientes ecuaciones lógicas:

$$G_{01} = g_1 + p_1 g_0$$
$$P_{01} = p_1 p_0$$

De forma más general, para cualquier j con $i < j, j + 1 < k$, tenemos las relaciones recursivas

A.8.4
$$c_{k+1} = G_{ik} + P_{ik}c_i$$

A.8.5
$$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$$

A.8.6
$$P_{ik} = P_{ij}P_{j+1,k}$$

La Ecuación A.8.5 indica que se genera un acarreo fuera del bloque, que consta de los bits i a k inclusive si se genera en la parte de orden superior del bloque $(j + 1,k)$ o si se genera en la parte de orden inferior (i,j) del bloque y después se propaga a través de la parte superior. Estas ecuaciones también se cumplen para $i \leqslant j < k$ si ponemos $G_{ii} = g_i$ y $P_{ii} = p_i$.

Ejemplo

Expresar P_{03} y G_{03} en función de p y g.

Respuesta

Usando A.8.6, $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$. Como $P_{ii} = p_i$, $P_{03} = p_0p_1p_2p_3$. Para G_{03}, la Ecuación A.8.5 indica $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$.

Con estos preliminares, ahora podemos mostrar el diseño de un sumador CLA práctico. El sumador consta de dos partes. La primera parte calcula varios valores de P y G a partir de p_i y g_i, utilizando las Ecuaciones A.8.5 y A.8.6; la segunda parte utiliza estos valores de P y G para calcular todos los acarreos vía la Ecuación A.8.4. La primera parte del diseño está en la Figura A.11. En la parte superior del diagrama, los números de entrada $a_7...a_0$ y $b_7...b_0$ se convierten a p y g utilizando celdas del tipo 1. Entonces se generan diversos P y G combinando celdas del tipo 2 en una estructura de árbol binario. La segunda parte del diseño se muestra en la Figura A.12. Conectando c_0 en la parte inferior de este árbol, todos los bits de acarreo salen por la parte superior. Cada celda debe conocer un par de valores (P,G) con el fin de realizar la conversión, y el valor que necesita está escrito dentro de las celdas. Ahora comparar la Figura A.11 y la Figura A.12. Hay una correspondencia uno a uno entre las celdas, y el valor de (P,G) que necesitan las celdas que generan acarreo es exactamente el valor conocido por las celdas que generan los correspondientes

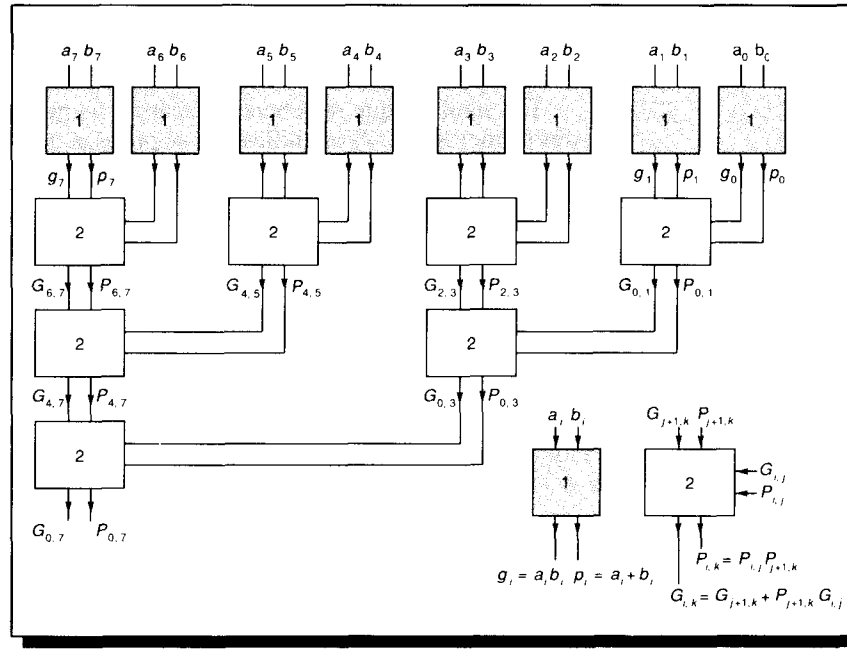


FIGURA A.11 Primera parte del árbol del anticipación de acarreo. Tal como las señales fluyen de la parte superior a la inferior, se calculan varios valores de P y G.

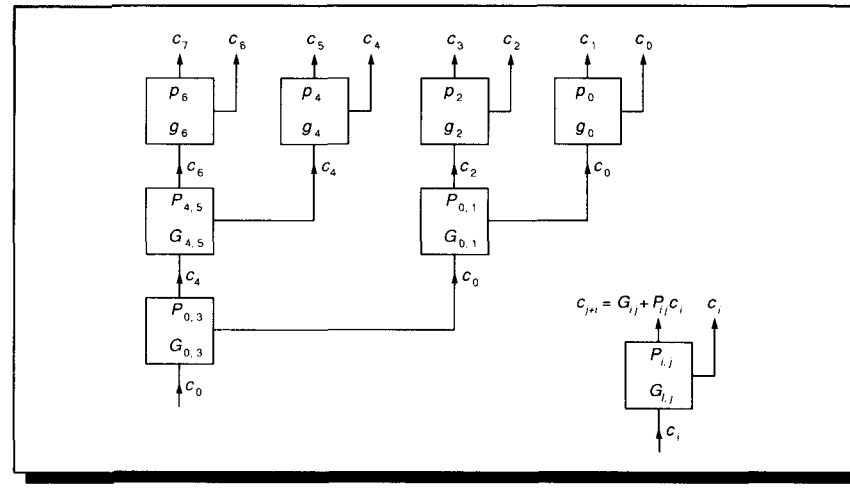


FIGURA A.12 Segunda parte del árbol de anticipación de acarreo. Las señales fluyen de la parte inferior a la superior, combinando P y G para formar los acarreos.

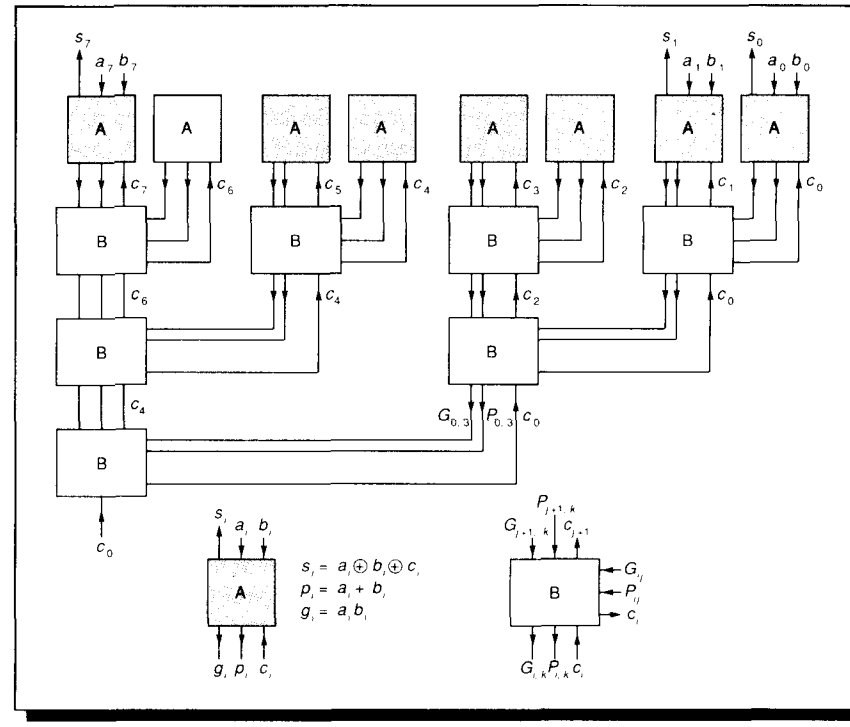


FIGURA A.13 Arbol completo del sumador con antipación de acarreo. Esta es la combinación de las Figuras A.11 y A.12 Los números que se van a sumar entran por la parte superior, fluyen al extremo inferior para combinarse con c_0, y después vuelven a fluir hacia arriba para calcular los bits de la suma.

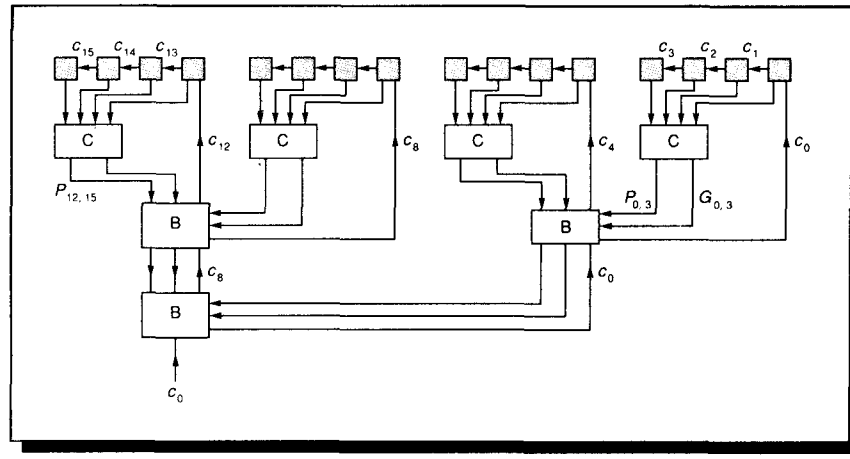


FIGURA A.14 Combinación del sumador CLA y el sumador con propagación de acarreo. En la fila superior, los acarreos se propagan dentro de cada grupo de cuatro cajas.

(*P,G*). La celda combinada se muestra en la Figura A.13. Los números que se van a sumar fluyen en la parte superior hacia la inferior a través del árbol, combinándose con c_0 en la parte inferior, y fluyendo hacia arriba para formar los acarreos. Observar que falta una cosa en la Figura A.13: una pequeña parte de lógica extra para calcular c_8, el acarreo de salida del sumador.

Los bits en un CLA deben pasar a través de $\log_2 n$ niveles lógicos, en comparación con $2n$ para un sumador con propagación de acarreo. Esto es una mejora sustancial, especialmente para un n grande. Sin embargo, mientras que el sumador con transmisión de acarreo tiene n celdas, el sumador CLA tiene $2n$ celdas, aunque en nuestra organización ocupa un espacio $n \log n$. La cuestión es que una pequeña inversión en tamaño amortiza una enorme mejora en velocidad.

Hay una serie de modificaciones dependientes de la tecnología que pueden mejorar los sumadores CLA. Por ejemplo, si cada nodo del árbol tiene tres entradas en lugar de dos, entonces la altura del árbol decrecerá de $\log_2 n$ a $\log_3 n$. Por supuesto, las celdas serán más complejas y, por tanto operarán más lentamente, anulando la ventaja de la disminución de altura. Para tecnologías donde las propagaciones funcionan bien, puede ser mejor un diseño híbrido. Esto se ilustra en la Figura A.14. Los acarreos se propagan entre los sumadores en el nivel superior, mientras que las cajas «B» son las mismas que las de la Figura A.13. Este diseño será más rápido si el tiempo de propagación entre cuatro sumadores es más rápido que el tiempo que se tarda en atravesar un nivel de cajas «B».

Sumadores con salto de acarreo

Un *sumador con salto de acarreo* (*carry-skip adder*) se sitúa a mitad de camino entre un sumador con transmisión de acarreo y un sumador con anti-

cipación de acarreo, tanto en velocidad como coste. (Un sumador con salto de acarreo no se denomina CSA, ya que el nombre está reservado para sumadores que no propagan el acarreo.) La motivación para este sumador proviene del examen de las ecuaciones de P y G. Por ejemplo,

$$P_{03} = p_0 p_1 p_2 p_3$$
$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Calcular P es mucho más simple que calcular G, y un sumador con salto de acarreo sólo calcula los P. Tal sumador se ilustra en la Figura A.15. Los acarreos comienzan a transmitirse simultáneamente en cada bloque. Si cualquier bloque genera un acarreo, entonces el acarreo que sale de un bloque es cierto, aun cuando el acarreo que entre en ese bloque pueda no ser correcto todavía. Si al comienzo de cada operación de suma, el acarreo de entrada de cada bloque es cero, entonces no se generará ningún acarreo espurio de salida. Por tanto, el acarreo de salida de cada bloque se puede considerar, entonces, como si fuese la señal G. Una vez que se genere el acarreo de salida del bloque menos significativo no sólo alimenta al bloque siguiente, sino que también se conecta a la puerta AND con la señal P de ese bloque siguiente. Si las señales de acarreo de salida y P son ambas ciertas, entonces el acarreo **salta** el segundo bloque y está listo para atacar al tercer bloque, y así sucesivamente. El sumador con salto de acarreo es sólo práctico si las señales de acarreo de entrada se pueden borrar fácilmente al comienzo de cada operación —por ejemplo por precarga en CMOS.

Para analizar la velocidad de un sumador con salto de acarreo, supongamos que una señal que pasa a través de dos niveles lógicos necesita una unidad de tiempo. Entonces empleará k unidades de tiempo para que un acarreo se propague a través de un bloque de tamaño k, y tardará una unidad de tiempo para que un acarreo salte un bloque. El camino más largo de señal en el sumador con salto de acarreo comienza cuando se genera un acarreo en la posición 0. Entonces emplea k unidades de tiempo para propagarse a través del primer bloque, $n/k - 2$ unidades de tiempo para saltar los bloques, y k más para propagarse a través del último bloque. Para ser específico: si tenemos un sumador de 20 bits descompuesto en grupos de 4 bits, empleará 11 unidades de tiempo en realizar una suma. Suponer que tenemos el bloque menos significativo con 4 bits, pero combinamos los dos bloques siguientes en un solo bloque de 8 bits. Entonces el tiempo del sumador cae a 10 unidades de tiempo.

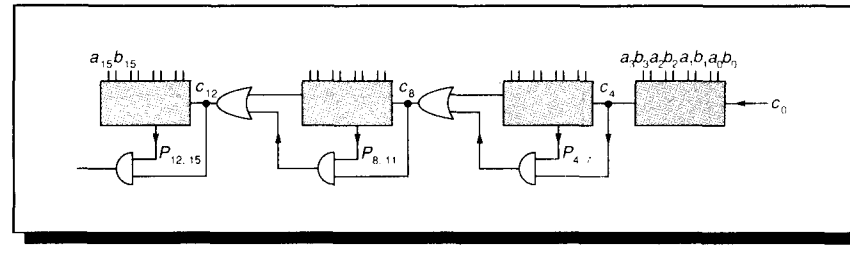


FIGURA A.15 Sumador con salto de acarreo.

Sin embargo, si hubiésemos combinado tres bloques en lugar de dos, entonces el tiempo de propagación a través de esta unidad de 3 bloques (12 bits en total) dominaría el tiempo de suma. Sin embargo, el principio general es importante: para un sumador con salto de acarreo, hacer los bloques interiores mayores acelerará el sumador. En efecto, la misma idea de variar el tamaño de bloques, a veces, puede acelerar también otros diseños de sumadores. Debido a la gran cantidad de propagación de señales, un sumador con salto de acarreo es más apropiado para tecnologías donde la propagación de señales sea rápida.

Sumador con selección de acarreo

Un *sumador con selección de acarreo* (*carry select-adder*) funciona según el siguiente principio: se realizan dos sumas en paralelo, una suponiendo que el acarreo es cero y la otra suponiendo que es uno. Cuando, finalmente se conoce el acarreo, la suma correcta (que ha sido precalculada) se selecciona simplemente. Un ejemplo de dicho diseño se muestra en la Figura A.16. Un sumador de 8 bits se divide en dos partes, y el acarreo de salida de la parte de menor peso se utiliza para seleccionar la mitad más significativa. Si cada bloque está calculando su suma utilizando propagación (un algoritmo de tiempo lineal), entonces el diseño de la Figura A.16 es el doble de rápido al 50 por 100 más de coste. Sin embargo, observar que la señal c_4 debe controlar muchos multiplexores, que pueden ser muy lentos en algunas tecnologías. En lugar de dividir el sumador en mitades, se podría dividir en cuartos para lograr mayor velocidad. Esto se ilustra en la Figura A.17. Si se tardan k unidades de

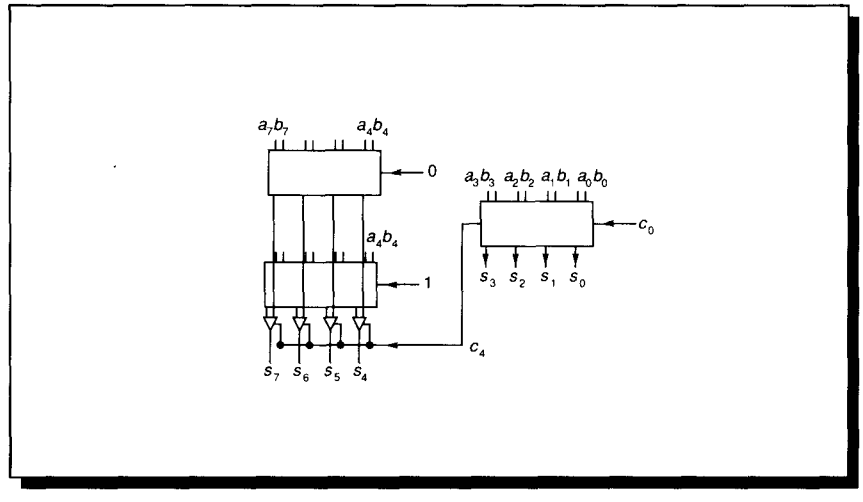


FIGURA A.16 Sumador con selección de acarreo simple. Al mismo tiempo que se está calculando la suma de los cuatro bits de orden inferior, los bits de orden superior se calculan dos veces en paralelo: una vez suponiendo que $c_4 = 0$, y otra suponiendo que $c_4 = 1$.

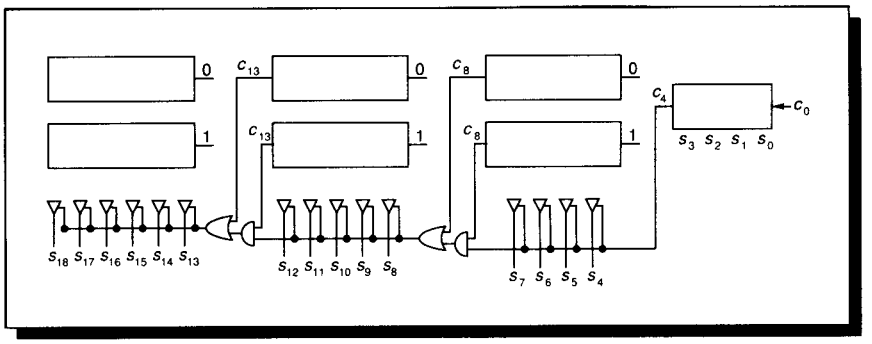


FIGURA A.17 Sumador con selección de acarreo. Tan pronto como se conoce el acarreo de salida del bloque de más a la derecha, se utiliza para seleccionar los otros bits de la suma.

tiempo en un bloque, para sumar números de k bits, y se emplea una unidad de tiempo para calcular la entrada del multiplexor desde dos señales de acarreo de salida, entonces para la operación óptima, cada bloque debería tener un bit más que el siguiente, como muestra la Figura A.17. Por lo tanto, como en el sumador con salto de acarreo, el mejor diseño involucra bloques de tamaño variable.

Como resumen de esta sección, los requerimientos asintóticos de espacio y tiempo para los diferentes sumadores se dan en la Figura A.18. Estos sumadores no deben considerarse como elecciones disjuntas, sino como bloques de construcción que se pueden utilizar para construir un sumador. La utilidad de estos diferentes bloques de construcción diferentes depende enormemente de la tecnología utilizada. Por ejemplo, el sumador con selección de acarreo funciona bien cuando una señal puede controlar muchos multiplexores, y el sumador con salto de acarreo es atractivo en tecnologías donde las señales se pueden borrar al comienzo de cada operación. Conocer el comportamiento asintótico de los sumadores es útil para comprenderlos, pero confiar mucho en ese comportamiento es una pifia. La razón es que el comportamiento asintótico sólo es importante cuando n es muy grande. Pero n para un sumador son los bits de precisión, y la doble precisión es, hoy día, igual que hace veinte años —con aproximadamente 53 bits—. Aunque sea cierto que, así como los

	Tiempo	Espacio
Propagación	$O(n)$	$O(n)$
CLA	$O(\log n)$	$O(n \log n)$
Salto de acarreo	$O(\sqrt{n})$	$O(n)$
Selección de acarreo	$O(\sqrt{n})$	$O(n)$

FIGURA A.18 Requerimientos asintóticos de tiempo y espacio para cuatro tipos diferentes de sumadores.

computadores son más rápidos, los cálculos son más largos —y, por tanto, tienen más error de redondeo, lo que a su vez requiere más precisión— este efecto crece muy lentamente con el tiempo.

A.9 Aceleración de la multiplicación y división enteras

Los algoritmos de multiplicación y división presentados en la Sección A.2 son bastante lentos, produciendo un bit por ciclo (aunque ese ciclo pueda ser una fracción del tiempo de ciclo de instrucción de CPU). En esta sección explicaremos diversas técnicas, de más alto rendimiento, para la multiplicación y la división.

Desplazamiento sobre ceros

El desplazamiento sobre ceros *(shifting over zeros)* es una técnica que actualmente no se utiliza mucho, pero es instructivo considerarla. Se distingue por el hecho que el tiempo de ejecución es dependiente del operando. Su falta de uso es atribuible, principalmente, su imposibilidad de ofrecer suficientes mejoras sobre los algoritmos bit-a-bit *(bit at a time algorithms)*. Además, la segmentación, sincronización con la CPU, y buena optimización de los compiladores son difíciles con algoritmos que emplean un tiempo variable. En la multiplicación, la idea que subyace en el desplazamiento sobre ceros es que la suma lógica detecte cuándo es cero el bit de orden inferior del registro A (ver Fig. A.2(a)) y, si es así, saltar el paso de la suma y proceder, directamente, con el paso de desplazamiento —de aquí el término *desplazamiento sobre ceros*—. Esta técnica se hace más útil si se puede incrementar el número de ceros del operando A. Los ejercicios explican cómo la recodificación de Booth incrementa los ceros.

¿Qué ocurre en el desplazamiento para la división? En la división sin restauración, en cada paso se realiza una operación de la ALU (bien una suma o una resta) lo que sugiere que no hay oportunidad de saltar ninguna operación. Pero considerar la división de esta forma: para calcular a/b, restar múltiplos de b de a, y después informar del número de restas que se han hecho. En cada etapa del proceso de resta, el resto debe caber en el registro P de la Figura A.2(b). En el caso que el resto sea un número positivo pequeño, normalmente se resta b; pero suponer en cambio que únicamente se desplaza el resto y b se resta la siguiente vez. Mientras el resto sea suficientemente pequeño (su bit de orden superior 0), después de desplazarlo, todavía cabrá en el registro P, y no se pierde ninguna información. Sin embargo, este método requiere cambiar la forma en que contabilizamos el número de veces que b se ha restado de a. Esta idea, habitualmente, se conoce con el nombre de *división SRT*, por Sweeney, Robertson y Tocher, que propusieron independientemente algoritmos de esta naturaleza. La complicación principal extra de la división SRT es que los bits del cociente no se pueden determinar inmediatamente a partir del signo de P en cada paso, como puede hacerse en la división ordinaria sin restauración.

Más precisamente, para dividir a y b donde a y b son números de n bits, cargar a y b en los registros A y B, respectivamente, de la Figura A.2.

1. Si B tiene k ceros de encabezamiento cuando se expresa utilizando n bits, desplazar todos los registros k bits a la izquierda. Después de este desplazamiento, como b tiene $n + 1$ bits, su bit más significativo será 0, y su segundo bit más significativo será 1.

2. Para $i = 0, n - 1$ hacer

   Si los tres bits superiores de P son iguales, poner $q_i = 0$ y desplazar un bit a la izquierda (P,A).

   Si los tres bits superiores de P no son iguales y P es negativo, poner $q_i = \overline{1}$, desplazar un bit a la izquierda (P,A), y sumar B.

   En cualquier otro caso poner $q_i = 1$, desplazar (P,A) un bit a la izquierda y restar B.

   Fin del bucle

3. Si el resto final es negativo, corregir el resto sumando B, y corregir el cociente restando 1 de q_0. Finalmente, el resto se debe desplazar k bits a la derecha, donde k es el desplazamiento inicial.

Un ejemplo numérico se da en la Figura A.19. Aunque estamos explicando la división entera, el desplazar el punto binario desde la derecha del bit menos significativo a la izquierda del bit más significativo ayuda a explicar el algoritmo. Entonces si $n = 4$ y la operación es 9/4, el resgistro A contiene 0,1001 y (recordando que el registro B tiene $n + 1$ bits), el registro B contiene 0,0100.

P	A	
00000	1000	B contiene 0011, así se desplazan todos los registros dos posiciones a la izquierda
00010	0000	B contiene ahora 1100. Los bits superiores de P son iguales, así se desplazan y se pone $q_0 = 0$
00100	0000	Los bits superiores no son iguales, así $q_1 = 1$
01000	0000	desplaza y
+10100		resta B
11100	0000	Los bits superiores son iguales, así se desplaza y se pone $q_2 = 0$
11000	0000	Los bits superiores son distintos, así se pone $q_3 = -1$
10000	0000	desplaza y
+01100		suma B
11100		El resto es negativo, así restaurarlo y restar 1 de q_0
+01100		
01000		Este se debe desplazar dos posiciones a la derecha para dar el resto
		Resto = 10, $q = 010\overline{1}-1=0010$

FIGURA A.19 División SRT de 1000/0011.

Como esto cambia el punto binario en el numerador y denominador, el cociente no es afectado. Siendo el resto un número en complemento a dos, un registro P de 1.1110_2 representa $-1/8$. Con este convenio el registro P contiene números que satisfacen $-1 \leqslant P < 1$. El primer paso del algoritmo desplaza b ya que $b \geqslant 1/2$. Como antes, sea r el valor del par (P,A). Nuestra regla para que se realice la operación de la ALU es ésta: si $-1/4 \leqslant r < 1/4$ (cierto siempre que los tres bits superiores de P sean iguales), entonces calcular $2r$ desplazando un bit a la izquierda (P,A); si no, si $r < 0$ (y por tanto $r < -1/4$, ya que en otro caso habría sido eliminado por la primera condición), entonces calcular $2r + b$ desplazando y después sumando, si no $r \geqslant 1/4$ y se resta b de $2r$. Utilizando $b \geqslant 1/2$, es fácil de comprobar que estas reglas mantienen $-1/2 \leqslant r < 1/2$. Para la división sin restauración, sólo tenemos $|r| \leqslant b$, y necesitamos que P sea de $n + 1$ bits. Pero para la división SRT, el límite sobre el registro r es mas estricto, $-1/2 \leqslant r < 1/2$. Por tanto, nos podemos ahorrar un bit eliminando el bit de orden superior de P (y b y el sumador). En particular, el test para la igualdad de los tres bits superiores de P se convierte en un test de, exactamente, dos bits.

El algoritmo puede cambiar ligeramente en una implementación de la división SRT. Después de cada operación de la ALU, el registro P puede ser desplazado, tantos lugares como sea necesario, para hacer bien $P \geqslant 1/4$ o $P < -1/4$. Al desplazar k posiciones, k bits del cociente se hacen, a la vez, igual a cero. Por esta razón la división SRT a veces se describe como la que mantiene el resto normalizado a $|r| \geqslant 1/4$.

Observar que el valor del bit del cociente, calculado en un paso dado, está basado en la operación que se realiza en ese paso (que además depende del resultado de la operación del paso anterior). Esto contrasta con la división sin restauración, donde el bit del cociente, calculado en el paso *i*ésimo, depende del resultado de la operación en el mismo paso. Esta diferencia se refleja en el hecho que cuando el resto final es negativo, el último bit del cociente se debe ajustar en la división SRT, pero no en la división sin restauración. Sin embargo, el hecho clave con respecto a los bits del cociente en la división SRT es que pueden incluir $\bar{1}$. Por consiguiente, los bits del cociente no se pueden almacenar en los bits de orden inferior del registro A; además, el cociente se debe convertir a complemento a dos ordinario mediante un sumador completo. Una forma común de hacer esto es acumular los bits positivos del cociente en un registro y los negativos en otro, y después restar los dos registros una vez que todos los bits sean conocidos. Como hay más de una manera de escribir un número en función de los dígitos -1, 0, 1, la división SRT se dice que utiliza una representación de cociente *redundante*.

Las diferencias entre la división SRT y la división ordinaria sin restauración pueden resumirse como sigue:

1. Regla de decisión de la ALU: en la división sin restauración, se determina por el signo de P; en SRT, se determina por los dos bits más significativos de P.

2. Determinación del cociente: en la división sin restauración, es inmediato a partir de los signos de P; en SRT, se debe calcular en un sumador completo de n bits.

3. Velocidad: la división SRT será más rápida sobre operandos que produzcan bits de cociente cero.

Aceleración de la multiplicación con un único sumador

Como mencionamos antes, las técnicas de desplazamiento sobre cero no se utilizan mucho en el hardware actual. Ahora explicamos algunos métodos que tienen un uso más extendido. Los métodos que incrementan la velocidad de la multiplicación pueden dividirse en dos clases: los que utilizan un único sumador y los que utilizan múltiples sumadores. Primero expliquemos las técnicas que utilizan un único sumador.

En la discusión de la suma observamos que, debido a la propagación del acarreo, no es práctico realizar sumas con dos niveles de lógica. Utilizando las celdas de la Figura A.13, sumar dos números de 64 bits requerirá un viaje a través de siete celdas para calcular los P y G, y siete más para calcular los bits de acarreo, lo que requiere como mínimo 28 niveles lógicos. Cada paso de la multiplicación requerirá utilizar una vez este sumador. Una forma de evitar este cálculo en cada paso es utilizar *sumadores sin propagación de acarreo. (carry-save adder)* (CSA). Un sumador sin propagación de acarreo es simplemente un conjunto de n sumadores completos independientes. Un multiplicador que utiliza tal sumador se ilustra en la Figura A.20. Cada círculo marcado «A» es un sumador completo de un bit, y cada caja representa un bit de un registro. Cada operación de suma da como resultado un par de bits, almacenados en las partes de suma y acarreo de P. Como cada suma es independiente, sólo intervienen dos niveles lógicos en la suma —una gran mejora sobre 28.

Para operar, el multiplicador de la Figura A.20 carga los bits de suma y acarreo de P con cero y realiza la primera operación de la ALU. (Si se utiliza la recodificación de Booth, puede ser una resta en lugar de una suma.) Después, desplazar el bit de suma de orden inferior de P a A, a la vez que despla-

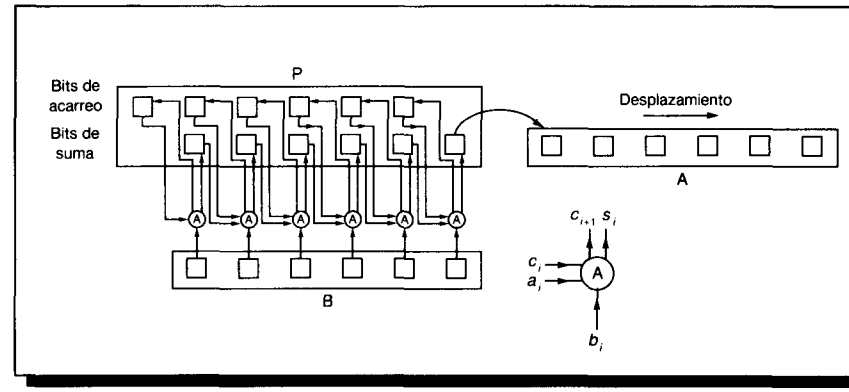


FIGURA A.20 Multiplicador sin propagación de acarreo. Cada círculo representa un sumador (3,2) trabajando independientemente. En cada paso, el único bit de P que necesita desplazarse es el bit de suma de orden inferior.

za A. Los $n - 1$ bits de orden superior de P no necesitan ser desplazados, porque en el siguiente ciclo los bits de suma alimentan el siguiente sumador de orden inferior. La velocidad de cada paso de suma aumenta de manera espectacular, ya que cada celda de la suma está trabajando independientemente de las otras, y no se propaga ningún acarreo. Hay dos inconvenientes para los sumadores sin propagación de acarreo. Primero, requieren más hardware, porque debe haber una copia del registro P, para que contenga las salidas de acarreo del sumador. Segundo, después del último paso, la palabra de orden superior del resultado se debe conectar a un sumador ordinario para combinar las partes de suma y acarreo. Esto se podría realizar conectando la salida de P en el sumador utilizado para realizar la operación de suma. Multiplicar con un sumador sin propagación de acarreo, a veces, se denomina multiplicación redundante, porque P se representa utilizando dos registros. Ya que hay múltiples formas para representar P como suma de dos registros, esta representación es redundante. El término *sumador con propagación de acarreo (carry-propagate adder)* (CPA) se utiliza para denotar un sumador que no es un CSA. Un sumador de propagación puede propagar sus acarreos utilizando transmisión de acarreo, anticipación de acarreo o algún otro método.

Otra forma de acelerar la multiplicación sin utilizar sumadores extra es examinar, en cada paso, los k bits de orden inferior de A, en lugar de exactamente un bit. Esto, con frecuencia, se denomina *multiplicación de base superior (higher-radix)*. Como ejemplo, supongamos que $k = 2$. Si el par de bits es 00, sumar 0 a P, y si es 01, sumar B. Si es 10, simplemente desplazar b un bit a la izquierda antes de sumarlo a P. Desgraciadamente, si el par es 11, parece que habría que calcular $b + 2b$. Pero esto se puede evitar utilizando una versión de base superior de la recodificación de Booth. Imaginar A como un número de base 4: cuando aparezca el dígito 3, cambiarlo a $\bar{1}$ y sumar 1 al siguiente dígito más elevado para compensar. El nombre de esta técnica, *solapamiento de tripletas*, proviene del hecho de que se examinan 3 bits para determinar qué múltiplo de b utilizar, mientras que la recodificación ordinaria de Booth examina 2 bits.

Las reglas precisas para el solapamiento de tripletas se dan a la Figura A.21. A pesar de tener lógica de control más compleja, esta técnica también requiere que el registro P sea un bit más ancho para acomodar la posibilidad de que se sumen a él $2b$ o $-2b$. También es posible utilizar una versión de base 8 (o incluso mayor) de la recodificación de Booth. Sin embargo, en ese caso, será necesario utilizar el múltiplo 3B como sumando potencial. Los multiplicadores de base 8 normalmente calculan 3B una vez, y para siempre, al comienzo de una operación de multiplicación.

Multiplicación más rápida con muchos sumadores

Si hay disponible espacio para muchos sumadores, entonces la velocidad de la multiplicación puede mejorarse. La Figura A.22 muestra un diagrama de bloques de un sencillo *multiplicador en array* para multiplicar dos números de 8 bits, utilizando siete CSA y un sumador de propagación. Como todavía necesita ocho sumas para calcular el producto, la latencia de calcular un producto no es muy diferente de utilizar un único sumador sin propagación de

Par actual		Anterior	Múltiple
$i+1$	i	$i-1$	
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

FIGURA A.21 Múltiplos de *b* para utilizar la recodificación de Booth en base 4. Por ejemplo, si los dos bits de orden inferior del registro A son 1, y el último bit que se desplazó fuera del registro A fue 0, entonces el múltiplo correcto es $-b$, obtenido de la penúltima fila de la tabla.

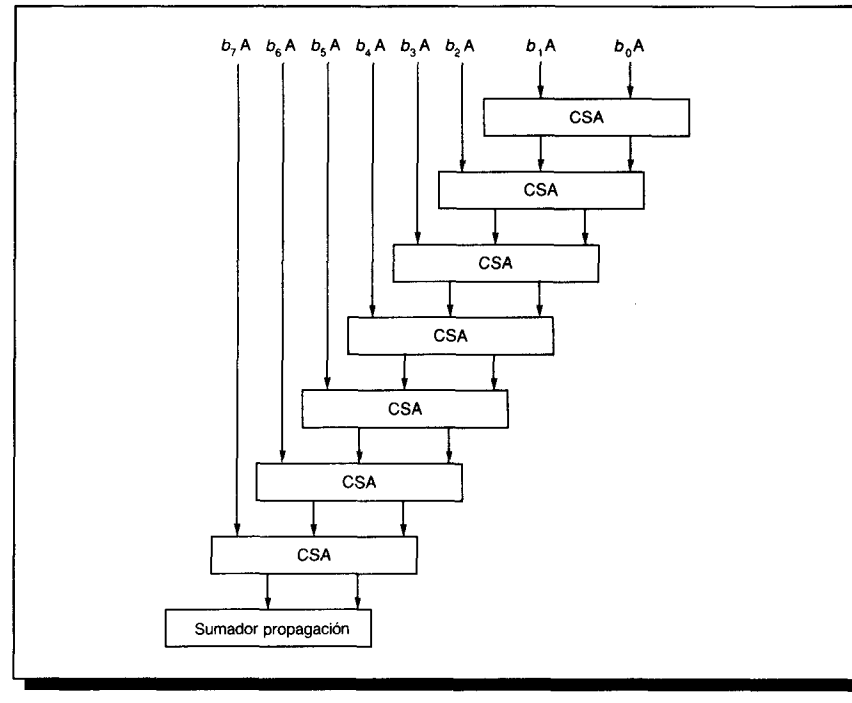


FIGURA A.22 Diagrama de bloques de un multiplicador de arrays. El número de 8 bits de A es multiplicado por $b_7b_6...b_0$. Cada caja marcada «CSA» es un sumador de ahorro de acarreo.

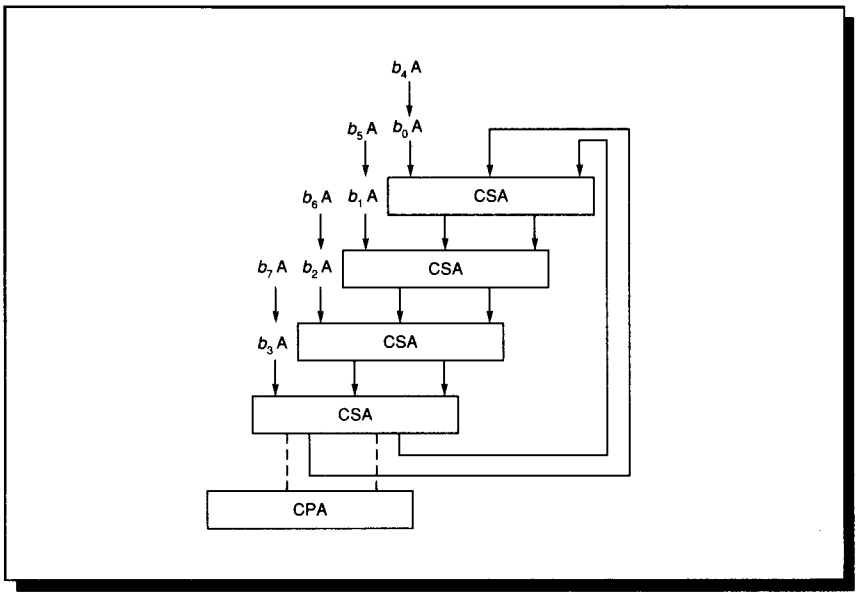


FIGURA A.23 Multiplicador en array multipaso. Multiplica dos números de 8 bits con aproximadamente la mitad del hardware que en la Figura A.22. Al final del segundo paso, los bits fluyen al CPA.

acarreo. Sin embargo, con el hardware de la Figura A.22, la multiplicación se puede segmentar incrementando la productividad total. Por otro lado, aunque este nivel de segmentación encauzada, a veces, se utilice en procesadores en array, no se utiliza en ninguno de los chips de los aceleradores de punto flotante explicados en la Sección A.10. La segmentación se explica, en general, en el Capítulo 6 y por Kogge [1981] en el contexto de los multiplicadores.

Con la tecnología de 1990, no es posible ubicar un array suficientemente grande para multiplicar dos números en doble precisión en un solo chip, y tener espacio sobrante para las demás operaciones aritméticas. Por tanto, un diseño popular es utilizar una organización de dos pasos como la mostrada en la Figura A.23. El primer paso a través del array «retira» cuatro bits de B. Después, el resultado de este primer paso se vuelve a conectar a la parte superior para combinarlo con los siguientes cuatro sumandos. El resultado de este segundo paso se conecta entonces a un CPA. Sin embargo, este diseño pierde la posibilidad de ser segmentado.

Si las organizaciones en array requieren tantos pasos de suma como la organización más barata de la Figura A.2, ¿por qué son tan populares? En primer lugar, utilizar un array tiene menor latencia que utilizar un solo sumador —como el array es un circuito combinacional, las señales fluyen a través de él, directamente, sin necesidad de reloj. Aunque el sumador de dos pasos de la Figura A.23 normalmente utiliza un reloj, la duración del ciclo para pasar a través de k arrays puede ser menor que k veces el ciclo de reloj que se necesitaría para un diseño como el de la Figura A.2. En segundo lugar, el array

es indicado para diversos esquemas para el aumento de la velocidad. Uno de ellos se muestra en la Figura A.24. La idea de este diseño es realizar dos sumas en paralelo o, dicho con otras palabras, que cada cadena pase solamente a través de la mitad de los sumadores. Por tanto, funciona a casi dos veces la velocidad del multiplicador de la Figura A.22. Este multiplicador *par/impar* es popular en VLSI a causa de su estructura regular. Los arrays también se pueden acelerar utilizando lógica asíncrona. Una de las razones por la cual el multiplicador de la Figura A.2 necesita un reloj es para que la salida del sumador no vuelva a alimentar la entrada del sumador antes de que aquélla se estabilice completamente. Por tanto, si el array de la Figura A.23 es suficientemente grande para que no se pueda propagar ninguna señal, desde la parte superior a la inferior, en el tiempo que emplea en estabilizarse el primer sumador, se pueden omitir los relojes. Williams y cols. [1987] explican un diseño utilizando esta idea, aunque sea para divisores en lugar de para multiplicadores.

Las técnicas del párrafo anterior todavía tienen un tiempo de multiplicación de O(n), pero se puede reducir a log n utilizando un árbol. El árbol más

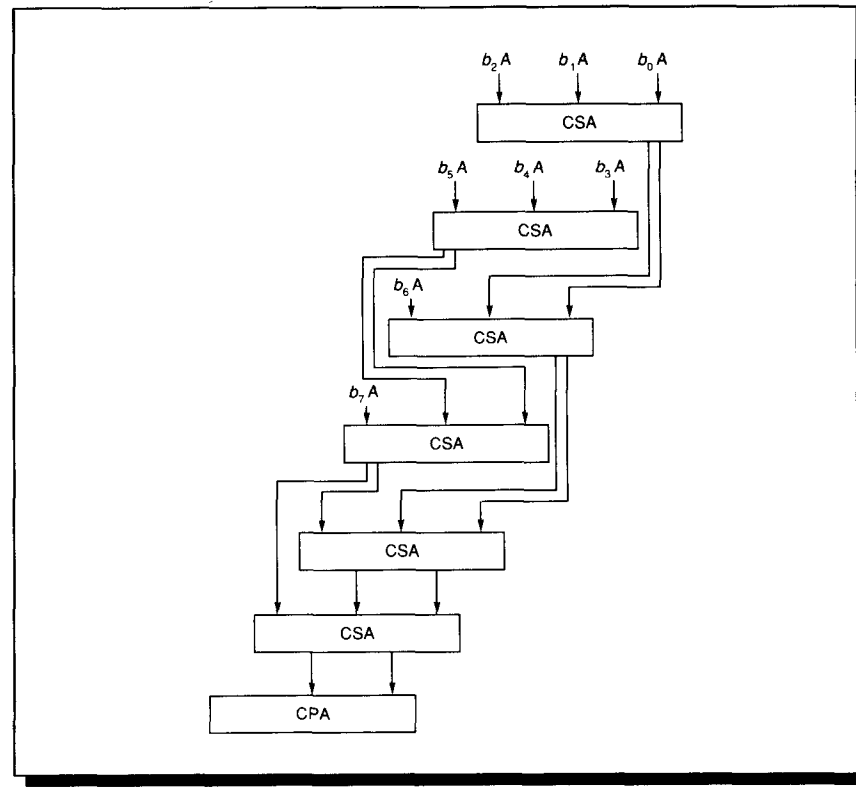


FIGURA A.24 Array par/impar. Los dos primeros sumadores trabajan en paralelo. Sus resultados se conectan a los sumadores tercero y cuarto, que también trabajan en paralelo y así sucesivamente.

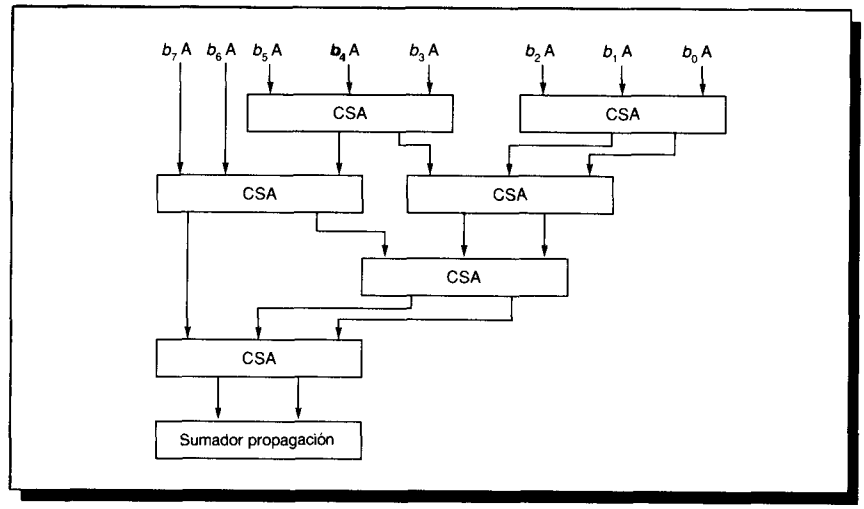


FIGURA A.25 Multiplicador mediante árbol de Wallace.

simple combinaría pares de sumandos b_0A ... b_{n-1}A, reduciendo el número de sumandos de n a $n/2$. Entonces estos $n/2$ números se sumarían de nuevo por pares, reduciéndolos a $n/4$ y así sucesivamente, y como resultado se obtendría una simple suma después de log n pasos. Sin embargo, esta sencilla idea del árbol binario no es adecuada para sumadores completos (3,2), que reducen tres entradas a dos en lugar de reducir dos entradas a una. Un árbol que hace uso de sumadores completos, conocido como *árbol de Wallace*, se muestra en la Figura A.25. Cuando las unidades aritméticas del computador se construían con partes MSI, un árbol de Wallace era el diseño elegido para los multiplicadores de alta velocidad. Sin embargo, hay un problema al implementarlos en VLSI.

Las Figuras A.22-A.24 son tan concisas que puede ser difícil visualizar todos los sumadores involucrados en un multiplicador en array. La Figura A.26 muestra cada sumador individual en un multiplicador en array de 4 bits. La Figura A.26(c) muestra cómo esas entradas son conectadas por sumadores. Cada fila de sumadores en A.26(c) corresponde a una sola caja en A.26(a). En implementaciones reales, el array se podría organizar como un cuadrado, no «torcido», como se muestra en la figura. (Alineando los bits del mismo peso en la misma columna, la figura es más comprensible.) Si se intenta detallar todos los sumadores el árbol de Wallace de la Figura A.25, se descubrirá que no tiene la estructura agradable y regular de la Figura A.26. Esta es la razón por la que los diseñadores de VLSI, con frecuencia, han escogido utilizar otros diseños log n, tal como el *multiplicador de árbol binario*, que se explica a continuación.

El problema de realizar sumas mediante un árbol binario es el de utilizar un sumador (2,1) que combina dos dígitos y produce una suma de un dígito. Debido a los acarreos, esto no es posible utilizando una notación binaria, pero puede hacerse con alguna otra representación. Utilizaremos la *representación*

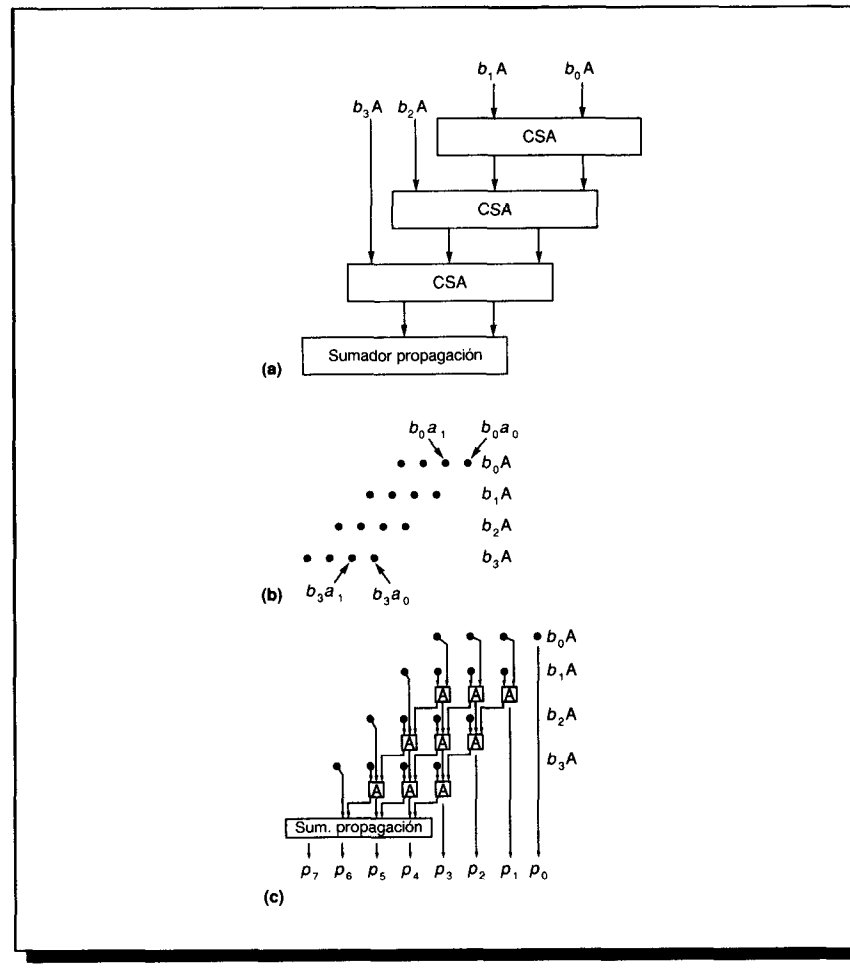


FIGURA A.26 Diagrama de bloques de un multiplicador en array (a); las entradas al array (b); el array expandido para mostrar todos los sumadores (c).

de dígitos con signo 1, $\bar{1}$ y 0, que utilizamos previamente para comprender el algoritmo de Booth. Esta representación tiene dos costes. Primero, emplea dos bits para representar cada dígito con signo. Segundo, el algoritmo para sumar dos dígitos con signo a_i y b_i es complejo y requiere examinar $a_ia_{i-1}a_{i-2}$ y $b_ib_{i-1}b_{i-2}$. Aunque esto significa que se deben examinar nuevamente dos bits anteriores, en la suma binaria se puede necesitar examinar un número arbitrario de bits (debido a los acarreos).

Podemos describir el algoritmo para sumar dos dígitos con signo como sigue: primero, calcular la suma y bits de acarreo s_i y c_{i+1} utilizando la tabla de la Figura A.27. Después calcular la suma final como s_i y c_i. Las tablas se inicializan para que esta suma final no genere acarreo.

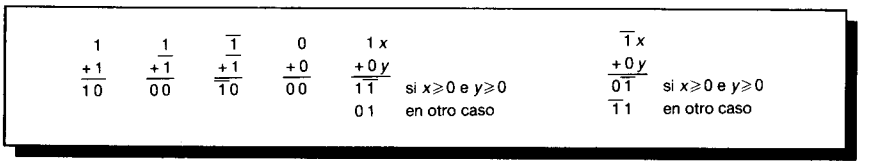


FIGURA A.27 Tabla de suma de dígitos con signo. La suma de más a la izquierda muestra que cuando se calcula 1 + 1, el bit de suma es 0 y el bit de acarreo es 1.

Ejemplo

¿Cuál es la suma de los números con signo $1\bar{1}0$ y 001?

Respuesta

Los dos bits de orden inferior suman $0 + 1 = 1\bar{1}$, el siguiente par suma $\bar{1} + 0 = 0\bar{1}$, y el par de orden superior suma $1 + 0 = 01$, así la suma es $1\bar{1} + 0\bar{1}0 + 0100 = 10\bar{1}$.

Esto define, entonces, un sumador (2,1). Teniéndolo en cuenta, podemos utilizar un sencillo árbol binario para realizar la multiplicación. En el primer paso se suma $b_0A + b_1A$ en paralelo con $b_2A + b_3A$, ..., $b_{n-2}A + b_{n-1}A$. El siguiente paso se suma el resultado de estas sumas por parejas, y así sucesivamente. Aunque la suma final se deba ejecutar en un sumador con propagación de acarreo para convertirla de la forma de dígitos con signo a complemento a dos, el paso final de suma es necesario en cualquier multiplicador utilizando CSA.

Resumiendo, ambos árboles, el de Wallace y el de dígitos con signo, son multiplicadores log n. El árbol de Wallace utiliza menor número de puertas pero es más difícil de organizar. El árbol de dígitos con signo tiene una estructura más regular, pero requiere dos bits para representar cada dígito y tiene una lógica de suma más complicada. Como con los sumadores, es posible combinar diferentes técnicas de multiplicar. Por ejemplo, se pueden combinar la recodificación de Booth y los arrays. En la Figura A.22, en lugar de que cada entrada sea b_iA, podríamos tener $b_ib_{i-1}A$, y con el fin de evitar el cálculo del múltiplo $3b$, podemos utilizar la recodificación de Booth.

División más rápida con un sumador

Las dos técnicas para acelerar la multiplicación con un solo sumador fueron los sumadores sin propagación de acarreo y la multiplicación de base superior. Hay una dificultad cuando se tratan de utilizar estas aproximaciones para acelerar la división sin restauración. El problema con los CSA es que al final de cada ciclo el valor de P, como está en la forma requerida para evitar la propagación de acarreo, no se conoce exactamente. En particular, el signo de P es incierto. Sin embargo, el signo de P se utiliza para calcular el dígito del cociente y decidir sobre la siguiente operación de la ALU. Cuando se utiliza una base superior, el problema es decidir qué valor restar de P. En el método de papel y lápiz, se tiene que estimar el dígito del cociente. En la división binaria sólo hay dos posibilidades; solucionamos el problema suponiendo inicialmente una y después ajustando la hipótesis basada en el signo de P. Esto no

funciona con bases más altas porque hay más de dos dígitos posibles para el cociente, presentándose una selección del cociente potencialmente bastante complicada: se han de calcular todos los múltiplos de b y compararlos con P.

Ambas técnicas, la de evitar acarreos y la división de base superior, pueden funcionar si utilizamos una representación redundante del cociente. Recordar que en la discusión de la división SRT, al permitir que los dígitos del cociente fuesen −1, 0, ó 1, había con frecuencia una posibilidad de elección. La idea en el algoritmo anterior era escoger cero siempre que fuese posible porque eso significaba que podía evitar una operación de la ALU. En la división sin propagación de acarreo, la idea es que como no se conoce exactamente el resto (registro P) (que está almacenado en la forma requerida para evitar la propagación de acarreo), tampoco se conoce el dígito exacto del cociente. Pero gracias a la representación redundante, el resto no se tiene que conocer precisamente, con el fin de escoger un dígito del cociente. Esto se ilustra en la Figura A.28, donde el eje x representa r_i, el contenido del par de registros (P,A) después de i pasos. La línea etiquetada $q_i = 1$ muestra el valor que debería tener r_{i+1} si escogiésemos $q_i = 1$, y análogamente para las líneas $q_i = 0$ y $q_i = -1$. Podemos escoger cualquier valor de q_i, siempre que $r_{i+1} = rP_i - q_iB$ satisfaga $|r_{i+1}| \leqslant B$. Los rangos permisibles se muestran en la mitad derecha de la Figura A.28. Entonces sólo necesitamos conocer r con suficiente precisión para decidir en qué rango de la Figura A.28 se encuentra.

Esta es la base para utilizar sumadores sin propagación de acarreo. Examinar los bits de orden superior del sumador sin propagación de acarreo y sumarlos en un sumador con propagación. Después, utilizar esta aproximación de r para calcular q_i, habitualmente por medio de una tabla. La misma técnica funciona para la división de base superior (independientemente de que se utilice un sumador sin propagación de acarreo). Los bits de orden superior de P se pueden utilizar para indexar una tabla que proporcione uno de los dígitos permisibles del cociente.

El desafío de diseño cuando se construye un divisor SRT de alta velocidad

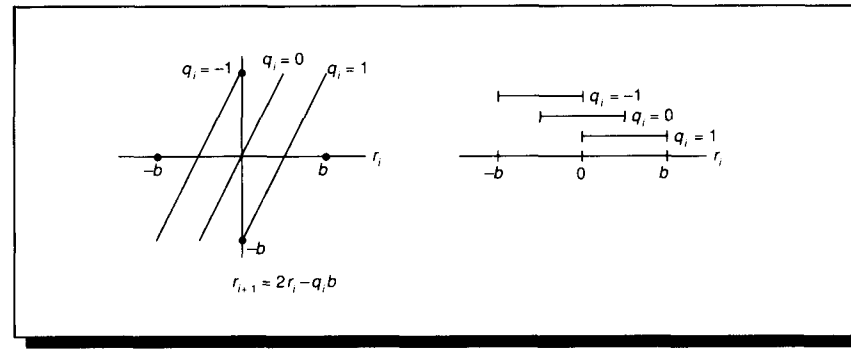


FIGURA A.28 Selección de cocientes para una división en base 2. El eje x representa el $i$ésimo resto, que es la cantidad en el par de registros (P,A). El eje y muestra el valor del resto después de un paso adicional de división. Cada barra en el gráfico de la derecha da el rango de valores r_i para los cuales es permisible seleccionar el valor asociado de q_i.

es determinar cuántos bits de P y B necesitan ser examinados. Por ejemplo, suponer que tomamos una base 4; se utilizan como dígitos del cociente 2, 1, 0, $\bar{1}$, $\bar{2}$, pero tenemos un sumador de propagación. ¿Cuántos bits de P y B necesitan ser examinados? Decidir esto involucra dos pasos. Para la división ordinaria sin restauración de base 2, debido a que en cada etapa $|r| \leqslant b$, el buffer P nunca causa desbordamiento. Pero para base 4, $r_{i+1} = 4r_i - q_i b$ se calcula en cada etapa, y si r_i es próximo a b, entonces, $4r_i$ será próximo a $4b$, incluso el dígito mayor del cociente no traerá a r al rango $|r_{i+1}| \leqslant b$. En otras palabras, el resto puede crecer sin límites. Sin embargo, restringiendo $|r_i| \leqslant 2b/3$ hace fácil comprobar que r_i estará acotado.

Después de determinar el límite que r_i debe satisfacer, podemos dibujar el diagrama de la Figura A.29, que es análogo a la Figura A.28. Si r_i está entre $(1/12)b$ y $(5/12)b$, podemos escoger $q = 1$, y así sucesivamente. O, puesto de otra forma, si r/b está entre 1/12 y 5/12, podemos escoger $q = 1$. Suponer que examinamos 4 bits de P y 4 bits de b, y que los bits superiores de P (sin contar el $(n + 1)$-ésimo bit de signo) son $0011xxx...$, mientras que los bits superiores de b son $1001xxx...$. Para simplificar el cálculo, imaginar el punto binario en el extremo izquierdo de cada registro. Como truncamos, r (el valor de P concatenado con A) podría tener un valor entre 0,0011 y 0,0100, y b podría tener un valor entre 0,1001 y 0,1010. Por tanto, r/b puede ser tan pequeño como 0,0011/0,1010 o tan grande como 0,0100/0,1001. Pero $0{,}0011_2/0{,}1010_2 = 3/10 < 1/3$ requeriría un bit de cociente de 1, mientras que $0{,}0100_2/0{,}1001_2 = 4/9 > 5/12$ requeriría un bit de cociente de 2. En otras palabras, 4 bits de P y 4 bits de b no son suficientes para escoger un bit de cociente. Se puede comprobar que 5 bits de P y 4 bits de b son suficientes. Esto puede verificarse escribiendo un sencillo programa que compruebe todos los casos.

Ejemplo

Suponer que la base es 4 y los dígitos del cociente son 2, 1, 0, $\bar{1}$, $\bar{2}$, pero esta vez se utiliza un CSA en lugar de un sumador con propagación. ¿Cuántos bits de los registros P y B necesitan ser examinados?

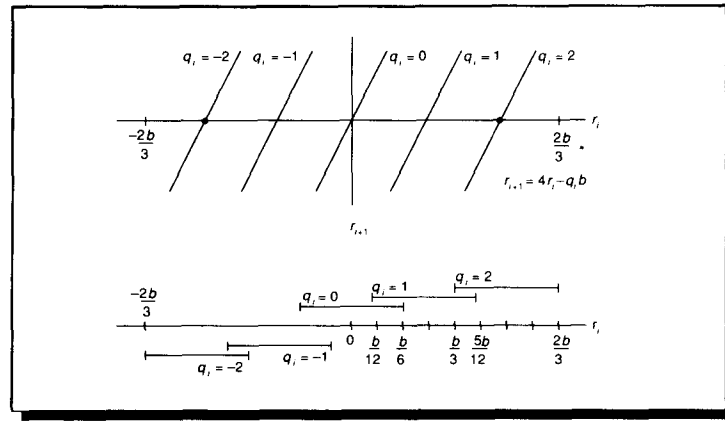


FIGURA A.29 Selección de cocientes para división en base-4.

Respuesta De nuevo $|r_i| \leqslant 2b/3$, y los rangos de q_i son todavía los de la Figura A.29. Si los 4 bits superiores de la parte de la suma y de la parte del acarreo de P son respectivamente 0010 y 0001, entonces la parte de la suma varía desde 0010 a 0011 y la parte de acarreo desde 0001 hasta 0010. Consiguientemente, el valor verdadero de r varía desde 0010 + 0001 = 0011 a 0011 + 0010 = 0101. Dado, por lo tanto, un CPA que sume los 4 bits superiores de las partes de acarreo y suma de P, y una suma de 0011, la suma verdadera estará en cualquier caso entre 0011 y 0101. Un programa que compruebe todos los casos mostrará que son necesarios 6 bits de P y 4 bits de b para predecir un dígito del cociente. El resultado de tal programa se muestra en la Figura A.30. Por ejemplo, si b es 1001*xxx*... y r es 001101*xxx*..., entonces los 4 bits superiores de b son 9 y los 6 bits superiores de r son 13, haciendo el dígito del cociente 1. Pero si r fuese $001110_2 = 14$, el dígito del cociente sería 2.

b	Rango de P		q	b	Rango de P		q
8	−21	−14	−2	12	−32	−20	−2
8	−13	−5	−1	12	−20	−7	−1
8	−5	3	0	12	−8	6	0
8	3	11	1	12	5	18	1
8	12	21	2	12	18	32	2
9	−24	−16	−2	13	−34	−21	−2
9	−15	−6	−1	13	−21	−7	−1
9	−6	4	0	13	−8	6	0
9	4	13	1	13	5	19	1
9	14	24	2	13	19	34	2
10	−26	−17	−2	14	−37	−22	−2
10	−16	−6	−1	14	−23	−7	−1
10	−6	4	0	14	−9	7	0
10	4	14	1	14	5	21	2
10	15	26	2	14	20	37	2
11	−29	−18	−2	15	−40	−24	−2
11	−18	−6	−1	15	−25	−8	−1
11	−7	5	0	15	−10	8	0
11	4	16	1	15	6	23	1
11	16	29	2	15	22	40	2

FIGURA A.30 Dígitos de cociente para división SRT en base 4 con un CSA. La fila superior indica que si los 4 bits de orden superior de b son $1000_2 = 8$, y los 6 bits superiores de P están entre $110010_2 = -14$ y $10101_2 = -21$, entonces el dígito del cociente es −2.

Aunque éstos son casos simples, todos los análisis de SRT se realizan de la misma forma. Primero calculan el rango de r_i, después dibujan r_i, frente a r_{i+1} para calcular los rangos del cociente, y finalmente escriben un programa para calcular el número de bits que es necesario. (También, a veces, es posible calcular el número de bits requeridos analíticamente.) Dos comentarios finales sobre la división SRT de base superior. Primero, la Figura A.30 no es simétrica. Por tanto, para un divisor CSA de base 4, la tabla no necesita sólo 6 bits de P, sino también el signo de P. Segundo, la tabla de cocientes tiene una estructura bastante regular. Esto significa que es habitualmente más barato codificarla como un PLA mejor que como una ROM.

A.10 Juntando todo

En esta sección, compararemos el Weitek 3364, el MIPS R3010 y el Texas Instruments 8847 (ver Figuras A.31 y A.32). Bajo muchos aspectos, éstos son chips ideales para comparar. Cada uno de ellos implementa el estándar del IEEE para suma, resta, multiplicación y división en un solo chip. Todos se introdujeron en 1988 y utilizan un ciclo de aproximadamente 40 ns. Sin embargo, como veremos, utilizan algoritmos bastante diferentes. El chip de Weitek está bien descrito en Birman y cols. [1988]; el chip MIPS está descrito con menos detalle en Rowen, Johnson y Ries [1988], y los detalles del chip de TI han de publicarse todavía.

Hay una serie de cosas que estos tres chips tienen en común. Realizan sumas y multiplicaciones en paralelo, y no implementan ni la precisión extendida ni la operación resto del IEEE. Anteriormente se explicó cómo puede

	MIPS R3010	Weitek 3364	TI 8847
Duración ciclo reloj (ns)	40	50	30
Tamaño (mil^2)	114 857	147 600	156 180
Transistores	75 000	165 000	180 000
Patillas	84	168	207
Potencia (vatios)	3,5	1,5	1,5
Ciclos/suma	2	2	2
Ciclos/mult.	5	2	3
Ciclos/división	19	17	11
Ciclos/raíz cuad.	—	30	14

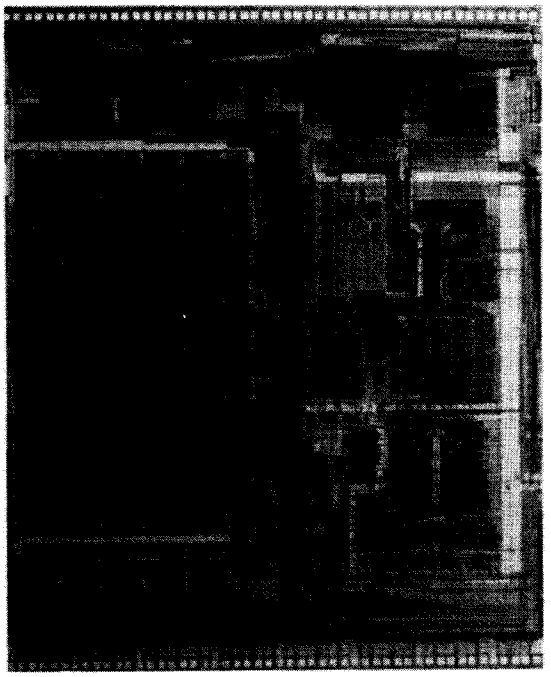
FIGURA A.31 Resumen de los tres chips de punto flotante explicados en esta sección. Las duraciones de los ciclos corresponden a las partes de producción disponibles en junio de 1989. El número de ciclos corresponde a operaciones en doble precisión.

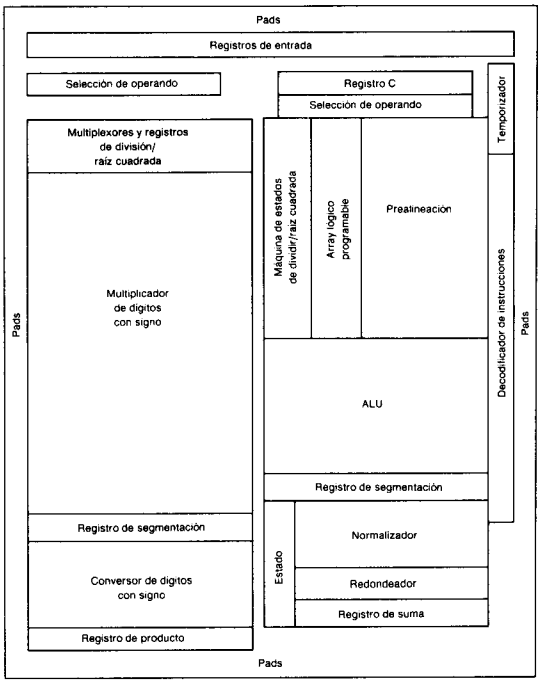
proporcionarse un REM (operación resto) eficiente en software cuando los chips implementan únicamente una función de paso-de-resto. Los diseñadores de estos chips, probablemente, decidieron no proporcionar precisión extendida, debido a que los usuarios más influyentes son aquellos que utilizan códigos portables, que no pueden contar con precisión extendida. Sin embargo, como hemos visto, la precisión extendida puede emplearse para bibliotecas matemáticas más rápidas y más simples.

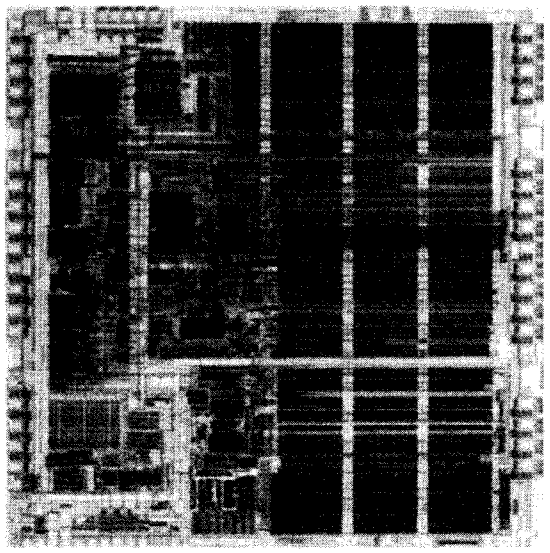
Un resumen de los tres chips se da en las Figuras A.31 y A.32. Observar que un número más alto de transistores, generalmente, conduce a un número menor de ciclos. Comparar los números de ciclos/op necesarios, debe hacerse cuidadosamente porque las cifras para el chip de MIPS son las de un sistema completo (par R3000/3010), mientras que los números de Weitek y TI son para chips autónomos, y son habitualmente mayores cuando se usan en un sistema completo.

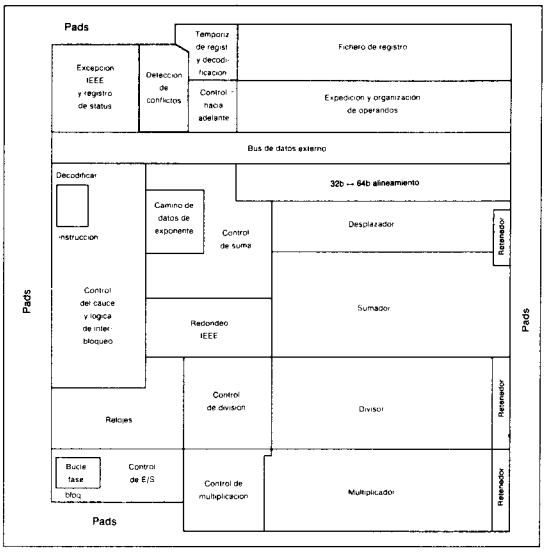
El chip de MIPS tiene el menor número de transistores de los tres. Esto se refleja en el hecho de que es el único chip de los tres que no tiene segmentación ni raíz cuadrada por hardware. Además, las operaciones de multiplicación y suma no son completamente independientes porque comparten el sumador con propagación de acarreo que realiza el redondeo final (así como el redondeo lógico). Además, en el R3010 se utiliza una mezcla de transmisión, CLA, y selección de acarreo. El sumador con selección de acarreo se utiliza como indica la Figura A.16. En cada mitad, los acarreos se propagan utilizando un esquema híbrido transmisión-CLA del tipo indicado en la Figura A.14. Sin embargo, éste se afina posteriormente variando el tamaño de cada bloque, en lugar de tener cada uno fijo a cuatro bits (como están en la Figura A.14). El multiplicador está a mitad de camino entre los diseños de la Figura A.2 y A.22. Tiene un array lo suficientemente grande para que la salida pueda conectarse de nuevo a la entrada sin necesidad de reloj. También, utiliza la recodificación de Booth de base 4 y la técnica par-impar de la Figura A.24. El R3010 puede hacer una división y multiplicación en paralelo (como el chip de Weitek, pero no el de TI). El divisor utiliza el método SRT de base 4 con dígitos de cociente -2, -1, 0, 1 y 2, y es similar al descrito en Taylor [1985]. La división en doble precisión es aproximadamente cuatro veces más lenta que la multiplicación. El R3010 pone de manifiesto que, para los chips que utilizan un multiplicador de $O(n)$, un divisor SRT puede operar suficientemente rápido para conseguir una relación razonable entre multiplicación y división.

El Weitek 3364 tiene unidades independientes de suma, multiplicación y división, y también utiliza la división SRT de base 4. Sin embargo, las operaciones de suma y multiplicación en el chip Weitek están segmentadas. Las tres etapas de la suma son: (1) comparar exponentes; (2) sumar seguido por desplazamiento (o viceversa), y (3) redondeo final. Las etapas (1) y (3) emplean sólo medio ciclo, permitiendo que la operación completa se haga en dos ciclos, aun cuando la segmentación consta de tres etapas. El multiplicador utiliza un array del estilo de la Figura A.23, pero utiliza la recodificación de Booth, de base 8, lo que significa que debe calcular 3 por el multiplicador. Las tres etapas de la segmentación del multiplicador son: (1) calcular $3b$; (2) pasar a través del array, y (3) redondeo y suma final con propagación de acarreo. La precisión simple pasa a través del array una vez; la doble precisión dos veces.




Pads
Registros de entrada
Selección de operando
Registro C
Selección de operando
Temporizador
Multiplexores y registros de división/ raíz cuadrada
Maquina de estados de dividir/raiz cuadrada
Array lógico programable
Prealineación
Multiplicador de dígitos con signo
Decodificador de instrucciones
ALU
Registro de segmentación
Registro de segmentación
Normalizador
Conversor de dígitos con signo
Estado
Redondeador
Registro de suma
Registro de producto
Pads
Pads
Pads




Pads
Temporiz de regist y decodificacion
Excepcion IEEE y registro de status
Deteccion de conflictos
Control hacia adelante
Fichero de registro
Expedicion y organización de operandos
Bus de datos externo
Decodificar
32b -- 64b alineamiento
Camino de datos de exponente
Control de suma
Desplazador
Retenedor
instruccion
Control del cauce y logica de inter bloqueo
Sumador
Redondeo IEEE
Pads
Pads
Control de division
Divisor
Retenedor
Relojes
Bucle fase
Control de E/S
Control de multiplicacion
Multiplicador
Retenedor
Pads

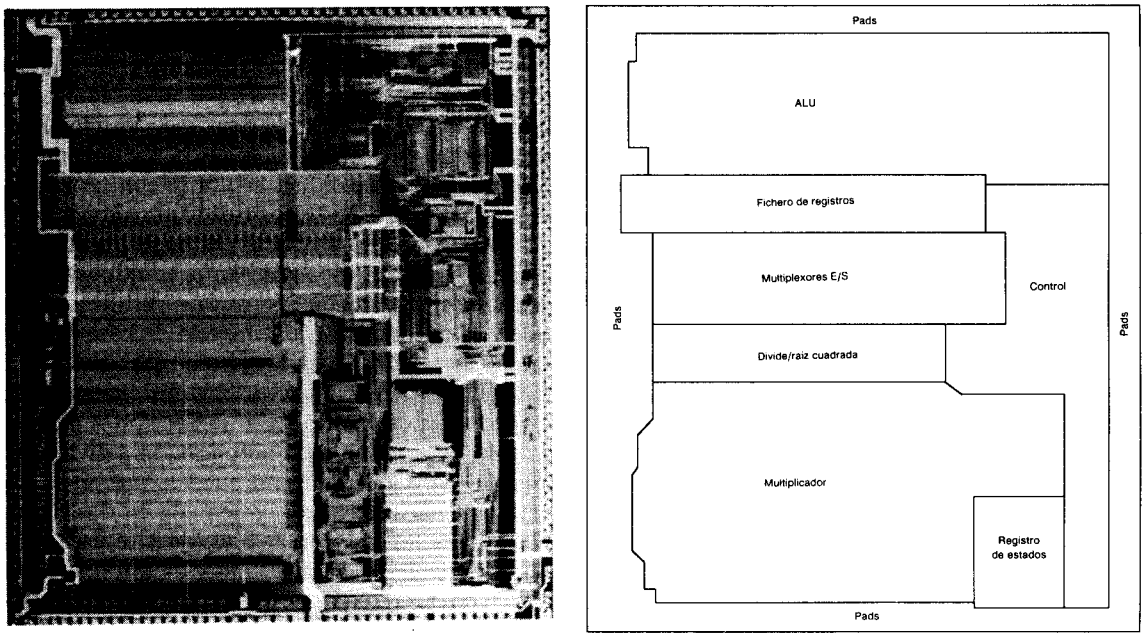


FIGURA A.32 Organización física (layout) de los chips. En la columna de la izquierda están las fotomicrografías; la columna de la derecha muestra los correspondientes planos de base. La parte superior izquierda es el TI 8847, la inferior izquierda es el MIPS R3010, y la superior en esta página es el Weitek 3364.

Como en la suma, la latencia es de dos ciclos. El chip Weitek utiliza un algoritmo de suma interesante. Es una variante del sumador con salto de acarreo dibujado en la Figura A.15. Sin embargo, P_{ij}, que es la AND lógica de muchos términos, se calcula en cascada, realizando una AND por nivel. Por tanto, mientras los acarreos se propagan hacia la izquierda dentro de un bloque, el valor de P_{ij} se está propagando dentro del siguiente bloque, y los tamaños de bloque se escogen para que ambas propagaciones se completen al mismo tiempo. De forma distinta al chip MIPS, el 3364 tiene raíz cuadrada por hardware, que utiliza el hardware de la división. La relación de multiplicación de doble precisión a división es 2:17. La gran disparidad entre multiplicación y división se debe al hecho que la multiplicación utiliza la recodificación de Booth de base 8, mientras que la división utiliza un método de base 4. En el MIPS R3010, la multiplicación y división utilizan la misma base.

La característica notable del TI 8847 es que hace la división por iteración (utilizando el algoritmo de Goldschmidt explicado en la Sección A.6). Esto mejora la velocidad de la división (la relación de multiplicación a división es 3:11), pero significa que multiplicación y división no pueden hacerse en paralelo como en los otros dos chips. La suma está segmentada en dos etapas. Comparar exponentes, desplazar fracciones y sumar fracciones se hace en la primera etapa, la normalización y redondeo en la segunda. La multiplicación utiliza un árbol binario de sumadores de dígitos con signo y tiene una segmentación de tres etapas. La primera etapa pasa a través del array usando la mitad de los bits; la segunda etapa pasa a través del array una segunda vez, y la tercera etapa convierte los dígitos con signo a complemento a dos. Como sólo hay un array, una nueva operación de multiplicación puede sólo ser iniciada en ciclos alternos. Sin embargo, ralentizando el reloj, los dos pasos a través del array pueden hacerse en un solo ciclo. En este caso, se puede iniciar una nueva multiplicación cada ciclo. El sumador del 8847 utiliza un algoritmo de selección de acarreo en lugar de anticipación del acarreo. Como mencionamos en la Sección A.6, el TI utiliza 60 bits de precisión con el fin de realizar correctamente el redondeo de la división.

Estos tres chips ilustran las diferentes tendencias realizadas por diseñadores con restricciones similares. Una de las cosas más interesantes sobre estos chips es la diversidad de sus algoritmos. Cada uno utiliza un algoritmo diferente de suma, así como un algoritmo diferente de multiplicación. En efecto, la recodificación de Booth es la única técnica que es universalmente utilizada por todos los chips.

A.11 Falacias y pifias

Falacia: El desbordamiento a cero raramente se presenta en el código de aplicaciones reales de punto flotante.

Aunque muchos códigos raramente tienen desbordamiento a cero, frecuentemente, hay códigos reales con desbordamiento a cero frecuente. SDRWAVE [Kahaner, 1988], que resuelve una ecuación de onda unidimensional, es un ejemplo. Este programa causa desbordamiento con bastante frecuencia, aun

cuando funcione adecuadamente. Las medidas sobre una máquina muestran que añadir soporte hardware para desbordamiento a cero gradual podría provocar que SDRWAVE se ejecutase aproximadamente el 50 por 100 más rápido.

Falacia: Las conversiones entre entero y punto flotante son raras.

En efecto, en Spice son tan frecuentes como las divisiones. La suposición de que las conversiones son raras conduce a un error en el repertorio de instrucciones de SPARC, que no proporciona ninguna instrucción que transfiera de registros enteros a registros en punto flotante.

Pifia: No incrementar la velocidad de una unidad de punto flotante sin incrementar su ancho de banda con memoria.

Un uso típico de una unidad de punto flotante es sumar dos vectores para producir un tercer vector. Si estos vectores constan de números en doble precisión, entonces cada suma en punto flotante utilizará tres operandos de 64 bits cada uno, o 24 bytes de memoria. Los requerimientos de ancho de banda de memoria son aún mayores si la unidad de punto flotante puede realizar sumas y multiplicaciones en paralelo (como hace la mayoría).

Pifia: $-x$ no es lo mismo que $0 - x$.

Esto es un refinamiento en el estándar del IEEE que ha equivocado a algunos diseñadores. Como los números en punto flotante utilizan el sistema signo/magnitud, hay dos ceros, $+0$ y -0. El estándar dice que $0 - 0 = +0$, mientras que $-(0) = -0$. Entonces $-x$ no es lo mismo que $0 - x$ cuando $x = 0$.

A.12 Perspectiva histórica y referencias

Los primeros computadores utilizaban punto fijo en lugar de punto flotante. En «Preliminary Discussion of the Logical Design of an Electronic Computing Instrument», Burks, Goldstine y von Neumann dicen lo siguiente:

Parece que hay dos propósitos importantes en un sistema decimal de punto «flotante», los cuales surgen del hecho de que el número de dígitos de una palabra sea una constante fijada por consideraciones de diseño para cada máquina particular. El primer propósito es mantener en una suma o producto tantos dígitos significativos como sea posible y el segundo es liberar al operador humano de la carga de estimar e insertar en un problema «factores de escala» —constantes multiplicativas que sirven para mantener a los números dentro de los límites de la máquina.

Por supuesto, no negamos el hecho de que el tiempo humano se emplea en organizar la introducción de factores de escala aconsejables. Solamente argüimos que el tiempo consumido así es un porcentaje muy pequeño de todo el tiempo que emplearemos en preparar un problema interesante para nuestra

máquina. La primera ventaja del punto flotante es, pensamos, algo ilusorio. Con el fin de tener un punto flotante, se debe malgastar capacidad de memoria que en cualquier otro caso se podría utilizar para obtener más dígitos por palabra. Por tanto, no nos parece nada claro si las modestas ventajas de un punto binario flotante compensa la pérdida de capacidad de memoria y la complejidad creciente de los circuitos aritméticos y de control. [Bell y Newell, 1971, 97].

Esto nos da la posibilidad de ver las cosas desde la perspectiva de los primeros diseñadores de computadoras, que pensaban que ahorrar tiempo de computador y memoria era más importante que ahorrar tiempo de programador.

Los artículos originales que introdujeron el árbol de Wallace, la recodificación de Booth, la división SRT, las tripletas solapadas, y así sucesivamente, se encuentran en Swartzlander [1980]. Una buena explicación de una de las primeras máquinas (el IBM 360/91) que utilizó un árbol de Wallace segmentado, recodificación de Booth y división iterativa está en Anderson y cols. [1967]. Una explicación del tiempo medio para la división SRT bit a bit está en Freiman [1961]; éste es uno de los pocos artículos históricos interesantes que no aparecen en Swartzlander.

El libro estándar de Mead y Conway [1980] no recomendaba el uso de CLA por no tener una buena relación coste/rendimiento en VLSI. Brent y Kung [1982] fue un importante artículo que ayudó a combatir esa visión. Un ejemplo de una organización detallada para CLA puede encontrarse en Ngai e Irwin [1985] o en Weste y Eshraghian [1985]. Takagi, Yasuura y Yajima [1985] proporcionan una descripción detallada de un multiplicador en árbol de dígitos con signo.

Aunque el estándar del IEEE se está adoptando ampliamente, todavía hay otros tres importantes sistemas de punto flotante en uso: el IBM/370, el DEC VAX y el Cray. Brevemente, explicaremos estos formatos más antiguos. El formato VAX es el más próximo al estándar del IEEE. Su formato de simple precisión (formato F) es como la simple precisión del IEEE, ya que tiene un bit oculto, 8 bits de exponente y 23 bits de fracción. Sin embargo, no tiene un bit retenedor, que hace que se redondee hacia arriba en los casos a mitad en lugar de al par. El VAX tiene un rango de exponentes ligeramente diferente que el del IEEE: $E_{\text{mín}}$ es -128 en lugar de -126 como en el IEEE y $E_{\text{máx}}$ es 126 en lugar de 127. Las diferencias principales entre VAX e IEEE son la falta de valores especiales y desbordamiento a cero gradual. El VAX tiene un operando reservado, pero funciona como un NaN de señalización: causa un trap siempre que es referenciado. Originalmente, la dobe precisión de VAX (formato D) tenía también 8 bits de exponente. Sin embargo, como éste es muy pequeño para muchas aplicaciones, se añadió un formato G; como el estándar del IEEE, este formato tiene 11 bits de exponente. El VAX también tiene un formato H, que es de 128 bits.

El formato de punto flotante de IBM/370 utiliza la base 16 en lugar de la base 2. Esto significa que no puede utilizar un bit oculto. En simple precisión, tiene 7 bits de exponente y 24 bits (6 dígitos hexadecimales) de fracción. Por tanto, el número mayor es $16^{2^7} = 2^{4\times 2^7} = 2^{2^9}$, comparado con 2^{2^8} para el IEEE. Sin embargo, un número que está normalizado en el sentido hexadecimal sólo necesita tener un dígito distinto de cero al principio. Cuando se interpreta en

binario, los tres bits más significativos podrían ser cero. Por tanto, hay potencialmente menos de 24 bits significativos. La razón para utilizar la base más alta, fue minimizar la cantidad de desplazamientos requeridos cuando se suman números en punto flotante. Sin embargo, esto es menos importante en las máquinas actuales, donde el tiempo de suma en punto flotante es fijo, habitualmente, independientemente de los operandos. Otra diferencia entre la aritmética del 370 y la del IEEE es que el 370 no tiene dígitos de redondeo ni dígito de retención *(sticky)*, lo que significaque trunca en lugar de redondear. Por tanto, tras muchos cálculos, el resultado sistemáticamente será demasiado pequeño. De forma distinta a la aritmética VAX y del IEEE, cualquier patrón de bits es un número válido. Por tanto, las rutinas de biblioteca deben establecer convenios sobre qué devolver en caso de errores. En la biblioteca FORTRAN IBM, por ejemplo, $\sqrt{-4}$ devuelve 2.

La aritmética en los computadores Cray es interesante, porque está conducida por una motivación para el rendimiento más alto posible de punto flotante. Tiene un campo de exponente de 15 bits y un campo fraccionario de 48 bits. La suma en los computadores Cray no tiene dígito de guarda, y la multiplicación es aún menos precisa que la suma. Pensando en la multiplicación como una suma de p números, cada uno de longitud $2p$ bits, lo que los computadores Cray hacen es suprimir los bits de orden inferior de cada sumando. Así, analizar las características exactas de error de la operación de multiplicar no es fácil. Los recíprocos se calculan utilizando la iteración, y la división de a por b se hace multiplicando a por $1/b$. Los errores en la multiplicación y el cálculo de inversos se combinan para hacer no fiables los tres últimos bits de una operación de división. ¡Cómo mínimo, los computadores Cray sirven para mantener a los analistas numéricos en constante estado de alerta!

El proceso de estandarización del IEEE comenzó en 1977, inspirado principalmente por W. Kahan, y está basado parcialmente en el trabajo de Kahan con el IBM 7094 en la Universidad de Toronto [Kahan, 1968]. El proceso de estandarización fue un asunto lento, siendo el desbordamiento a cero gradual el causante de las mayores controversias. (Según Cleve Moler, a los visitantes de Estados Unidos se les recomendaba que no debían perderse Las Vegas, el Gran Cañón y la reunión del comité de estándares del IEEE). El estándar se aprobó finalmente en 1985. El Intel 8087 fue la primera implementación comercial importante del IEEE y apareció en 1981, antes que se terminase el estándar. Contiene características que fueron eliminadas en el estándar final, como los *projective bits*. Según Kahan, la longitud de la doble precisión extendida estaba basada en lo que se pudo implementar en el 8087. Aunque el estándar del IEEE no estaba basado en ningún sistema existente de punto flotante, muchas de sus características estaban presentes en (algunos) sistemas. Por ejemplo, el CDC 6600 reservaba patrones especiales de bits para INDEFINITE (INDEFINIDO) e INFINITY (INFINITO), mientras que la idea de números denormales apareció en Goldberg [1967], así como en Kahan [1968]. Kahan fue galardonado en 1989 con el premio Turing en reconocimiento de su trabajo sobre punto flotante.

Referencias

ANDERSON, S. F., J. G. EARLE, R. E. GOLDSCHMIDT, AND D. M. POWERS [1967]. «The IBM System/360 Model 91: Floating-point execution unit», *IBM J. Research and Development 11, 34-53. Reprinted in [Swartzlander, 1980].*

Good description of an early high-performance floating-point unit that used a pipelined Wallace-tree multiplier and iterative division.

ATKINS, D. E. [1968]. «Higher-radix division using estimates of the divisor and partial remainders», *IEEE Trans. on Computers* C-17:10, 925-934. Reprinted in [Swartzlander, 1980].

This is the standard reference for high-radix SRT division.

BELL, C. G., AND A. NEWELL [1971]. *Computer Structures: Readings and Examples*, McGraw-Hill, New York.

BIRMAN, M., G. CHU, L. HU, J. MCLEOD, N. BEDARD, F. WARE, L. TORBAN AND C. M. LIM [1988]. «Design of a high-speed arithmetic datapath», *Proc. ICCD: VLSI Computers and Processors*, 214-216.

Fairly detailed description of the Weitek 3364 floating-point chip.

BRENT, R. P., AND H. T. KUNG [1982]. «A regular layout for parallel adders», *IEEE Trans. on Computers* C-31, 260-264.

This is the paper that popularized CLA adders in VLSI.

BURKS, A. W., H. H. GOLDSTINE AND J. VON NEUMANN [1946]. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.*

CODY, W. J. [1988]. «Floating point standards: Theory and practice», in *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, R. E. Moore, (ed.), Academic Press, Boston, Mass., 99-107.

Presents a status of hardware and software implementations of the standar.

CODY, W. J., J. T. COONEN, D. M. GAY, K. HANSON, D. HOUGH, W. KAHAN, R. KARPINSKI, J. PALMER, F. N. RIS, AND D. STEVENSON [1984]. «A proposed radix- and word-length-independent standard for floating-point arithmetic», *IEEE Micro* 4:4, 86-100.

contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754.

COONEN, J. [1984]. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, Ph. D. Thesis, Univ. of Calif., Berkeley.

The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversión.

FREIMAN, C. V. [1961]. «Statistical analysis of certain binary division algorithms», *Proc. IRE* 49:1, 91-103.

Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.

GOLDBERG, D. [1989]. «Floating-point and computer systems», *Xerox Tech. Rep.* CSL-89-9. A versión of this paper will appear in *Coomputing Surveys.*

Contains an in-depth tutorial on the IEEE standard from the software point of view.

GOLDBERG, I. B. [1967]. «27 bits are not enough for 8-digit accuracy», *Comm. ACM 10:2, 105-106.*

This paper proposes using hidden bits and gradual underflow.

GOSLING, J. B. [1980]. *Design of Arithmetic Units for Digital Computers*, Springer-Verlag New York, Inc., New York.

A concise, well-written book, although it focuses on MSI designs.

HAMACHER, V. C., Z. G. VRANESIC, AND S. G. ZAKY [1984]. *Computer Organization* , 2nd ed., McGraw-Hill, New York.

Introductory computer architecture book with a good chapter on computer arithmetic.

HWANG, K [1979]. *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York.

This book contains the widest range of topics of the computer arithmetic books.

IEEE [1985]. «IEEE standard for binary floting-point arithmetic», *SIGPLAN Notices* 22:2, 9-25.

IEEE 754 is reprinted here.

KAHAN, W. [1968]. «7094-II system support for numerical analysis», *SHARE Secretarial Distribution* SSD-159.

This system had many features that were incorporated into the IEEE floating-point standard.

KAHANER, D. K. [1988]. «Benchmarks for «real» programs», *SIAM News* (November).

The benchmark presented in this article turns out to cause many underflows.

KNUTH, D. [1981]. *The Art of Computer Programming*, vol. II, 2nd ed., Addison-Wesley, Reading, Mass.
Has a section on the distribution of floating numbers.

KOGGE, P. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.
Has bried discussion of pipelined multiplier.

KOHN, L., AND S.-W. FU [1989]. «A 1 000 000 transistor microprocessor», *IEEE Int'l Solid-State Circuits Conf.*, 54-55.
A brief overview of the Intel 860, whose floating-point addition algorithm is discussed in Section A.4.

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS, AND D. ZURAS [1988]. «Integer multiplication and division on the HP Precision Architecture», *IEEE Trans. on Computers*, 37:8, 980-990.
Rationale for the integer-and divide-step instructions in the Precision architecture.

MEAD, C., AND L. CONWAY [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

NGAI, T.-F., AND M. J. IRWIN [1985]. «Regular, area-time efficient carry-lookahead adders», *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 9-15.
Describes a CLA adder like that of Figure A.13, where the bits flow up and then come back down.

PENG, V., S. SAMUDRALA, AND M. GAVRIELOV [1987]. «On the implementation of shifters, multipliers, and dividers in VLSI floating point units», *Proc. Eighth IEEE Symposium on Computer Arithmetic*, 95-102.
Highly recommended survey of different techniques actually used in VLSI designs.

ROWEN, C., M. JOHNSON, AND P. RIES [1988]. «The MIPS R3010 floating-point coprocessor», *IEEE Micro*, 53-62 (June).

SANTORO, M. R., G. BEWICK, AND M. A. HOROWITZ [1989]. «Rounding algorithms for IEEE multipliers», *Proc. Ninth IEEE Symposium on Computer Arithmetic*, 176-183.
A very readable discussion of how to efficiently implement rounding for floating-point multiplication.

SCOTT, N. R. [1985]. *Computer Number Systems and Arithmetic*, Prentice-Hall, Englewood Cliffs, N. J.

SWARTZLANDER, E., ED. [1980]. *Computer Arithmetic*, Dowden, Hutchison and Ross (distributed by Van Nostrand, New York).
A collection of historical papers.

TAKAGI, N., H. YASUURA, AND S. YAJIMA [1985]. «High-speed VLSI multiplication algorithm with a redundant binary addition tree», *IEEE Trans. on Computers* C-34:9, 789-796.
A discussion of the binary-tree signed multiplier that was the basis for the design used in the TI 8847.

TAYLOR, G. S. [1981]. «Compatible hardware for division and square root», *Proc. Fifth IEEE Symposium on Computer Arithmetic*, 127-134.
Good discussion of a radix-4 SRT division algorithm.

TAYLOR, G. S. [1985]. «Radix 16 SRT dividers with overlapped quotient selection stages», *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 64-71.
Describes a very sophisticated high-radix division algorithm.

WESTE, N., AND K. ESHRAGHIAN [1985]. *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, Mass.
This textbook has a section on the layouts of various kinds of adders.

WHILLIAMS, T. E., M. HOROWITZ, R. L. ALVERSON, AND T. S. YANG [1987]. «A self-timed chip for division», *Advanced Research in VLSI, Proc. 1987 Stanford Conf.*, The MIT Press, Cambridge, Mass.
Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.

EJERCICIOS

A.1 [15/15/20] <A.3> Representar los siguientes números como números en punto flotante del IEEE en simple precisión y doble precisión.

a. [15] 10

b. [15] 10,5

c. [20] 0,1

A.2 [10/15/20] <A.8> Completar los detalles de los diagramas de bloque para los siguientes sumadores.

a. [10] En la Figura A.11, mostrar cómo implementar las cajas «1» y «2» en función de puertas AND y OR.

b. [15] En la Figura A.14, ¿qué señales se necesitan que fluyan desde las celdas del sumador de la parte superior a las celdas «C»? Escribir las ecuaciones lógicas para la caja «C».

c. [20] Mostrar cómo extender el diagrama de bloques de A.13 para que produzca el bit de acarreo de salida c_8.

A.3 [15/15] <A.4> Suma en punto flotante.

a. [15] En un sistema decimal con $p = 5$, calcular $-4{,}5673 + 4{,}9999 \times 10^{-5}$ suponiendo redondeo al más próximo. Dar el valor de los dígitos de guarda y redondeo, y el bit retenedor.

b. [15] ¿Cuál es el valor de la suma para los otros tres modos de redondeo?

A.4 [15] <A.3> Mostrar que si no se utiliza desbordamiento a cero gradual, entonces no es cierto que $x \neq y$ si y sólo si $x - y \neq 0$.

A.5 [25] <A.9> Reescribir la Figura A.21 para la recodificación de Booth en base 8.

A.6 [15] <A.3> ¿Es la ordenación de los números en punto flotante no negativos igual que la de los enteros cuando se consideran también números denormalizados? ¿Qué ocurre si los números denormalizados se representan utilizando la representación aproximada *(wrapped)* mencionada en la Sección A.5?

A.7 [25/10] <A.2> Complemento a uno.

a. [25] Cuando se suman números en complemento a dos, se descarta el acarreo de salida del bit más significativo. Mostrar que en complemento a uno, se debe conectar el acarreo de salida al extremo inferior.

b. [10] Encontrar la regla para detectar desbordamiento en complemento a uno.

A.8 [15] <A.2> Las Ecuaciones A.2.1 y A.2.2 son para sumar dos números de n bits. Derivar ecuaciones similares para la resta, donde habrá un préstamo en lugar de un acarreo.

A.9 [15/20] <A.2> Más sobre complemento a uno.

a. [15] Una complicación que surge con la aritmética de complemento a uno es que

el cero tiene dos representaciones. Mostrar que aunque la forma negativa de cero no sea una entrada, el sumador de la Ecuación A.2.1 (siendo c_0 el acarreo de realimentación) puede todavía producir un cero negativo.

b. [20] Utilizar el hecho que $a + b = a - (-b)$ junto con el circuito restador del problema anterior para obtener un sumador diferente en complemento a uno. ¿Puede este sumador producir ceros negativos?

A.10 [20] <A.2> En una máquina que no detecte desbordamiento de enteros en hardware, mostrar cómo se podría detectar desbordamiento en una operación de suma con signo en software.

A.11 [25] <A.9> En el array de la Figura A.23, no se explota el hecho que un array pueda estar segmentado. ¿Se puede obtener un diseño que conecte la salida del CSA inferior en los CSA inferiores en lugar de en el superior, y que vaya más rápido que la organización de la Figura A.23?

A.12 [15] <A.9> Para la recodificación ordinaria de Booth, el múltiplo de b utilizado en el paso iésimo es sencillamente $a_{i-1} - a_i$. ¿Se puede encontrar una fórmula similar para la recodificación de Booth de base 4 (tripletas solapadas)?

A.13 [25/15/30[<A.9> Multiplicación con desplazamiento sobre ceros.

a. [25] ¿La recodificación de Booth incrementa siempre el número de ceros de un número? ¿Puede en alguna ocasión decrementar el número de ceros?

b. [15] Dado el número $a_{n-1}...a_0$, definir $c_0 = 0$, y definir c_1 para que sea el acarreo de salida de sumar a_i, a_{i-1} y c_{i-1}. Entonces la recodificación *de Booth modificada* da un número con los dígitos $A_i = a_i + c_i - 2c_{i+1}$. ¿Cuál es la recodificación de 01101?

c. [30] Mostrar que la recodificación de Booth modificada nunca decrementa el número de ceros.

A.14 [20/15/20/15/20/15] <A.6> Raíz cuadrada iterativa.

a. [20] Utilizar el método de Newton para obtener un algoritmo iterativo para la raíz cuadrada. La fórmula involucrará una división.

b. [15] ¿Cuál es la forma más rápida que se puede pensar para dividir un número en punto flotante por 2?

c. [20] Si la división es lenta, entonces la rutina de raíz cuadrada iterativa también será lenta. Utilizar el método de Newton con $f(x) = 1/x^2 - a$ para obtener un método que no utilice ninguna división.

d. [15] Suponer que la relación de división por 2 : suma punto flotante : multiplicación punto flotante es 1:2:4. ¿Qué relación entre el tiempo de multiplicación y división hace que cada iteración en el método de la Parte c sea más rápida que cada iteración en el método de la Parte a?

e. [20] Cuando se utilice el método de la Parte a, ¿cuántos bits se necesitan en la aproximación inicial con el fin de obtener exactitud de doble precisión después de 3 iteraciones? (Se puede ignorar el error de redondeo.)

f. [15] Suponer que cuando Spice se ejecuta en el TI 8847, emplea el 16,7 por 100 de su tiempo en la rutina de la raíz cuadrada (este porcentaje ha sido medido en otras máquinas). Utilizando los valores de la Figura A.31 y suponiendo 3 iteracio-

nes, ¿cuántas veces más lento sería Spice si la raíz cuadrada se implementase en software utilizando el método de la Parte a?

A.15 [30/10] <A.2> Este problema presenta un algoritmo para sumar números en signo y magnitud. Si A y B son enteros de signos opuestos, sean a y b sus magnitudes.

a. [30] Mostrar que las siguientes reglas para manipular los números sin signo a y b da $A + B$.
   1. Complementar uno de los operandos.
   2. Utilizando realimentación de acarreo (como en el sumador de complemento a uno del problema a.7) sumar el operando complementado y el otro (sin complementar).
   3. Si hubiese un acarreo de salida, el signo del resultado es el signo asociado al operando no complementado.
   4. En cualquier otro caso, si no hubiese acarreo de salida, complementar el resultado y darle el signo del operando complementado.

b. [10] <A.4> En nuestra discusión de la suma en punto flotante, sugerimos que cuando el resultado es negativo, el +1 se necesitaba para realizar el complemento a dos en la unidad de redondeo. Utilizar el resultado de la Parte A para obtener un sumador de punto flotante que no requiera esto.

A.16 [15] <A.7> Nuestro ejemplo que mostraba que el doble redondeo puede dar una respuesta de redondeo diferente una vez utilizada la regla de redondeo al par. Si los casos a mitad se redondean hacia arriba, ¿es todavía peligroso el doble redondeo?

A.17 [15/30] <A.9> El texto explicaba la división SRT en base 4 con dígitos de cociente −2, −1, 0, 1, 2. Suponer que 3 y −3 son también dígitos permitidos del cociente.

a. [15] ¿Qué relación sustituye a $|r_i| \leqslant 2b/3$?

b. [30] ¿Cuántos bits de b y P es necesario examinar?

A.18 [25] <A.6, A.9> La explicación de la instrucción de-paso del resto suponía que la división se hacía utilizando un algoritmo bit a bit. ¿Qué tendría que cambiar si la división se implementase utilizando un método de base superior?

A.19 [20/20/25/25/20] <A.3> Representación mediante logaritmos con signo.

a. [20] Suponer que se quiere representar un número x por su signo y $\log |x|$. Entonces si $\log |x|$ es no negativo, x debe ser $\geqslant 1$. Se pueden permitir menores x si se representa x por $\log k|x| < 1$ para una constante k. Utilice 0 si $k|x| < 1$. Ahora $\log k|x|$ no será un entero, pero se puede representar como un número de punto fijo. Si ponemos el punto binario m bits a la izquierda del bit menos significativo, escribir fórmulas para convertir x a la forma de logaritmo con signo.

b. [20] Dar las reglas de multiplicación y división.

c. [25] Mostrar que sin importar qué base de logaritmo se utilice, este sistema no puede representar exactamente 1, 2 y 3.

d. [25] Mostrar cómo implementar la suma utilizando una tabla que contiene 2^{p-1} entradas de $p - 1$ bits cada una, donde el número de logaritmo con signo se almacena en un registro de p bits.

e. [20] Mostrar que para los números que son exactamente representables en este sistema, la multiplicación es exacta, la suma no, pero $a(b + c) = ab + ac$ exactamente (cuando no hay desbordamiento/desbordamiento a cero).

A.20 [20/10] <A.8> Sumadores con salto de acarreo.

a. [20] Suponiendo que el tiempo es proporcional a los niveles lógicos, ¿qué tamaño de bloque (fijo) da la suma más rápida para un sumador de una longitud total fija?

b. [10] Explicar por qué el sumador con salto de acarreo emplea un tiempo $\sqrt{n}$.

A.21 [Discusión] En la aproximación MIPS para la manipulación de excepciones, se necesita un test para determinar si dos operandos en punto flotante pueden provocar una excepción. Esto debería ser rápido y no causar muchas falsas alarmas. ¿Se puede obtener un test práctico? El coste del rendimiento del diseño dependerá de la distribución de los números en punto flotante. Esto está explicado en Knuth [1981] y Swartzlander [1980].

A.22 [35] <A.8> El sumador más sencillo con selección de acarreo sustituye un sumador de n bits por sumadores de $n/2$ bits y un multiplexor. Un sumador de selección de acarreo, más complejo, podría utilizar sumadores de $n/4$ bits y más multiplexores. ¿Se puede diseñar un sumador que utilice multiplexores y sumadores de 1 bit y funcione en tiempo O(log n)? Tal sumador se denomina *sumador con suma condicional.*

A.23 [10/15/20/15/15] <A.6> Redondeo correcto en la división iterativa. Sean a y b números en punto flotante con p bits significativos ($p = 53$ en doble precisión). Sea q el cociente exacto $q = a/b$. Suponer que $\overline{q}$ es el resultado de un proceso de iteración, tal que $\overline{q}$ tiene algunos bits de precisión extra, y que $0 < q - \overline{q} < 2^{-p}$.

a. [10] Si x es un número en punto flotante, y $1 \leqslant x < 2$, ¿cuál es el siguiente número representable después de x?

b. [15] Mostrar cómo calcular q' a partir de $\overline{q}$, donde q' tiene $p + 1$ bits de precisión y $|q - q'| < 2^{-p}$.

c. [20] Suponiendo redondeo al más próximo, mostrar que el cociente redondeado correctamente es o bien q', $q' - 2^{-p}$ o $q' + 2^{-p}$.

d. [15] Dar reglas para calcular el cociente correctamente redondeado a partir de q', basándose en el bit de orden inferior de q' y en el signo de $a - bq'$.

e. [15] Resolver la Parte c para los otros tres modos de redondeo.

B Tablas completas de repertorios de instrucciones

B.1 Repertorio de instrucciones de usuario del VAX
B.2 Repertorio de instrucciones del sistema/360
B.3 Repertorio de instrucciones del 8086

B.1 Repertorio de instrucciones de usuario del VAX

Las tablas siguientes incluyen todas las instrucciones de usuario del VAX; las instrucciones de sistema no se incluyen.

El subrayado que sigue al nombre de la instrucción implica que la instrucción operará sobre cualquier tipo de datos contenido en los paréntesis que siguen a esa instrucción. Las abreviaturas de los tipos de datos son:

B = byte (8 bits)
W = palabra (16 bits)
L = palabra larga (32 bits)
Q = palabra cuádruple (64 bits)
O = palabra óctuple (128 bits)
F = flotante_F (32 bits)
D = flotante_D (64 bits)
G = flotante_G (64 bits)
H = flotante_H (128 bits)

Instrucciones aritméticas y lógicas enteras y de punto flotante

Instrucción	Descripción
ADAWI	Suma interbloqueada palabras alineadas
ADD_2	Suma (B,W,L,F,D,G,H) 2 operandos
ADD_3	Suma (B,W,L,F,D,G,H) 3 operandos
ADWC	Suma con acarreo
ASH_	Desplazamiento aritmético (L,Q)
BIC_2	Pone a cero bit (B,W,L) 2 operandos
BIC_3	Pone a cero bit (B,W,L) 3 operandos
BICPSW	Pone a cero bits de la palabra de estado del procesador
BIS_2	Pone a 1 bit (B,W,L) 2 operandos
BIS_3	Pone a 1 bit (B,W,L) 3 operandos
BISPSW	Pone a 1 bit de palabra de status del procesador
BIT_	Test de bit (B,W,L)
CLR_	Pone a cero (B,W,L=F,Q=D=G,O=H)
CVT_	Convierte (B,W,L,F,D,G,H)(B,W,L,F,D,G,H) excepto BB, WW, LL, FF, DD, GG, HH, DG y GD
CVTR_	Convierte redondeado (F,D,G,H) a palabra larga
CMP_	Compara (B,W,L,F,D,G,H)
DEC_	Decrementa (B,W,L)
DIV_2	Divide (B,W,L,F,D,G,H) 2 operandos

Instrucción	Descripción
DIV_3	Divide (B,W,L,F,D,G,H) 3 operandos
EDIV	División extendida
EMOD_	Módulo extendido (F,D,G,H)
EMUL	Multiplicación extendida
INC_	Incrementa (B,W,L)
INDEX	Calcula índice
MCOM_	Transfiere (B,W,L) complementado
MNEG_	Transfiere (B,W,L,F,D,G,H) negado
MOVA_	Transfiere dirección (B,W,L=F,Q=D=G,O=H)
MOV_*	Transfiere (B,W,L,F,D,G,H,Q,O)** —transferencia general entre dos operandos
MOVPSL	Transfiere desde palabra larga de estado del procesador
MOVZ_	Transfiere con extensión de cero (BW,BL,WL)
MUL_2	Multiplica (B,W,L,F,D,G,H) 2 operandos
MUL_3	Multiplica (B,W,L,F,D,G,H) 3 operandos
POLY_	Evaluación polinómica (F,D,G,H)
POPR	Saca registros de la pila
PUSHA_	Introduce dirección (B,W,L=F,Q=D=G,O=H) en pila
PUSHL	Introduce palabra larga en pila
PUSHR	Introduce registros en pila
ROTL	Desplazamiento circular de palabra larga
SBWC	Resta con acarreo
SUB_2	Resta (B,W,L,F,D,G,H) 2 operandos
SUB_3	Resta (B,W,L,F,D,G,H) 3 operandos
TST_	Test (B,W,L,F,D,G,H)
XOR_2	Or exclusiva (B,W,L) 2 operandos
XOR_3	Or exclusiva (B,W,L) 3 operandos

Instrucciones de salto, bifurcación y llamada a procedimientos

Instrucción	Descripción
ACB_	Suma, compara y salta (B,W,L,F,D,G,H)
AOBLEQ	Suma uno y bifurca menor o igual
AOBLSS	Suma uno y bifurca menor que
BB_	Salta sobre bit (a 1, a 0)
BBS_	Salta sobre bit (a 1, a 0) y pone (a 1, a 0) bit
BB_I	Salta si bit a 1 (a cero) y pone a 1 (a cero) el bit con interbloqueo
BCC	Salta acarreo a 0
BCS	Salta acarreo a 1
BEQL	Salta igual
BEQLU	Salta igual sin signo
BGEQ	Salta mayor o igual que
BGEQU	Salta mayor o igual que sin signo
BGTR	Salta mayor que
BGTRU	Salta mayor que sin signo
BLB_	Salta sobre bit inferior (puesto a 1, borrado a 0)
BLEQ	Salta menor o igual que
BLEQU	Salta menor o igual que sin signo
BLSS	Salta menor que
BLSSU	Salta menor que sin signo
BNEQ	Salta no igual
BNEQU	Salta no igual sin signo
BR_	Bifurca con desplazamiento (B, W)
BSB_	Salta a subrutina con desplazamiento (B, W)
BV_	Salta desbordamiento (a 1, a 0)
CALLG	Llama a procedimiento con lista general de argumentos
CALLS	Llama a procedimiento con lista en la pila de argumentos
CASE_	«Case» sobre (B, W, L)
JMP	Bifurca

Instrucción	Descripción
JSB	Bifurca a subrutina
RET	Retorno de procedimiento
RSB	Retorno de subrutina
SOBGEQ	Resta uno y salta mayor o igual que
SOBGTR	Resta uno y salta mayor que

Instrucciones decimales y de cadena

Instrucción	Descripción
ADDP4	Suma 4 operandos empaquetados
ADDP6	Suma 6 operandos empaquetados
ASHP	Desplazamiento aritmético empaquetado y redondeo
CMPC3	Compara caracteres 3 operandos
CMPC5	Compara caracteres 5 operandos
CMPP3	Compara 3 operandos empaquetados
CMPP4	Compara 4 operandos empaquetados
CRC	Calcula comprobación de redundancia cíclica
CVTLP	Convierte palabra larga a empaquetado
CVTPL	Convierte empaquetado a palabra larga
CVTPT	Convierte empaquetado a desempaquetado con signo en dígito menos significativo *(trailing)*
CVTTP	Convierte desempaquetado con signo en dígito menos significativo *(trailing)* a empaquetado
CVTPS	Convierte empaquetado a separado
CVTSP	Convierte separado a empaquetado
DIVP	Divide empaquetado
EDITPC	Edita empaquetado a cadena de caracteres
LOCC	Localiza carácter
MATCHC	Coincidencia de caracteres
MOVC3	Transfiere caracteres 3 operandos
MOVC5	Transfiere caracteres 5 operandos
MOVP	Transfiere empaquetado

Instrucción	Descripción
MOVTC	Transfiere caracteres traducidos
MOVTUC	Transfiere traducidos hasta carácter
MULP	Multiplicación empaquetada
SCANC	Explora caracteres
SKPC	Salta carácter
SPANC	Extiende carácter
SUBP4	Resta 4 operandos empaquetados
SUBP6	Resta 6 operandos empaquetados

Instrucciones de campo de bits de longitud variable

Instrucción	Descripción
CMPV	Compara campo
CMPZV	Compara campo extendido-cero
EXTV	Extrae campo
EXTZV	Extrae campo extendido-cero
INSV	Inserta campo
FFS	Encuentra primer 1
FFC	Encuentra primer 0

Instrucciones de cola

Instrucción	Descripción
INSQHI	Inserta entrada en cabeza de cola, interbloqueada
INSQTI	Inserta entrada en cola en la parte posterior, interbloqueada
INSQUE	Inserta entrada en cola
REMQMI	Elimina entrada en cabeza de cola, interbloqueada
REMQTI	Elimina entrada última de cola, interbloqueada
REMQUE	Elimina entrada de la cola

B.2 Repertorio de instrucciones del sistema/360

El repertorio de instrucciones del 360 se muestra en las tablas siguientes, organizadas por tipo y formato de instrucciones. El sistema 370 contiene 15 instrucciones de usuario adicionales.

Instrucciones enteras/lógicas y de punto flotante R-R

El * indica que la instrucción es de punto flotante, y puede ser bien D (doble precisión) o E (simple precisión).

Instrucción	Descripción
ALR	Suma lógica de registros
AR	Suma registros
A*R	Suma FP
CLR	Comparación lógica de registros
CR	Comparación de registros
C*R	Comparación FP
DR	División de registros
D*R	División FP
H*R	Mitad FP
LCR	Cargar registro complemento
LC*R	Cargar complemento
LNR	Cargar registro negativo
LN*R	Cargar negativo
LPR	Cargar registro positivo
LP*R	Cargar positivo
LR	Cargar registro
L*R	Cargar registro FP
LTR	Cargar y examinar registro
LT*R	Cargar y examinar registro FP
MR	Multiplicar registro
M*R	Multiplicar FP
NR	And de registros
OR	Or de registros

Instrucción	Descripción
SLR	Resta lógica de registros
SR	Restar registro
S*R	Resta FP
XR	Or exclusiva de registros

Saltos e instrucciones de inicialización de estado R-R

Estas son las instrucciones de formato R-R que bien saltan o ponen a 1 alguno estado del sistema; algunas de ellas son privilegiadas y legales sólo en modo supervisor.

Instrucción	Descripción
BALR	Salta y enlaza
BCTR	Salta sobre cuenta
BCR	Salta/condición
ISK	Inserta clave
SPM	Inicializa máscara de programa
SSK	Inicializa clave de almacenamiento
SVC	Llamada supervisor

Instrucciones de punto flotante y enteras/lógicas. Formato RX

Estas son las instrucciones de formato RX. El símbolo «+» significa una operación de una palabra (y equivale a nada) o H (que significa media palabra); por ejemplo, A+ significa los dos códigos de operación A y AH. El símbolo «*» es D o E significando punto flotante en doble o simple precisión.

Instrucción	Descripción
A+	Suma
A*	Suma FP
AL	Suma lógica
C+	Comparación

Instrucción	Descripción
C*	Comparación FP
CL	Comparación lógica
D	División
D*	División FP
L+	Carga
L*	Carga registro FP
M+	Multiplicación
M*	Multiplicación FP
N	And
O	Or
S+	Resta
S*	Resta FP
SL	Resta lógica
ST+	Almacena
X	Or exclusiva

Saltos y almacenamiento y cargas especiales. Formato RX

Instrucción	Descripción
BAL	Salta y enlaza
BC	Salta condición
BCT	Salta si cuenta
CVB	Convierte a binario
CVD	Convierte a decimal
EX	Ejecuta
IC	Inserta carácter
LA	Carga dirección
STC	Almacena carácter

Instrucciones de formato RS y SI

Estas son las instrucciones de formato RX y SI. El símbolo «*» puede ser A (aritmética) o L (lógica).

Instrucción	Descripción
BXH	Salta/mayor
BXLE	Salta/menor-igual
CLI	Comparación lógica inmediato
HIO	Alto E/S
LPSW	Carga PSW
LM	Carga múltiple
MVI	Transferencia inmediato
NI	And inmediato
OI	Or inmediato
RDD	Lectura directa
SIO	Comienza E/S
SL*	Desplaza a la izquierda A/L
SLD*	Desplaza a la izquierda doble A/L
SR*	Desplaza a la derecha A/L
SRD*	Desplaza a la derecha doble A/L
SSM	Inicializa máscara del sistema
STM	Almacenamiento múltiple
TCH	Test de canal
TIO	Test de E/S
TM	Test máscara
TS	Test e inicializa (Test and Set)
WRD	Escritura directa
XI	Or exclusiva inmediato

Instrucciones de formato SS

Estas son las instrucciones de cadena o decimales.

Instrucción	Descripción
AP	Suma empaquetada
CLC	Comparación lógica de caracteres
CP	Comparación empaquetada
DP	División empaquetada
ED	Editar
EDMK	Editar y marcar
MP	Multiplicación empaquetada
MVC	Transferir carácter
MVN	Transferencia numérico
MVO	Transferencia con desplazamiento
MVZ	Transferir zona
NC	And de caracteres
OC	Or de caracteres
PACK	Empaquetar (Carácter → decimal)
SP	Resta empaquetada
TR	Traducir
TRT	Traducir y test
UNPK	Desempaqueta
XC	Or exclusiva de caracteres
ZAP	Cero y suma empaquetada

B.3 Repertorio de instrucciones del 8086

Estas tablas contienen el repertorio de instrucciones del 8086; las instrucciones de punto flotante que no están incluidas ni son utilizadas por los benchmarks del 8086 no se incluyen.

Instrucciones aritméticas y lógicas

Instrucción	Descripción
AAA	Ajuste ASCII de la suma
AAD	Ajuste ASCII antes de la división
AAM	Ajuste ASCII después de la multiplicación
AAS	Ajuste ASCII después de la resta
ADC	Suma con acarreo
ADD	Suma entera
AND	And lógica
CBW/CWD/CDQ	Convierte byte a palabra/palabra a doble palabra/doble palabra a quad
CLC	Pone a cero el señalizador de acarreo
CLD	Pone el señalizador de dirección
CLI	Pone el señalizador de interrupción
CMC	Complementa el señalizador de acarreo
CMP	Comparación
DAA	Ajuste decimal después de la suma
DAS	Ajuste decimal después de la resta
DEC	Decrementar
DIV	División sin signo
IDIV	División con signo
IMUL	Multiplicación con signo
INC	Incrementar
MUL	Multiplicación sin signo
NEG	Negación
NOT	Not
OR	Or inclusiva
RCL	Desplazamiento a la izquierda a través del acarreo
RCR	Desplazamiento circular a la derecha a través del acarreo
ROL	Desplazamiento circular a la izquierda
ROR	Desplazamiento circular a la derecha
SAL/SHL	Desplazamiento aritmético a la izquierda

Instrucción	Descripción
SAR	Desplazamiento aritmético a la derecha
SBB	Resta con préstamo
SHR	Desplazamiento lógico a la derecha
STC	Pone a 1 del señalizador de arrastre
STD	Pone a 1 del señalizador de dirección
STI	Pone a 1 del señalizador de interrupción
SUB	Resta
TEST	Comparación lógica
XOR	Or exclusiva

Instrucciones de control

Instrucción	Descripción
CALL	Llamada a procedimiento (intrasegmento)
CALL	Llamada a procedimiento (intersegmento)
HLT	Alto
INT	Llamada a procedimiento de interrupción
INTO	Llamada a procedimiento de interrupción en caso de desbordamiento
IRET	Retorno de interrupción
JB/JNAE/JC	Bifurcación debajo
JBE/JNA	Bifurcación debajo o igual
JCXZ/JECXZ	Bifurcación CX/ECX ceros
JE/JZ	Bifurcación igual
JL/JNGE	Bifurcación menor
JLE/JNG	Bifurcación menor o igual
JMP	Bifurcación (intrasegmento)
JMPF	Bifurcación (intersegmento)
JNB/JAE/JNC	Bifurcación no debajo
JNBE/JA	Bifurcación no debajo o igual
JNE/JNZ	Bifurcación no igual

Instrucción	Descripción
JNL/JCE	Bifurcación no menor
JNLE/JG	Bifurcación no menor o igual
JNO	Bifurcación no desbordamiento
JNP/JPO	Bifurcación no paridad
JNS	Bifurcación no signo
JO	Bifurcación desbordamiento
JP/JPE	Bifurcación paridad
JS	Bifurcación signo
LOCK	Bloqueo de bus
RET	Retorno (intrasegmento)
RETF	Retorno (intersegmento)

Instrucciones de transferencia de datos

Instrucción	Descripción
IN	Entrada de un puerto
LAHF	Carga señalizadores en registro AH
LDS	Carga puntero en DS
LEA	Carga dirección efectiva
LES	Carga puntero en ES
LOCK	Bloqueo de bus
MOV	Transferencia
OUT	Salida a un puerto
POP	Sacar de pila
POPF/POPFD	Sacar de pila a señalizadores
PUSH	Introducir en pila
PUSH	Introducir registro segmento en pila
PUSHF/PUSHFD	Introducir señalizadores en pila
SAHF	Almacenar registro AH en señalizadores
XCHC	Intercambiar
XLAT/XLATB	Traducción mediante tabla

Instrucciones de cadena

Instrucción	Descripción
CMPS/CMPSB/CMPSW/CMPSD	Comparación de cadena
LODS/LODSB/LODSW/LODSD	Cargar cadena
MOVS/MOVSB/MOVSW/MOVSD	Transferir cadena
REP	Repetir
REPE/REPZ	Repetir mientras igual
REPNE/REPNZ	Repetir mientras no igual
SCAS/SACSB/SCASW/SACSD	Explorar cadena
STOS/STOSB/STOSW/STOSD	Almacenar cadena

C Medidas detalladas de repertorios de instrucciones

C.1 Medidas detalladas del VAX
C.2 Medidas detalladas del 360
C.3 Medidas detalladas del Intel 8086
C.4 Medidas detalladas del repertorio de instrucciones de DLX

C.1 Medidas detalladas del VAX

Instrucción	GCC	Spice	TeX	COBOLX	Promedio
Control	**30%**	**18%**	**30%**	**25%**	**26%**
Salto condicional	20%	13%	19%	18%	17%
BRB,BRW	6%	3%	4%	5%	5%
CALLS,CALLG	2%	1%	4%	0%	2%
RET	2%	1%	4%	0%	2%
JMP				2%	1%
Aritmética, lógica	**40%**	**23%**	**33%**	**24%**	**30%**
CMP*	12%	5%	11%	9%	9%
ADDL_	5%	12%	4%		5%
INCL	3%		3%	5%	3%
MOVA*	1%	3%	4%	2%	3%
TSTL	4%	2%	3%		2%
CLRL	3%	1%	2%	3%	2%
SUB*_	3%	1%	3%		2%
CVT*L	6%			0%	2%
ASHL	3%		3%	0%	2%
MULL_	0%			5%	1%
Transferencia de datos	**19%**	**15%**	**28%**	**4%**	**16%**
MOVL	15%	9%	17%	4%	11%
PUSHL	3%		7%		2%
MOVQ		6%			1%
MOVZ*L	1%		4%		1%
Punto flotante	**0%**	**23%**	**0%**	**0%**	**6%**
MULD_		9%			2%
SUBD_		6%			1%
ADDD_		6%			1%
DIVD_		3%			1%
CMPD		2%			
Decimal, cadena	**0%**	**0%**	**1%**	**38%**	**10%**
CVTTP,CVTPT				19%	5%
MOVC3,MOVC5			1%	9%	2%

Instrucción	GCC	Spice	TeX	COBOLX	Promedio
ADDP4				6%	1%
CMPP_				2%	1%
CMPC3				2%	1%
Totales	**88%**	**79%**	**92%**	**88%**	**87%**

FIGURA C.1 Instrucciones responsables de más de 1,5 por 100 de las ejecuciones dinámicas en cualquier benchmark. Las instrucciones se descomponen en cinco clases, impresas en negrita. Los datos de esas filas dan la frecuencia total para las operaciones de esa clase. Las celdas que representan una contribución de 1 por 100 o menos están vacías, excepto la columna de promedio que puede tener una entrada de 1 por 100. A causa del redondeo, la media puede diferir de la media que parecería correcta si se basara en las figuras de las columnas individuales.

C.2 Medidas detalladas del 360

Instrucción	PLIC	FORTGO	PLIGO	COBOLGO	Promedio
Control	**32%**	**13%**	**5%**	**16%**	**16%**
BC,BCR	28%	13%	5%	14%	15%
BAL,BALR	3%			2%	1%
Aritmética, lógica	**29%**	**35%**	**29%**	**9%**	**26%**
A,AR	3%	17%	21%		10%
SR	3%	7%			3%
SLL		6%	3%		2%
LA	8%	1%	1%		2%
CLI	7%				2%
NI				7%	2%
C	5%	4%	4%	0%	3%
TM	3%	1%		3%	2%
MH			2%		1%
Transferencia de datos	**17%**	**40%**	**56%**	**20%**	**33%**
L,LR	7%	23%	28%	19%	19%
MVI	2%		16%	1%	5%
ST	3%		7%		3%
LD		7%	2%		2%

Instrucción	PLIC	FORTGO	PLIGO	COBOLGO	Promedio
Control	**32%**	**13%**	**5%**	**16%**	**16%**
STD		7%	2%		2%
LPDR		3%			1%
LH	3%				1%
IC	2%				1%
LTR		1%			0%
Punto flotante		**7%**			**2%**
AD		3%			1%
MDR		3%			1%
Decimal, cadena	**4%**			**40%**	**11%**
MVC	4%			7%	3%
AP				11%	3%
ZAP				9%	2%
CVD				5%	1%
MP				3%	1%
CLC				3%	1%
CP				2%	1%
ED				1%	0%
Total	**82%**	**95%**	**90%**	**85%**	**88%**

FIGURA C.2 (Ver página anterior.) Distribución de las frecuencias de ejecución de instrucciones para los cuatro programas del 360. Se incluyen todas las instrucciones con una frecuencia de ejecución mayor del 1,5 por 100. Las instrucciones inmediatas, que operan sobre sólo un único byte, se incluyen en la sección que caracteriza su operación, en lugar de con las versiones de largas cadenas de caracteres de la misma operación. A efectos de comparación, las frecuencias medias para las principales clases de instrucciones del VAX son 23 por 100 (control), 28 por 100 (aritmética), 29 por 100 (transferencia de datos), 7 por 100 (punto flotante) y 9 por 100 (decimal). De nuevo, se puede presentar una entrada del 1 por 100, en la columna de promedio, debido a las entradas de las columnas constituyentes.

C.3 Medidas detalladas del Intel 8086

Instrucción	Turbo C	MASM	Lotus	Promedio
Control	**21%**	**20%**	**32%**	**24%**
Bifurcaciones condicionales	10%	12%	9%	10%
CALL,CALLF	4%	3%	5%	4%
RET,RETF	4%	3%	5%	4%
LOOP			12%	4%
JMP	3%	2%	2%	2%
Aritmética, lógica	**23%**	**24%**	**26%**	**25%**
CMP	8%	9%	5%	7%
SAL,SHR,RCR	2%	1%	11%	5%
ADD	3%	2%	3%	3%
OR,XOR	4%	2%	2%	3%
INC,DEC	3%	4%	3%	3%
SUB	2%	3%		2%
CBW	1%	1%		1%
TEST		2%	2%	1%
Transferencia de datos	**49%**	**46%**	**30%**	**42%**
MOV	29%	31%	21%	27%
LES	6%	2%		3%
PUSH	10%	8%	4%	7%
POP	5%	6%	5%	5%
Totales	**93%**	**90%**	**88%**	**90%**

FIGURA C.3 Instrucciones responsables de más del 1,5 por 100 de las ejecuciones sobre cualquiera de los tres benchmarks. Algunas instrucciones muy similares se combinaron por simplicidad. Aunque MASM haga algún uso de operaciones de cadena, la frecuencia es demasiado baja para incluirla en la tabla.

FIGURA C.4 (Ver página siguiente.) Mezclas de instrucciones para GCC, Spice, TeX, y el benchmark de COBOL U. S. Steel. Algunas instrucciones se combinaron en interés del espacio y porque la clase combinada refleja más correctamente lo que está haciendo el procesador. La clase de instrucciones «B--Z» incluye todos los saltos condicionales (que son todas las comparaciones con cero). La clase «S--,S--I» incluye todas las instrucciones condicionales de inicialización (set), tanto inmediatas como de registro-registro. Las operacio-

C.4 Medidas detalladas del repertorio de instrucciones de DLX

Instrucción	GCC	Spice	TeX	US Steel	Promedio
Control	**20%**	**5%**	**7%**	**23%**	**14%**
B--Z	19%	2%	7%	16%	11%
J	2%	3%		3%	2%
JAL				2%	0%
JR				2%	0%
Aritmética, lógica	**36%**	**28%**	**41%**	**49%**	**39%**
ADDU,ADDUI	17%	16%	20%	27%	20%
LHI	2%	7%	10%	3%	5%
SLL	5%	5%	5%	4%	5%
LI	4%		4%	6%	4%
S--,S--I	5%		3%	3%	3%
AND,ANDI	2%			3%	1%
SRA	2%			2%	1%
OR,ORI				2%	1%
Transferencia de datos	**28%**	**35%**	**33%**	**10%**	**26%**
LW	18%	8%	19%	5%	13%
SW	10%	2%	12%	5%	7%
LBU			2%		1%
LD		14%			4%
SD		6%			1%
MOVFP2I, MOVI2FP		5%			1%
Punto flotante	**0%**	**15%**	**0%**	**0%**	**4%**
FMUL		5%			1%
FADD		4%			1%
FSUB		3%			1%
FDIV		3%			1%
Totales	**85%**	**83%**	**82%**	**82%**	**83%**

nes inmediatas se han combinado con la clase no inmediata para todas las operaciones excepto las de carga, donde son diferentes. De nuevo, un espacio blanco significa que la instrucción no es responsable de más del 1,5 por 100 de las ejecuciones, y el promedio puede ser un 1 por 100 o menos debido a que la instrucción no la utilizan todos los benchmarks.

D Medidas de tiempo frente a frecuencia

D.1 Distribución de tiempos en el VAX-11/780

D.2 Distribución de tiempos en el IBM 370/168

D.3 Distribución de tiempos en un 8086 de un IBM PC

D.4 Distribución de tiempos en un procesador próximo a DLX

D.1 Distribución de tiempos en el VAX-11/780

Sabemos de los Capítulos 2 y 3 que medir únicamente el recuento de instrucciones puede ser erróneo. En este apéndice examinaremos las distribuciones de tiempo para algunos programas, ejecutándose en estas cuatro máquinas. Para el 360, el 8086 y DLX, mostraremos la distribución de tiempo promediada sobre los tres programas en el formato de gráficos utilizado anteriormente. Para el VAX, utilizaremos las medidas descritas en Clark y Levy [1982] (ver Referencias del Capítulo 4).

La Figura D.1 muestra la distribución de ejecuciones de instrucciones, por tiempo y frecuencia de ocurrencia. Estos datos fueron medidos por Emer y presentados por Clark y Levy para un VAX-11/780 bajo VMS con múltiples usuarios haciendo tres tareas principales:

1. Actualizar ficheros indexados.
2. Ejecutar una rutina de multiplicación matricial.
3. Realizar el desarrollo de un programa, incluyendo edición, compilación y depuración.

La Figura D.1 incluye cualquier instrucción de usuario que represente más del 1 por 100 de las ejecuciones de las instrucciones o más del 1 por 100 del tiempo de ejecución. Hay 26 instrucciones que se ajustan a esta descripción, y juntas contabilizan el 59 por 100 de las ejecuciones y el 58 por 100 del tiempo. Los datos medidos incluyen los gastos extras del sistema operativo y del sistema de ficheros.

Las distribuciones de tiempos son particularmente importantes en arquitecturas como el VAX, donde el número de ciclos por instrucción puede variar de uno o dos hasta decenas o cientos.

D.2 Distribución de tiempos en el IBM 370/168

La Figura D.2 muestra la distribución de tiempos en un IBM 370/168 para los mismos programas que explicamos en el Capítulo 4 e incluimos en la Figura 4.28. Se incluyen todas las instrucciones que son responsables de más del 1,5 por 100 de la frecuencia y del tiempo de ejecución, para al menos un programa. En la distribución de tiempos aparecen algunas instrucciones que no estaban en la distribución de frecuencias, donde su ocurrencia es muy baja. Estas instrucciones, que no están en la Figura 4.28, son

TRT. Traduce y test, una instrucción de cadenas utilizada por el compilador de PL/I, muy probablemente para explorar la fuente de entrada; emplea el 5,4 por 100 del tiempo en ese programa.

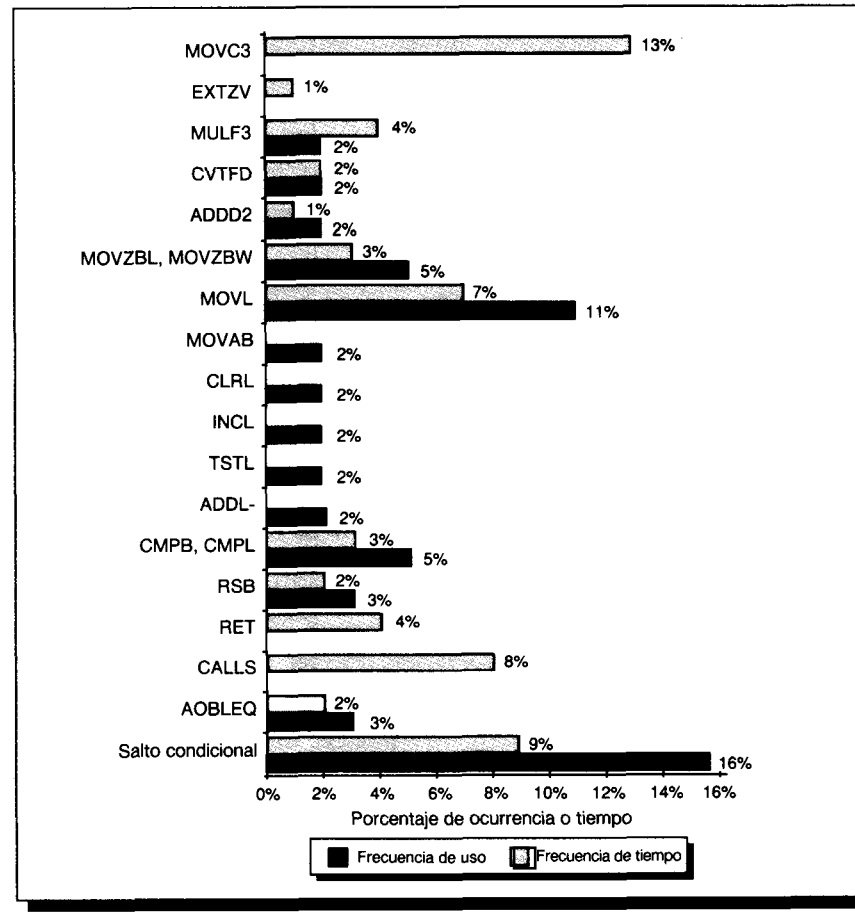


FIGURA D.1 Distribución de tiempo y frecuencia para una carga de trabajo multiusuario en un VAX-11/780 bajo VMS. Este dato incluye todas las instrucciones de usuario que son responsables de más del 1 por 100 de o bien las ejecuciones de instrucciones o del tiempo de ejecución. (Dos instrucciones del sistema operativo (REI y MTPR), cada una de las cuales contabiliza aproximadamente el 1 por 100 del tiempo de ejecución, no están incluidas.) La ausencia de una barra de frecuencia de ejecución o una barra de frecuencia-tiempo para una entrada (tal como MOVC3 o TSTL) significa que la frecuencia de tiempo o la frecuencia del tiempo de ejecución está por debajo del 1 por 100 (¡no que sea 0!). Clark y Levy [1982] comentaron que el gran porcentaje de tiempo consumido por MOVC3 en la distribución de tiempos es algo anormal para una carga de trabajo no comercial y no ha sido observada en otras medidas sobre el 11/780.

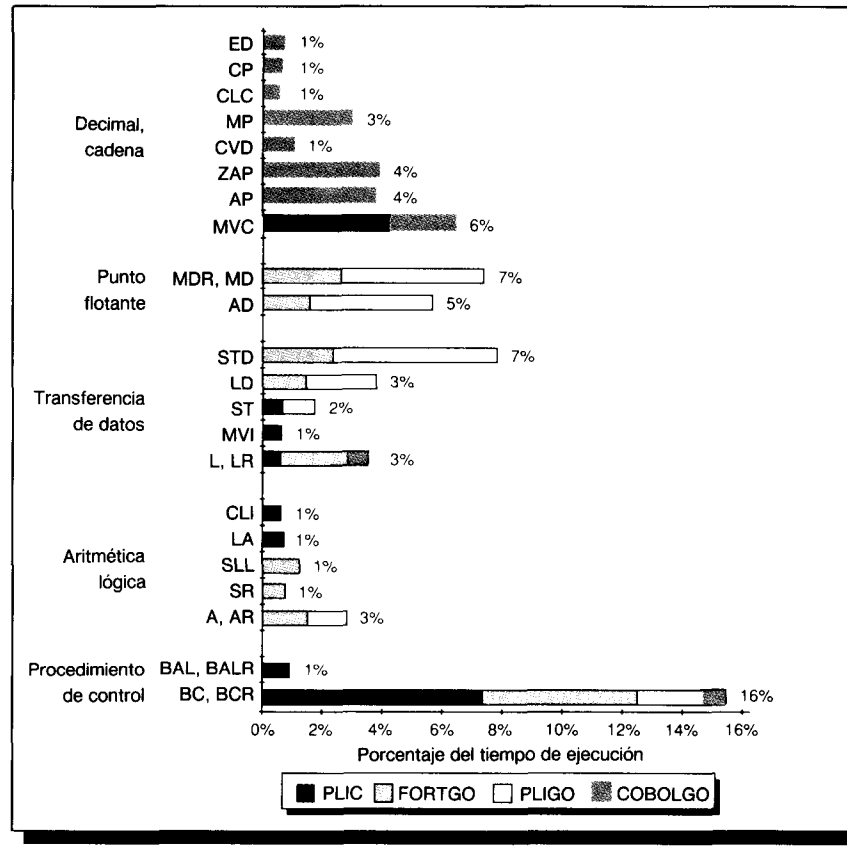


FIGURA D.2 Distribución de tiempos para los cuatro programas explicados en el Capítulo 4 ejecutados en un IBM 370/168. Los datos correspondientes sobre frecuencia de ejecución aparecen en la Figura 4.28 (pág. 188) o en forma de tabla en la Figura C.2. Cualquier instrucción con una frecuencia mayor del 1,5 por 100 en la distribución de tiempos y en la distribución de número de ejecuciones se incluye en este diagrama. Shustek [1978] (ver Referencias del Capítulo 4) calculó estos números utilizando un modelo de la CPU 370/168. El modelo predice el tiempo de ejecución para los programas y tiene una precisión global para cada programa de aproximadamente el 99 por 100, excepto en PLIGO, donde tiene un error del 8 por 100.

DP. División empaquetada, una instrucción de baja frecuencia, pero de larga ejecución, que emplea 18,7 por 100 del tiempo en COBOLGO.

DDR. División de dobles registros, una división de punto flotante, infrecuente pero de larga ejecución; es el 5,2 por 100 del tiempo de ejecución FORTGO.

LM y STM. Carga múltiple y almacenamiento múltiple, con frecuencias exactamente por debajo del 1 por 100, son algo más lentas que la instrucción media; por tanto, emplea del 3 al 4 por 100 de los ciclos en PLIGO.

BCT,BXLE. Saltos de bucles que involucran incrementar contadores o hacer otras comparaciones; BCT consume aproximadamente el 2 por 100 del tiempo en PLIC, y BXLE el 3,5 por 100 en FORTGO.

Algunas de las instrucciones de la ALU y de transferencia de datos más simples, pero de baja frecuencia que aparecían en la distribución de frecuencia no aparecen en la distribución de tiempos porque constituyen un porcentaje muy pequeño del tiempo de ejecución. En total, las instrucciones mostradas en la Figura D.2 contabilizan el 89 por 100 de las ejecuciones de instrucciones y el 72 por 100 del tiempo de ejecución.

La Figura D.3 da el tiempo medio de ejecución dividido por la frecuencia media para las instrucciones que aparecen en ambas distribuciones. Esta medida es una relación que indica el coste relativo de una instrucción. Por ejemplo, una instrucción que es responsable del 10 por 100 de las ejecuciones y del 10 por 100 del tiempo de ejecución tendrá una relación de 1:1, o un factor de coste de 1, y un CPI igual al CPI medio en la máquina.

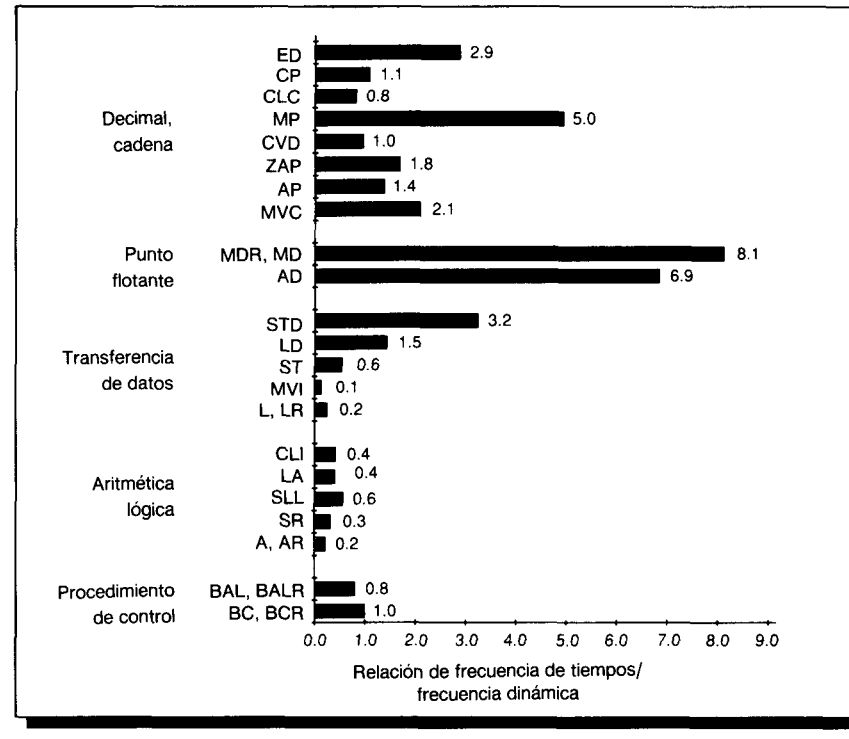


FIGURA D.3 Frecuencia de tiempo (porcentaje de ciclos ejecutando esta instrucción en un IBM 370/168) dividida por frecuencia dinámica (porcentaje de ejecuciones de esta instrucción). Los programas son los del Capítulo 4. Este dato se obtiene directamente de las Figuras 4.28 y D.2. Esto muestra claramente que las instrucciones de punto flotante son las más caras.

D.3 Distribución de tiempos en un 8086 de un IBM PC

La Figura D.4 continúa nuestro examen de la distribución de tiempos examinando las instrucciones que consumen más tiempo en el 8086 para los mismos programas que se midieron en el Capítulo 4. Estas curvas parecen muy similares a las de la Figura 4.32, la distribución de frecuencias para el 8086 (mostrado en forma de tabla en la Figura C.3). Dos instrucciones aritméticas y lógicas, `CBW` y `SUB`, que aparecían en la distribución de frecuencias, no aparecen en la parte superior de la distribución de tiempos de ejecución. Adicionalmente, hay cuatro instrucciones que tienen una contribución significativa a la frecuencia de tiempos, pero no están en la distribución de ejecución-frecuencia:

- Las instrucciones de cadena `SCAS` (una búsqueda de cadena) y `MOVS` (una transferencia de cadena). Ambas instrucciones se utilizan en MASM, donde contabilizan el 8 y 7 por 100 del tiempo de ejecución, respectivamente. `MOVS` también es utilizada en Lotus, donde contabiliza el 6,6 por 100 del tiempo de ejecución del programa.

- La multiplicación y división entera `ML16` y `DV16`. Estas se utilizan en Lotus, donde respectivamente contabilizan el 10 por 100 y el 4 por 100 del tiempo de ejecución del programa.

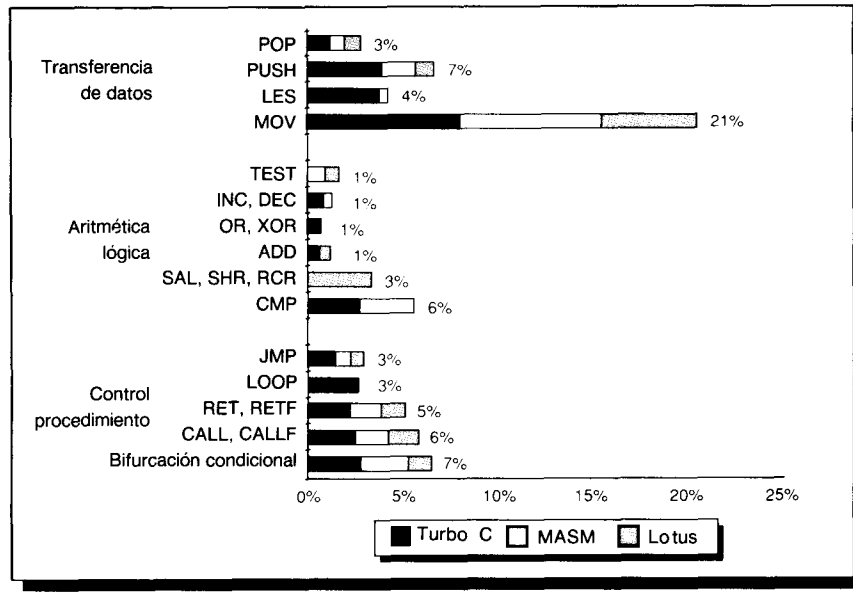


FIGURA D.4 Distribución de tiempo del 8086 en un IBM PC bajo MS-DOS. El formato y datos son iguales que los de la Figura 4.32.

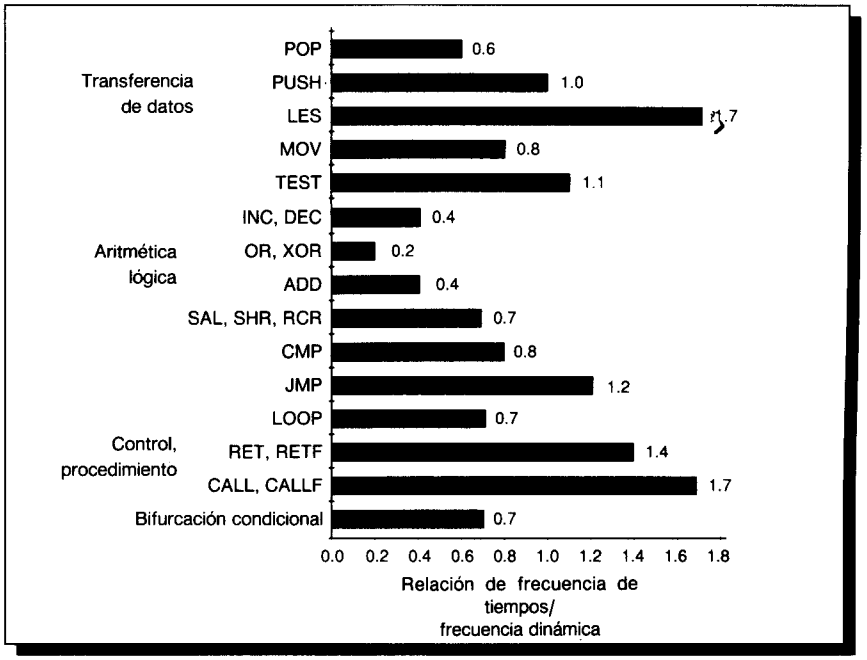


FIGURA D.5 Distribución de tiempo dividida por la distribución de frecuencia para el 8086. Este dato se obtiene directamente de las Figuras 4.28 y D.4. La distribución es remarcablemente más plana que la del IBM 360 o el VAX.

Juntas, las instrucciones de la Figura D.4 son responsables del 87 por 100 de las ejecuciones de las instrucciones y del 85 por 100 del tiempo de ejecución.

La Figura D.5 muestra la relación entre tiempos de ejecución y frecuencias de ejecución en la misma forma utilizada para el IBM 360. Llamadas, retornos y cargar un registro segmento consumen un gran porcentaje del tiempo de ejecución relativo a su ocurrencia dinámica. Sin embargo, el perfil global del tiempo de ejecución del 8086 está mucho más próximo al perfil de la frecuencia de ejecución —la correspondencia es, a menudo, 1:1, y nunca, tan alta como, 1:2. Esto se debe principalmente a que la variación del CPI entre las instrucciones es pequeña comparada con el CPI medio global de 14,1. Las instrucciones de larga ejecución que no aparecen en las frecuencias de ejecución pero son consumidoras importantes del tiempo de ejecución (y tendrían un CPI elevado) son las instrucciones de cadena y la multiplicación y la división enteras.

D.4 Distribución de tiempos en un procesador próximo a DLX

Para obtener una distribución de tiempos para DLX, volvemos a la DECstation 3100, que tiene una arquitectura a nivel lenguaje máquina muy

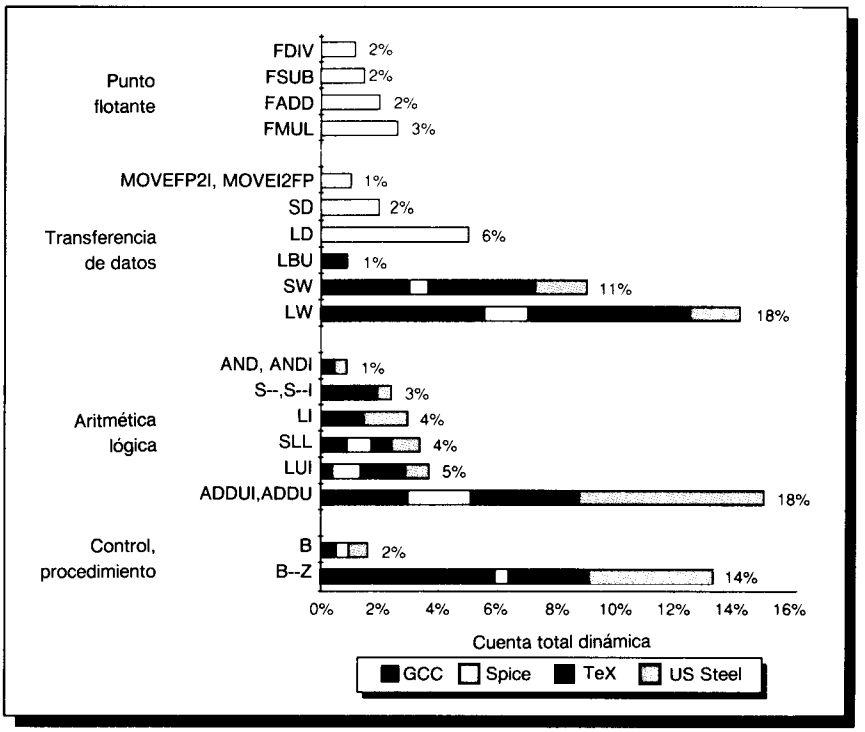


FIGURA D.6 Distribución de tiempos para nuestros tres benchmarks más el benchmark COBOL U. S. Steel cuando se ejecutan en DLX utilizando las medidas de CPI de una DECstation 3100.

similar a DLX (ver Apéndice E). La distribución de tiempo en la DECstation 3100 para los mismos programas medidos en el Capítulo 4 (Figura 4.34) y en forma de tabla en la Figura C.4 se muestra en la Figura D.6. La Figura D.6 incluye todas las instrucciones que contribuyen más de un 1 por 100 al tiempo de ejecución. En total, estas instrucciones contabilizan el 81 por 100 de todas las ejecuciones de las instrucciones y el 97 por 100 del tiempo de ejecución.

Esta distribución de tiempos es la más próxima a la distribución de frecuencias. Esto es porque, bajo condiciones ideales, casi todas las instrucciones en DLX necesitan un ciclo; sólo las instrucciones LD y SD deben emplear dos ciclos. Por supuesto, estas condiciones perfectas nunca se presentan. El CPI medio, utilizando la DECstation 3100 como base, es aproximadamente 1,6 para GCC. TeX y COBOLX, y aproximadamente 2,1 para Spice.

La Figura D.7 muestra la contribución al tiempo de ejecución dividida por la contribución a la frecuencia de ejecución para las instrucciones con valores más altos. Como en los diagramas del 360 y 8086, un valor por encima de 1 indica que esta instrucción tiene un CPI más alto que la instrucción prome-

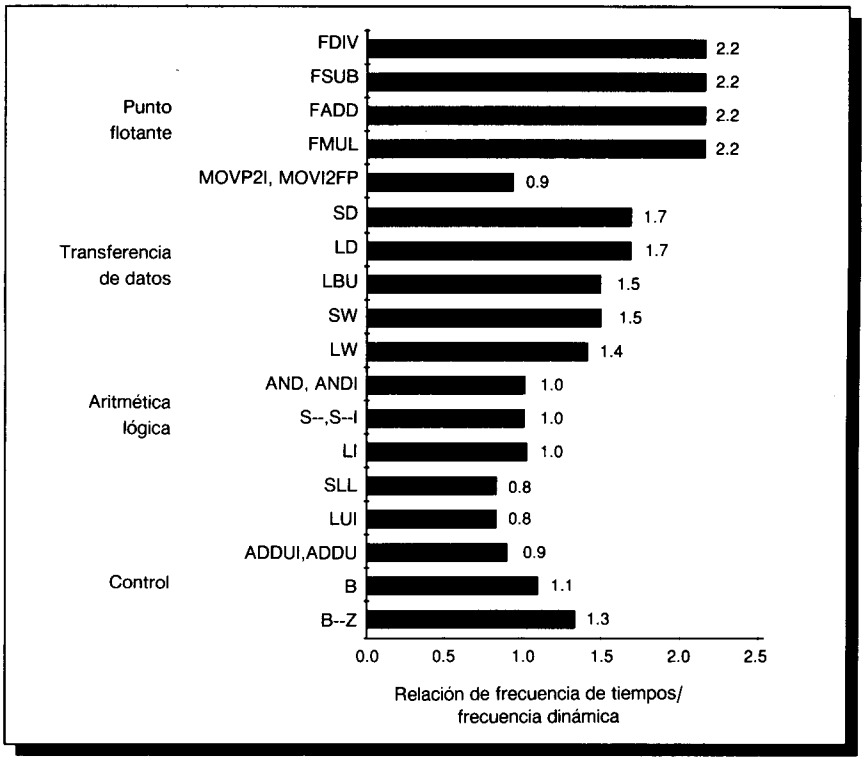


FIGURA D.7 Frecuencia de tiempo dividida por frecuencia de ejecución para DLX utilizando los datos de tiempo de la Figura D.6 y de frecuencia de la Figura 4.34. Las transferencias de registro entero-registro punto flotante son baratas, ya que son realmente operaciones registro-registro. Sorprendentemente, las referencias a memoria de doble precisión **no** son el doble de caras que las cargas y almacenamientos de 32 bits. ¿Puede usted hipotetizar el porqué basándose en las discusiones sobre segmentación y diseño de cache?

dio. No obstante, recordar que la relación no indica el CPI para la instrucción. Sin embargo, podemos utilizar este número para determinar el CPI de una instrucción, dado el CPI base para un programa específico.

RISC: cualquier computador anunciado después de 1985.

Steven Przybylski (un diseñador del MIPS de Stanford)

E.1 Introducción

E.2 Modos de direccionamiento y formatos de instrucción

E.3 Instrucciones: el subconjunto de DLX

E.4 Instrucciones: extensiones comunes a DLX

E.5 Instrucciones únicas del MIPS

E.6 Instrucciones únicas del SPARC

E.7 Instrucciones únicas del M88000

E.8 Instrucciones únicas del i860

E.9 Observaciones finales

E.10 Referencias

E Visión general de las arquitecturas RISC

E.1 Introducción

En este Apéndice analizamos cuatro ejemplos de arquitecturas de Computadores de Repertorio Reducido de Instrucciones (RISC):

- Intel 860;
- MIPS R3000/3010 (más una sección sobre MIPS II, utilizado en el R6000);
- Motorola M88000; y
- SPARC, desarrollado originalmente por Sun Microsystems.

También incluimos DLX, la arquitectura a nivel lenguaje máquina inventada para este libro. (Una revisión de DLX puede encontrarse en la cubierta posterior o en las páginas 172-180 del Capítulo 4.) Las características de estas arquitecturas se encuentran en la Figura E.1.

Nunca ha habido otra clase de computadores que fuera tan similar. Esta similitud permite la presentación de cuatro arquitecturas a la vez, ¡incluyendo DLX! Después de presentar los modos de direccionamiento y los formatos de instrucción, las instrucciones se presentan en tres pasos:

- Instrucciones de DLX;
- Instrucciones que no aparecen en DLX pero que se encuentran en dos o más arquitecturas; y
- Las instrucciones únicas y características de cada arquitectura.

Concluimos con una especulación sobre las direcciones futuras para los RISC.

	DLX	i860	MIPS	M88000	SPARC
Fecha anunciada	1990	1989	1986	1988	1987
Tamaño de instrucciones (bits)	32	32	32	32	32
Espacio de direcciones (tamaño, modelo)	32 bits, plano	32 bits, plano	32 bits, plano	32 bits, plano	32 bits identificador
Alineación de datos	Alineado	Alineado	Alineado	Alineado	Alineado
Modos de direccionamiento de datos	1	2	1	3	2
Protección	Página	Página	Página	Página	Página
Tamaño de página	4 KB	4 KB	4 KB	4 KB	4-64 KB
E/S	Mapeada en memoria	Mapeada en memoria	Mapeada en memoria	Mapeada en memoria	Mapeada en memoria
Registros enteros (tamaño, modelo, número)	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits	31 GPR x 32 bits
Registros separados de punto flotante	32 x 32 o 16 x 24 bits	30 x 32 o 15 x 64 bits	16 x 32 o 16 x 64	0	32 x 32 o 16 x 64
Formato de punto flotante	IEEE 754 simple, doble	IEEE 754 simple, doble	IEEE 754 simple, doble	IEEE 754 simple, doble	IEEE 754 simple, doble

FIGURA E.1 Resumen de cinco arquitecturas recientes. Exceptuando el número de modos de direccionamiento de datos y algunos detalles del repertorio de instrucciones, los repertorios de instrucciones enteras de estas arquitecturas de finales de los años ochenta son idénticas. Contrastar esto con la Figura E.13.

E.2 Modos de direccionamiento y formatos de instrucción

La Figura E.2 muestra los modos de direccionamiento de los datos soportados por cada arquitectura. Como todas tienen un registro que siempre tiene el valor 0 —en efecto, es `r0` en cada arquitectura— el modo de direccionamiento absoluto con rango limitado se puede sintetizar utilizando `r0` como base en el

Modo de direccionamiento	DLX	i860	MIPS	M88000	SPARC
Registro + desplazamiento (desplazamiento o registro base)	√	√	√	√	√
Registro + registro (indexado)	—	√	—	√	√
Registro + registro escalado (escalado)	—	—	—	√	—

FIGURA E.2 Resumen de modos de direccionamiento de los datos. (Estos modos de direccionamiento están explicados en la Sección 3.4.) Aunque el i860 tiene direccionamiento indexado de datos para todas las cargas y almacenamiento de punto flotante, no está disponible para almacenamientos de enteros.

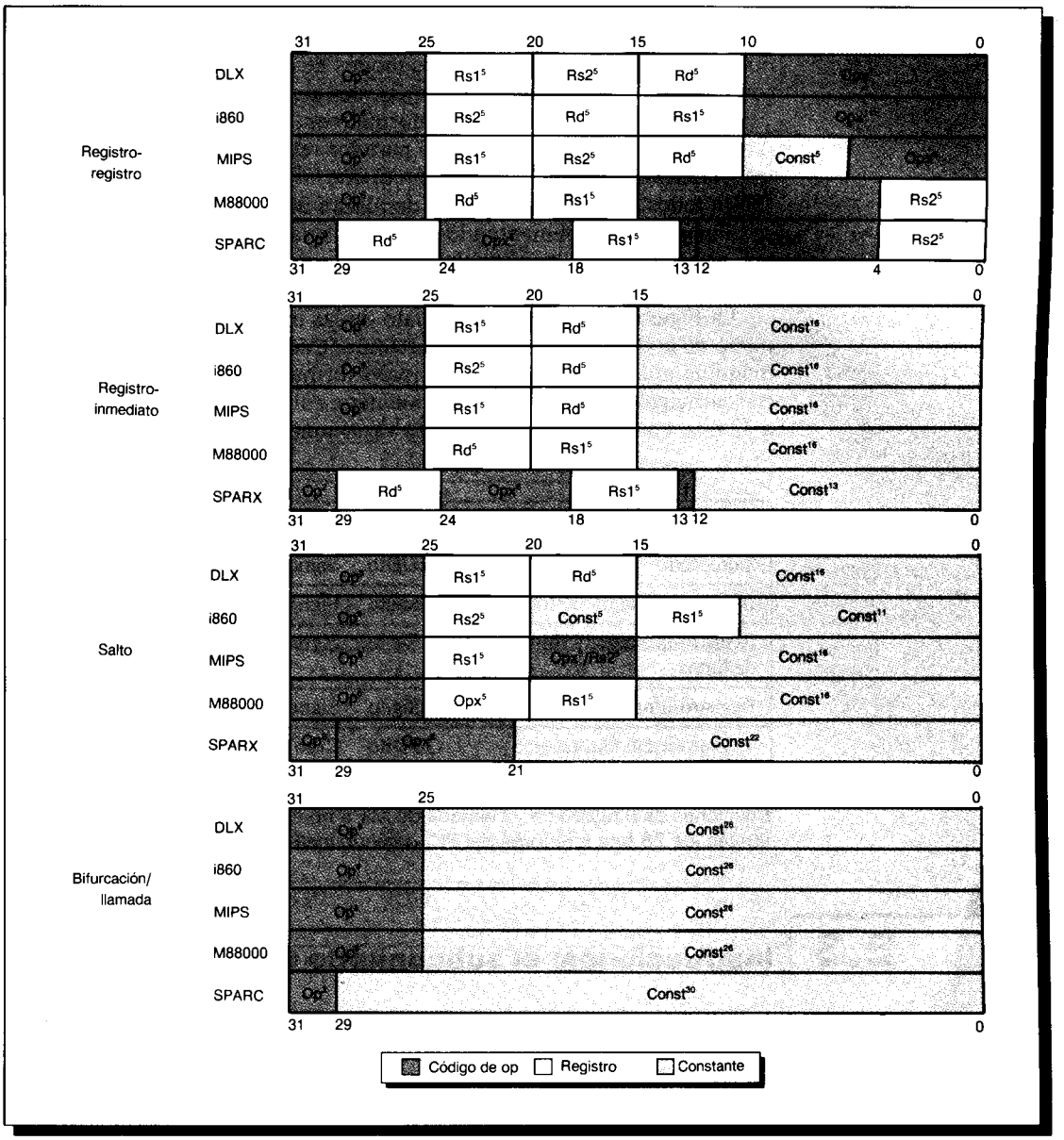


FIGURA E.3 Formatos de instrucción para cinco arquitecturas. Estos cuatro formatos se encuentran en las cinco arquitecturas. (La notación superíndice de esta figura es algo diferente de nuestra notación estándar; muestra la anchura de un campo de bits.) Aunque los campos de los registros se localicen en partes similares de la instrucción, obsérvese que se mezclan el destino y dos campos fuente. Aquí están los significados de las abreviaturas: Op = código de operación principal, Opx = una extensión del código de operación, Rd = registro destino, Rs1 = registro fuente 1, Rs2 = registro fuente 2 y Const = una constante. La diferencia principal del M88000 es el formato de registro inmediato cuando la operación no necesita un inmediato completo de 16 bits: un campo de extensión del código de operación se coloca en los bits superiores del campo constante. La diferencia del i860 es utilizar Rs1 en el formato de Salto para especificar una constante de 5 bits, así como un registro.

direccionamiento de desplazamiento. Análogamente, el direccionamiento indirecto de registros se sintetiza utilizando direccionamiento de desplazamiento con desplazamiento 0. Los modos de direccionamiento simplificados son una característica distintiva entre estas arquitecturas y las anteriores.

Las referencias al código son, normalmente, relativas al PC, aunque el indirecto de registros está soportado para los retornos de procedimiento y para las sentencias «case». Una variación es que las direcciones de salto relativas al PC en todos, excepto en DLX, se desplazan 2 bits a la izquierda antes de que se sumen al PC, incrementando así la distancia de salto. Esto funciona porque la longitud de todas las instrucciones es de una palabra y las instrucciones deben ser palabras alineadas de memoria.

La Figura E.3 muestra el formato de las instrucciones, que incluye el tamaño de la dirección en las instrucciones. Cada arquitectura a nivel lenguaje máquina utiliza estos cuatro principales formatos de instrucción. Las diferencias principales son sutiles, concernientes a cómo extender campos constantes a 32 bits. La Figura E.4 muestra las variaciones.

Formato: categoría de instrucción	DLX	i860	MIPS	M88000	SPARC
Salto: todo	Signo	Signo	Signo	Signo	Signo
Bifurcación/llamada: todo	Signo	Signo	—	Signo	Signo
Registro-inmediato: transferencia de datos	Signo	Signo	Signo	Cero	Signo
Registro-inmediato: aritmético	Signo	Signo	Signo	Cero	Signo
Registro-inmediato: lógico	Signo	Cero	Cero	Cero	Signo

FIGURA E.4 Resumen de extensión de constantes. La constante en las instrucciones de Bifurcación y Llamada de MIPS no extiende el signo, ya que sólo sustituyen los 28 bits inferiores del PC, dejando inalterables los 4 bits superiores.

E.3 Instrucciones: el subconjunto de DLX

Las analogías de cada arquitectura permiten descripciones simultáneas de las arquitecturas, comenzando con las operaciones equivalentes a DLX.

Instrucciones de DLX

Casi cualquier instrucción que aparece en el repertorio de DLX se encuentra en las otras arquitecturas, como muestra la Figura E.5. (Como referencia, las definiciones de las instrucciones DLX se encuentran en las páginas 172-180 del Capítulo 4 y en las páginas xi y xii.) Las instrucciones se listan bajo cuatro categorías: «Transferencia de datos», «Aritmética, lógica», «Control» y «Punto flotante». Una quinta categoría en la figura muestra los convenios para uso de registros y pseudoinstrucciones en cada arquitectura. Si una instrucción

Nombre de instrucción	DLX	i860	MIPS	M88000	SPARC
Transferencia de datos (formatos de instrucción)	**R-I**	**R-I, R-R**	**R-I**	**R-I, R-R**	**R-I, R-R**
Carga byte con signo	LB	LD.B	LB	LD.B	LDSB
Carga byte sin signo	LBU	LD.B; AND...,x00FF,...	LBU	LD.BU	LDUB
Carga media palabra con signo	LH	LD.S	LH	LD.H	LDSH
Carga media palabra sin signo	LHU	LD.S; AND...,xFFFF...	LHU	LD.HU	LDUH
Carga palabra	LW	LD.L	LW	LD	LD
Carga flotante SP	LF	FLD.L	LWC1	LD	LDF
Carga flotante DP (ver E.5 para MIPS)	LD	FLD.D	LWC1 Rd; LWC1 Rd+1	LD.D	LDDF
Almacena byte	SB	ST.B	SB	ST.B	STB
Almacena media palabra	SH	ST.S	SH	ST.H	STH
Almacena palabra	SW	ST.L	SW	ST	ST
Almacena flotante SP	SF	FST.L	SWC1	ST	STF
Almacena flotante DP (ver E.5 para MIPS)	SD	FST.D	SWC1 Rd; SWC1 Rd+1	ST.D	STDF
Lee, escribe registros especiales	MOVS2I, MOVI2S	LD.C, ST.C	MF_, MT_	LDCR, FLDCR STCR, FSTCR	RD, LDFSR WR, STFSR
Transfiere entero a reg. FP	MOVI2FP	IXFR	MFC1	no aplicable	ST;LDF
Transfiere FP a reg. entero	MOVFP2I	FXFR	MTC1	no aplicable	STF;LD
Aritmética, lógica (formatos de instrucciones)	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**
Suma	ADDU, ADDUI	ADD, ADDU	ADDU, ADDIU	ADDU	ADD
Suma (trap si desbordamiento)	ADD, ADDI	ADD; INTOVR	ADD, ADDI	ADD	ADDcc; TVS
Resta	SUBU,SUBUI	SUB,SUBU	SUBU	SUBU	SUB
Resta (trap si desbordamiento)	SUB, SUBI	SUB; INTOVR	SUB	SUB	SUBcc; TVS

Nombre de instrucción	DLX	i860	MIPS	M88000	SPARC
Multiplicación (ver E.6 para SPARC)	MULTU, MULTUI	FMLOW	MULT, MULTU	MUL	MULScc;...; MULScc
Multiplicación (trap si desbordamiento)	MULT, MULTI	—	—	—	— (ver E.6)
División	DIVU, DIVUI	—	DIV, DIVU	DIV, DIVU	— (ver E.6)
División (trap si desbordamiento)	DIV, DIVI	—	—	—	— (ver E.6)
And	AND, ANDI	AND	AND, ANDI	AND	AND
Or	OR, ORI	OR	OR, ORI	OR	OR
Xor	XOR, XORI	XOR	XOR, XORI	XOR	XOR
Carga parte superior reg.	LHI	OR.H...,r0,...	LUI	OR.U...,r0,...	SETHI *(B fmt.)*
Desplazamiento lógico a la izquierda	SLL, SLLI	SHL	SLLV, SLL	MAK	SLL
Desplazamiento lógico a la derecha	SRL, SRLI	SHR	SRLV, SRL	EXTU	SRL
Desplazamiento aritmético a la derecha	SRA, SRAI	SHRA	SRAV, SRA	EXT	SRA
Comparación	S- ($<,>,\leq,\geq,=,\neq$)	SUB r0,...	SLT, SLTU, SLTI, SLTIU	CMP	SUBcc r0,...
Control (formatos de instrucción)	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**
Salto sobre comparación entera	BEQ, BNE	BC.T, BNC.T, BTE, BTNE	BEQ, BNE, B_Z ($<,>,\leq,\geq$)	BB1.N, BB0.N, BCND.N	Bicc ($<,>,\leq,\geq,=,\neq$)
Salto sobre comparación en punto flotante	BFPT, BFPF	BC.T, BNC.T	BC1T, BC1F	BB1.N, BB0.N, BCND.N	FBfcc ($\leq,\geq,\leq,\geq,=,...$)
Bifurcación, con registro de bifurcación	J, JR	BR, BRI	J, JR	BR.N, JMP.N	B, JMPL r0,...
Llamada, con registro de llamada	JAL, JALR	CALL, CALLI	JAL, JALR	BSR.N, JSR.N	CALL, JMPL
Trap	TRAP	TRAP	BREAK	TCND, TB0	Ticc
Retorno de interrupción	RFE	BRI *(bits de excepción $\neq$ 0)*	JR; RFE	RTE	RETT
Punto flotante (formatos de instrucción)	**R-R**	**R-R**	**R-R**	**R-R**	**R-R**
Suma simple, doble	ADDF, ADDD	FADD.SS, FADD.DD	ADD.S, ADD.D	FADD.SSS, FADD.DDD	FADDS, FADDD

Nombre de instrucción	DLX	i860	MIPS	M88000	SPARC
Resta simple, doble	SUBF, SUBD	FSUB.SS, FSUB.DD	SUB.S, SUB.D	FSUB.SSS, FSUB.DDD	FSUBS, FSUBD
Multiplicación simple, doble	MULF, MULD	FMUL.SS, FMUL.DD	MUL.S, MUL.D	FMUL.SSS, FMUL.DDD	FMULS, FMULD
División simple, doble	DIVF, DIVD	—, —	DIV.S, DIV.D	FDIV.SSS, FDIV.DDD	FDIVS, FDIVD
Comparación	_F, _D (<,>,≤,≥,=,...)	PF_.SS, PF_.DD (>,≤,=)	C_.S, C_.D (<,>,≤,≥,=,...)	FCMP.SS, FCMP.DD	FCMPS, FCMPD
Transferencia R-R	MOVF	FIADD.SS ...,f0,	MOV.S	ADD ..., r0,...	FMOVS
Convierte (simple, doble, entero) a (simple, doble, entero)	CVTF2D CVTD2F, CVTF2I, CVTD2I, CVTI2F, CVTI2D	FADD.SD ..f0.., FADD.DS ..f0.., FIX.SS, FIX.DS —, —	CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D	FADD.SSD r0, —, INT.SS, INT.SD, FLT.SS, FLT.DS	FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD
Convenios					
Registro con valor 0	r0	r0	r0	r0	r0
Reg. dirección de reparto	r31	r1	r31	r1	r31
No op	ADD r0,r0,r0	SHL r0,r0,r0	SLL r0,r0,r0	OR r0,r0,r0	SETHI r0,0
Transferencia de entero R-R	ADD ...,r0,...	SHL ...,r0,...	ADD ...,r0,...	OR ...,r0,...	OR ...,r0,...
Orden de operandos	OP Rd,Rs1,Rs2	OP Rs1,Rs2,Rd	OP Rd,Rs1,Rs2	OP Rd,Rs1,Rs2	OP Rs1,Rs2,Rd

FIGURA E.5 Instrucciones equivalentes a DLX. Los guiones significan que la operación no está disponible en esa arquitectura, o no se sintetiza con pocas instrucciones. Tal secuencia de instrucciones se muestra separada por puntos y comas. Si hay varias opciones de instrucciones equivalentes a DLX, están separadas por comas. Finalmente, «no aplicable» significa que esta operación no está directamente disponible y no tiene sentido para esa arquitectura. Esta última categoría es para el M88000, ya que las instrucciones enteras y de punto flotante comparten los mismos registros, lo que significa que es innecesario disponer de instrucciones adicionales de transferencia de punto flotante. Observar que en la categoría «Aritmética, lógica» DLX y MIPS utilizan nemotécnicos separados de instrucciones para indicar un operando inmediato, mientras que el i860, M88000 y SPARC ofrecen versiones inmediatas de estas instrucciones, pero utilizan un solo nemotécnico. (¡Por supuesto, éstos son códigos de operación separados!) Tanto MIPS como SPARC tienen nuevas instrucciones que no estaban implementadas en la primera máquina y que se aplican a algunos de estos casos: ver Secciones E.5 y E.6

DLX requiere una secuencia corta de instrucciones, éstas están separadas por punto y coma en la Figura E.5. (Para evitar confusiones, el registro destino será **siempre** el operando más a la izquierda en este apéndice, independientemente de la notación normalmente utilizada en cada arquitectura.)

Cada arquitectura debe tener un esquema de comparación y salto condicional, pero incluso con todas las analogías, cada una de estas arquitecturas ha determinado una forma diferente para realizar la operación. Las ventajas y desventajas de las opciones generales se encuentran en las páginas 112-117 del Capítulo 3.

Comparación y salto condicional

SPARC utiliza los cuatro bits tradicionales de código de condición almacenados en la palabra de estado del programa: Negativo, Cero, Acarreo y Desbordamiento. Se pueden inicializar con una instrucción aritmética o lógica, pero de forma distinta a arquitecturas anteriores esta inicialización es opcional en cada instrucción. Esto conduce a problemas menores en la implementación de la segmentación (página 359 del Capítulo 6). Mientras que los códigos de condición pueden ser inicializados como un efecto lateral de una operación, las comparaciones explícitas se sintetizan con una resta, utilizando `r0` como destino. El punto flotante utiliza códigos de condición separados para codificar las condiciones del IEEE 754, necesitando una instrucción de comparación en punto flotante. Los saltos condicionales de SPARC examinan los códigos de condición para determinar todas las posibles relaciones con signo y sin signo.

MIPS utiliza el contenido de los registros para evaluar saltos condicionales. Dos registros cualesquiera se pueden comparar para igualdad (`BEQ`) o desigualdad (`BNE`) y después se realiza el salto si se cumple la condición. Las instrucciones de inicializar sobre menor que (`SLT,SLTI,SLTU,SLTIU`) comparan dos operandos y después inicializan el registro destino a 1 si es menor y a 0 en otro caso. Estas instrucciones son suficientes para sintetizar el conjunto completo de relaciones. Debido a la popularidad de las comparaciones con 0, MIPS incluye instrucciones especiales de comparación y salto para dichas comparaciones: mayor o igual que cero (`BGEZ`), mayor que cero (`BGTZ`), menor o igual que cero (`BLEZ`) y menor que cero (`BLTZ`). Por supuesto, igual y no igual a cero pueden sintetizarse utilizando `r0` con `BEQ` y `BNE`. Igual que SPARC, MIPS utiliza un código de condición para punto flotante con instrucciones de salto y comparación de punto flotante.

El M88000 también utiliza registros para evaluar condiciones y optimiza las comparaciones con 0 mediante un conjunto separado de instrucciones de comparación y salto (`BCND.N`). La comparación de operandos arbitrarios difiere. MIPS ofrece varias instrucciones de comparación para poner el registro a 0 o 1, dependiendo de la condición seleccionada, mientras que el M88000 utiliza una sola instrucción (`CMP`) e inicializa 10 bits del registro destino, mostrando las relaciones entre los dos operandos. Estos bits representan la igualdad ($=$, $\neq$) más todas las relaciones para operandos con signo y sin signo ($<$, $\leq$, $>$, $\geq$). Las instrucciones que saltan si un bit de un registro es 1 (`BB1.N`)

o 0 (BBO.N) completan el repertorio de saltos condicionales. (Otra opción es utilizar EXTU con CMP para inicializar un registro a 0 o 1 y después utilizar BCND.N. Utilizando EXT, en lugar de EXTU, se inicializa un registro a 0 o −1, si se desea.) Como hay un conjunto común de registros para enteros y punto flotante, la comparación en punto flotante utiliza el mismo esquema: inicializa los bits de un registro y salta según el resultado utilizando BB1.N o BBO.N.

El Intel i860 utiliza códigos de condición para saltos como SPARC, excepto que los códigos de condición del i860 se inicializan implícitamente como parte de cada instrucción aritmética o lógica. También de forma distinta a SPARC, el i860 utiliza exactamente dos bits de condición: OF y CC. OF lo inicializan sólo las instrucciones de suma y resta entera, y se utiliza para indicar desbordamiento. No hay instrucciones de salto condicional para comprobar este bit, pero la instrucción INTOVR provoca un trap si el bit está a 1. El bit CC se pone a 1 o a 0 dependiendo de la operación. Las instrucciones lógicas (AND,OR,XOR) ponen a 1 CC si el resultado es 0. Las instrucciones de aritmética sin signo (ADDU,SUBU) ponen a 1 CC si hay un acarreo de salida del bit más significativo. La resta con signo (SUBS) pone a 1 CC si Rs2 > Rs1, mientras que la suma con signo (ADDS) pone a 1 CC si Rs2 es menor que el complemento a dos de Rs1. Las instrucciones de comparación de punto flotante ponen a 1 CC si la condición probada es cierta: mayor que (PFGT), menor o igual que (PFLE), o igual (PFEQ).

	DLX	i860	MIPS	M88000	SPARC
Número de bits del código de condición (entero y FP)	1 FP	1 ambos, 1 entero	1 FP	—	4 enteros, 2 FP
Instrucciones básicas de comparación (entero y FP)	1 entero 1 FP	1 FP	1 entero, 1 FP	1 entero, 1 FP	1 FP
Instrucciones básicas de salto (entero y FP)	1 entero, 1 FP	1 ambos, 1 entero	2 enteros, 1 FP	1 ambos, 1 enteros	1 entero, 1 FP
Compara registro con registro/ const y salta	$=, \neq$	$=, \neq$	$=, \neq$	—	—
Compara registro con cero y salta	$=, \neq$	$=, \neq$	$=, \neq, <, \leq, >, \geq$	$=, \neq, <, \leq, >, \geq$	—

FIGURA E.6 Resumen de cinco estrategias para los saltos condicionales. La comparación entera en el i860 y en el SPARC está sintetizada con una instrucción aritmética que pone a 1 los códigos de condición, utilizando r0 como destino.

Las instrucciones de salto condicional del i860 (BC.T y BNC.T) examinan CC y saltan dependiendo que CC esté a 1 o 0. El i860 también tiene instrucciones de salto condicional basándose en la igualdad de dos operandos: BTE bifurca si son iguales y BTNE bifurca si no lo son.

La Figura E.6 resume los cuatro esquemas utilizados para los saltos condicionales.

Multiplicación y división entera

La multiplicación y división se implementan usualmente como instrucciones multiciclo lo que no se corresponde con el objetivo de ejecución monociclo del resto de las instrucciones enteras, requiriendo integración separada en la segmentación. Cada arquitectura emplea una aproximación diferente para la división y multiplicación entera, así como para los saltos condicionales. El i860 utiliza el mismo esquema que DLX: hay una instrucción en punto flotante (`FMLOW`) que trata los contenidos de dos registros de punto flotante como enteros, dejando un resultado de 32 bits en los 32 bits inferiores de un par de registros de punto flotante en doble precisión. Los programas hacen la división entera utilizando instrucciones de punto flotante del i860. (La división de punto flotante utiliza la iteración de Newton-Raphson; ver páginas 763-764.)

El fichero de registros combinado, de punto flotante y enteros, permite al M88000 utilizar la unidad de punto flotante para realizar multiplicaciones y divisiones enteras, ya que los operandos no tienen que ser desplazados a y desde los registros de punto flotante. La única complicación en la primera versión de la arquitectura, la MC88100, es que un dividendo o un divisor negativo da como resultado un trap. El software entonces hace los operandos positivos, utiliza la instrucción de división, y después complementa el cociente (si es necesario). Un divisor cero también produce un trap, como es de esperar.

En la arquitectura MIPS el producto de 64 bits de una multiplicación entera o el resto/cociente de una división entera se colocan en un registro especial HI y LO. Este cálculo se trata como una unidad de ejecución independiente, en paralelo, con las unidades enteras y de punto flotante. El resultado apropiado es transferido al registro correcto con una instrucción `MFHI` o `MFLO`. Intentos para leer los registros antes que se complete la computación detienen el procesador. No hay trap para el desbordamiento o la división por cero. Estas condiciones se comprueban normalmente mediante instrucciones explícitas enteras que se ejecutan en paralelo con la división. (Ver Sección E.5 para extensiones arquitectónicas no implementadas en las primeras máquinas MIPS.)

SPARC proporciona una instrucción de multiplicación paso a paso. Cuando se utiliza en un bucle, calcula un producto completo de 64 bits utilizando el registro especial, y éste se carga con el multiplicador y, al final, contiene la palabra menos significativa del producto. Magenheimer, Peters, Pettis y Zuras [1988] midieron el tamaño de los operandos en multiplicaciones y divisiones para mostrar cómo funcionaría la multiplicación paso a paso. Utilizando este dato para programas C, Muchnick [1988] encontró que, tratando cada caso de forma especial, la multiplicación media por una constante emplea 6 ciclos de reloj y la multiplicación de variables emplea 24 ciclos de reloj. No hay división paso a paso en el SPARC. (Ver Sección E.6 para extensiones arquitectónicas no implementadas en las primeras máquinas SPARC.)

E.4 Instrucciones: extensiones comunes a DLX

La Figura E.7 lista las instrucciones que no aparecen en la Figura E.5, clasificadas en las mismas cuatro categorías. Las instrucciones de esta figura pertenecen a más de una de las cuatro arquitecturas. Las instrucciones se definen utilizando el lenguaje de descripción hardware, que se describe en la contraportada posterior y en las páginas 172-180 del Capítulo 4.

Nombre	Definición	i860	MIPS	M88000	SPARC
Transferencia de datos					
Intercambio atómico R/M (para semáforos)	Temp←Rd; Rd←Mem[x]; Mem[x]←Temp	`LOCK;LD.L; UNLOCK; ST.L;`	— (ver E.5)	`XMEM, XMEMBU`	`SWAP`
Carga doble entero	Rd←Mem[x]; Rd+1←Mem[x+4]	—	—	`LD.D`	`LDD`
Almacena doble entero	Mem[x]←Rd; Mem[x+4]←Rd+1	—	—	`ST.D`	`STD`
Carga coprocesador	Coprocesador← Mem[x]	—	`LWCi`	—	`LDC`
Almacena coprocesador	Mem[x]←Coprocesador	—	`SWCi`	—	`STC`
Endian	¿Grande/pequeño Endian?)	Grande o pequeño	Grande o pequeño	Grande o pequeño	Grande
Limpia cache	(Limpia, bloque de cache en esta dirección)	`FLUSH`	— (ver E.5)	—	`FLUSH`
Aritmética, lógica					
Soporte para suma entera multipalabra	Arrastre de salida, Rd←Rs1 + Rs2 + Arrastre de salida antiguo	`ADDU; BNC; ADDU ..., ..., #1`	`ADDU; SLTU; ADDU`	`ADDU.CIO`	`ADDXcc`
Soporte para resta entera multipalabra	Arrastre de salida, Rd←Rs1 − Rs2 + Arrastre de salida antiguo	`SUBU; BNC; ADDU ..., ..., #1`	`SUBU; SLTU; SUBU`	`SUBU.CIO`	`SUBXcc`
No And	Rd←Rs1 & !(Rs2)	`ANDNOT`	—	`AND.C` ***(R–R)***	`ANDN`
No Or	Rd←Rs1 | !(Rs2)	—	—	`OR.C` ***(R–R)***	`ORN`
No Xor	Rd←Rs1 ^ !(Rs2)	—	—	`XOR.C` ***(R–R)***	`XNOR`

Nombre	Definición	i860	MIPS	M88000	SPARC
Aritmética, lógica *(continuación)*					
And superior inmediato	$Rd_{0..15} \leftarrow Rs1_{0..15}$ & (Const << 16); $Rd_{16..31} \leftarrow 0$	ANDH ***(R–I)***	—	AND.U ***(R–I)***	—
Or superior inmediato	$Rd_{0..15} \leftarrow Rs1_{0..15}$ | (Const << 16); $Rd_{16..31} \leftarrow 0$	ORH ***(R–I)***	—	OR.U ***(R–I)***	
Xor superior inmediato	$Rd_{0..15} \leftarrow Rs1_{0..15}$ ^ (Const << 16); $Rd_{16..31} \leftarrow 0$	XORH ***(R–I)***	—	XOR.U ***(R–I)***	—
Operaciones del coprocesador	(Definido por el coprocesador)	—	COPi	—	CPop
Control					
Saltos retardados optimizados	(Salto no siempre retardado)	BC, BNC	—	BB1, BB0, BCND	Bicc, A
Saltos punto flotante optimizados	(Salto no siempre retardado)	BC, BNC	—	BB1, BB0, BCND	Bfcc, A
Trap condicional	if(COND) {R31←PC; PC←0..0#i}	—	— (ver E.5)	TB1, TB0, TCND	Ticc
Salto sobre coprocesador	if(CoProc COND) {PC←PC+Cons}	—	BCiT, BCiF	—	Bccc
Núm. regs. control	Regs. Misc. (memoria virtual, interrupciones...)	6	12	32	7
Punto flotante					
Negación	Fd←Fs ^ x800000000	—	NEG.S, NEG.D	XOR.U 8000	NEGS
Valor absoluto	Fd←Fs & x7FFFFFFF	—	ABS.S, ABS.D	AND.U 7FFF	ABSS
Truncar a entero	Fd←parte entera no redondeada de Fs	FTRUNC.SS, FTRUNC.DS	—	TRNC.SS, TRNC.SD	—
Conversiones implícitas	Convierte como parte de operación	_.SD (2 operandos simples, 1 resultado doble)	—	_.SSD, _.SDS, _.SDD, _.DSS, _.DSD. _.DDS (todas las combinaciones)	—

FIGURA E.7 Instrucciones que no están en DLX, pero que se encuentran en dos o más de las cuatro arquitecturas. Tanto MIPS como SPARC tienen nuevas instrucciones que no se implementaron en la primera máquina y que se aplican a algunos de estos casos: ver Secciones E.5 y E.6.

Aunque la mayoría de las categorías son autoexplicativas, hagamos algunos comentarios:

- La fila «Intercambio atómico» significa una primitiva que puede intercambiar un registro con memoria sin interrupción. Esto es útil para los semáforos de los sistemas operativos en uniprocesadores, así como para la sincronización de multiprocesadores (ver páginas 508-511 del Capítulo 8.)
- En la fila «Endian», «Grande o Pequeño» también significa que hay un bit en el registro de estado del programa que permite que el procesador actúe como Gran «Endian» o Pequeño «Endian». Esto puede realizarse complementando simplemente alguno de los bits menos significativos de la dirección en las instrucciones de transferencia de datos.
- La fila de «Operaciones del coprocesador» lista diversas categorías que permiten que el procesador se extienda con hardware de propósito especial.
- La fila «Conversiones implícitas» bajo «Punto flotante» significa que los operandos de punto flotante en estas arquitecturas no tienen todos el mismo tamaño, y la unidad de punto flotante realiza una conversión como parte de la operación. El i860 permite dos operandos en simple precisión para producir un resultado en doble precisión, mientras que el M88000 permite cualquier combinación de simple y doble precisión para cada uno de los tres operandos.

Una diferencia que necesita una explicación más larga es la de los saltos optimizados. La Figura E.8 muestra las opciones. El i860 y el M88000 ofrecen saltos que tienen efecto inmediatamente, como los saltos en las primeras arquitecturas. Esto evita la ejecución de NOP cuando no hay ninguna instrucción para rellenar el hueco de retardo. SPARC proporciona una versión de salto retardado que hace más fácil rellenar el hueco de retardo. El salto con «anulación» ejecuta una instrucción en el hueco de retardo sólo si el salto es efectivo; en cualquier otro caso la instrucción se anula. Esto significa que la instrucción en el destino del salto se puede copiar sin ningún problema en el hueco de retardo, ya que sólo se ejecutará si el salto es efectivo. Las restricciones son que el destino no sea otro salto y que el destino no se conozca en tiempo de compilación. SPARC también ofrece una bifurcación no retardada, ya que un salto incondicional con el bit de anulación **no** ejecuta la instrucción siguiente.

Después de cubrir las analogías, analizaremos las características únicas de cada arquitectura, ordenándolas por su longitud de descripción de las más cortas a las más largas.

	Salto retardado	**Salto (común)**	**Anular salto retardado**
Encontrado en arquitecturas	Todos los 5 RISC	i860, M88000	SPARC
Ejecuta instrucción siguiente	Siempre	Sólo si salto **no** efectivo	Sólo si salto efectivo

FIGURA E.8 En qué casos la instrucción que sigue al salto se ejecuta para tres tipos de saltos.

E.5 Instrucciones únicas del MIPS

Comenzamos con las instrucciones de transferencia de datos. MIPS es distinta de las demás, ya que la arquitectura requiere que la instrucción que sigue a una carga no referencie el valor que se está cargando. El Ensamblador MIPS inserta una instrucción NOP si se presenta esta situación.

Transferencias de datos no alineados

La otra característica, única, de las transferencias de datos de MIPS son las instrucciones especiales para manejar palabras no alineadas en memoria. Este evento es raro en muchos programas, pero ocurre en los programas COBOL, donde el programador puede forzar mal alineamiento mediante declaraciones. Aunque todas estas arquitecturas producen un trap si se intenta cargar o almacenar una palabra en una dirección mal alineada, en todas las arquitecturas las palabras mal alineadas pueden ser accedidas sin traps, utilizando 4 instrucciones de carga byte y después ensamblando el resultado utilizando desplazamientos y operaciones lógicas OR. Las instrucciones MIPS de carga y almacenamiento de palabra (`LWL,LWR,SWL,SWR`) permiten que se haga esto en 2 instrucciones: `LWL` carga la parte izquierda del registro y `LWR` la derecha. `SWL` y `SWR` realizan los almacenamientos correspondientes. La Figura E.9 muestra cómo funcionan. De forma distinta a otras cargas, `LWL` seguida por `LWR` no requiere una NOP aun cuando ambas especifiquen el mismo registro, ya que los campos no se solapan.

Instrucciones TLB

Los fallos de TLB se manejan por software en el MIPS R2000, así que el repertorio de instrucciones también tiene instrucciones para manipular los registros del TLB (ver páginas 471-473 y 477-480 del Capítulo 8 para más sobre TLB). Estos registros están considerados parte del «coprocesador del sistema» y por tanto pueden ser accedidos por las instrucciones que hacen transferencias *(move)* entre los registros del coprocesador y los registros enteros. El contenido de una entrada del TLB se lee cargando mediante una lectura Entrada TLB Indexada (`TLBR`) y se escribe utilizando bien una Escritura Entrada TLB Indexada (`TLBWI`) o Escritura Entrada TLB Aleatoria (`TLBWR`). Los contenidos del TLB se examinan utilizando una Búsqueda en el TLB de una Entrada Coincidente (`TLBP`).

Instrucciones restantes

A continuación, se da una lista de los restantes detalles únicos de la arquitectura MIPS:

- *NOR.* Esta instrucción lógica calcula !(Rs1 | Rs2).

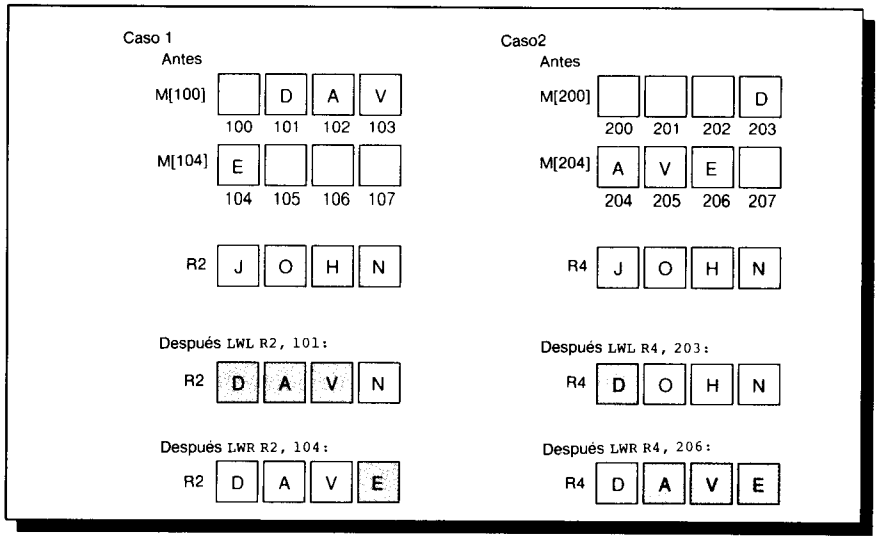


FIGURA E.9 Instrucciones de MIPS para lecturas de palabras no alineadas. Esta figura supone operar en modo Gran Endian. El caso (1) primero carga los 3 bytes 101, 102 y 103 a la izquierda de R2 dejando inalterado el byte menos significativo. La siguiente `LWR` simplemente carga el byte 104 en el byte menos significativo de R2, dejando a los demás bytes del registro inalterados utilizando `LWL`. El caso (2) primero carga el byte 203 en el byte más significativo de R4 y la siguiente `LWR` carga los otros 3 bytes de R4 desde los bytes de memoria 204, 205 y 206. `LWL` lee la palabra con el primer byte de memoria, desplaza a la izquierda para descartar el (los) byte(s) innecesario(s), y cambia sólo esos bytes de Rd. El/los byte(s) transferidos son desde el primer byte hasta el byte de orden más bajo de la palabra. La siguiente `LWR` direcciona el último byte, desplaza a la derecha para descartar el (los) byte(s) innecesario(s) y finalmente cambia sólo esos bytes de Rd. Los bytes transferidos son desde el último byte hasta el byte de orden más alto de la palabra. Almacenar palabra izquierda (`SWL`) es simplemente lo inverso de `LWL`, y almacenar palabra derecha (`SWR`) es lo inverso de `LWR`. Cambiar al modo de Pequeño Endian intercambia los bytes que se seleccionan y descartan. (Si grande/pequeño-izquierda/derecha-carga/almacenamiento parace confuso, no preocuparse, ¡funciona!)

- *Cantidad de desplazamiento constante.* Los desplazamientos no variables utilizan el campo constante de 5 bits, mostrado en el formato registro-registro de la Figura E.3.
- *SYSCALL.* Esta instrucción especial se utiliza para invocar el sistema operativo.
- *Transferir a/desde registros de control.* `CTCi` y `CFCi` transfieren entre los registros enteros y de control.
- *Registros limitados a simple precisión.* Aunque los 32 registros de punto flotante pueden ser direccionados individualmente para cargas y almacenamientos, los operandos de simple precisión para operaciones de punto flotante sólo pueden utilizar los 16 registros pares de punto flotante.

- *Bifurcación/Llamada no relativa al PC.* La dirección de 26 bits de las bifurcaciones y de las llamadas no se suma al PC. Se desplaza 2 bits a la izquierda y sustituye los 28 bits inferiores del PC. El resultado de este esquema sería diferente al de suma sólo si el programa estuviese localizado cerca del límite de 256 MB.
- *Instrucciones de llamada condicional a procedimiento.* `BGEZAL` guarda la dirección de retorno y salta si el contenido de Rs1 es mayor o igual que cero y `BLTZAL` hace lo mismo para menor que cero. El propósito de estas instrucciones es obtener una llamada relativa al PC.

No hay provisión específica en la arquitectura MIPS para que la ejecución en punto flotante proceda en paralelo con la ejecución entera, pero las implementaciones MIPS de punto flotante permiten que ocurra esto comprobando si las interrupciones aritméticas son posibles al principio del ciclo; normalmente las interrupciones no son posibles y enteros y punto flotante operan en paralelo (ver página 672 el Apéndice A).

MIPS II

Con el anuncio del R6000 apareció un conjunto de extensiones de la arquitectura MIPS original, descrita anteriormente. Aquí están las adiciones de MIPS II:

- *Cargas interbloqueadas.* El Ensamblador MIPS II no necesita insertar una NOP después de una carga si hay una dependencia sobre la siguiente instrucción, ya que el hardware se detendrá automáticamente.
- *Saltar probablemente.* Equivalente a los saltos anulados de SPARC, esta instrucción ejecuta la instrucción del hueco de retardo sólo si el salto es efectivo.
- *Carga punto flotante doble y almacenar punto flotante doble.* MIPS II emplea una sola instrucción para cargar o almacenar números de punto flotante en doble precisión.
- `SQRT`. La raíz cuadrada en punto flotante en simple y doble precisión se añade a las operaciones en punto flotante.
- *Instrucciones de trap condicional.* Estas coinciden con las instrucciones de salto condicional, excepto que no son retardadas: cuando el trap es efectivo, **no** se ejecuta la siguiente instrucción. Estas instrucciones son útiles para comprobación de rango, popular en Ada.

E.6 Instrucciones únicas del SPARC

Ventanas de registros

La principal característica única de SPARC son las ventanas de registros (páginas 485-489 del Capítulo 8), utilizadas para reducir el gasto de guardar/res-

taurar en las llamadas y retornos de los procedimientos. SPARC puede tener entre 2 y 32 ventanas, cada una de las cuales utiliza 8 registros para las variables locales, 8 para las globales, 8 para los parámetros de entrada y 8 para los de salida (ver Figura 8.34). (Dado que cada ventana tiene 16 registros únicos, una implementación de SPARC puede tener como mínimo 40 registros físicos y como máximo 520, aunque la mayoría tengan entre 128 y 136 en la actualidad.) En lugar de ligar los cambios de ventana con las instrucciones de llamada y retorno, SPARC tiene instrucciones separadas SAVE (GUARDAR) y RESTORE (RESTAURAR). SAVE se utiliza para «guardar» la ventana del llamador pasando a apuntar a la siguiente ventana de registros además de realizar una instrucción de suma. El truco está en que los registros fuente de la operación de suma provienen de la ventana del llamador, mientras que el registro destino está en la ventana llamada. Los compiladores SPARC, normalmente, utilizan esta instrucción para cambiar el puntero de pila con el fin de ubicar variables locales en una nueva estructura de pila. RESTORE es la inversa de SAVE, retornando la ventana del llamador mientras que actúa como una instrucción de suma, con los registros fuente de la ventana llamada y el registro destino en la ventana del llamador. Esto desasigna automáticamente la estructura de pila. Los compiladores también pueden utilizarla para generar el valor retornado por el procedimiento llamado. De forma distinta a las primeras arquitecturas de ventanas de registro, SPARC utiliza una Máscara de Invalidez de ventana, que se utiliza en aplicaciones de tiempo real y que permite que las ventanas se repartan entre diferentes procesos.

Otra característica de la transferencia de datos es la opción de espacio alterno para cargas y almacenamientos. Esto permite, simplemente, que la memoria del sistema identifique accesos a memoria para dispositivos de entrada/salida, o registros de control para dispositivos como la cache y la unidad de gestión de memoria.

Soporte para LISP y Smalltalk

La principal característica aritmética que queda es la de suma y resta etiquetadas. Los diseñadores de SPARC reflexionaron mucho tiempo sobre lenguajes como LISP y Smalltalk, y esto influyó en algunas de las características de SPARC ya comentadas: ventanas de registros, instrucciones de trap condicional, llamadas con direcciones de instrucciones de 32 bits, y aritmética multipalabra (ver Taylor [1986] y Ungar [1984]). Se ofrece algún soporte para tipos de datos señalizados en operaciones de suma, resta y, por tanto, comparación. Los dos bits menos significativos indican si el operando es entero (codificado 00); así, TADDcc y TSUBcc inicializan el bit de desbordamiento si algún operando no está etiquetado como entero o si el resultado es demasiado grande. Una instrucción de trap o de salto condicional puede decidir qué hacer. (Si los operandos no son enteros, el software recupera los operandos, comprueba los tipos de operandos, e invoca la operación correcta basada en estos tipos.) Otras dos versiones de estas instrucciones hacen innecesario el trap condicional, ya que TADDccTV y TSUBccTV causan un trap si el desbordamiento está a 1. El trap de acceso a memoria mal alineado pueda también utilizarse con datos etiquetados, ya que cargar desde un puntero con la etiqueta errónea puede

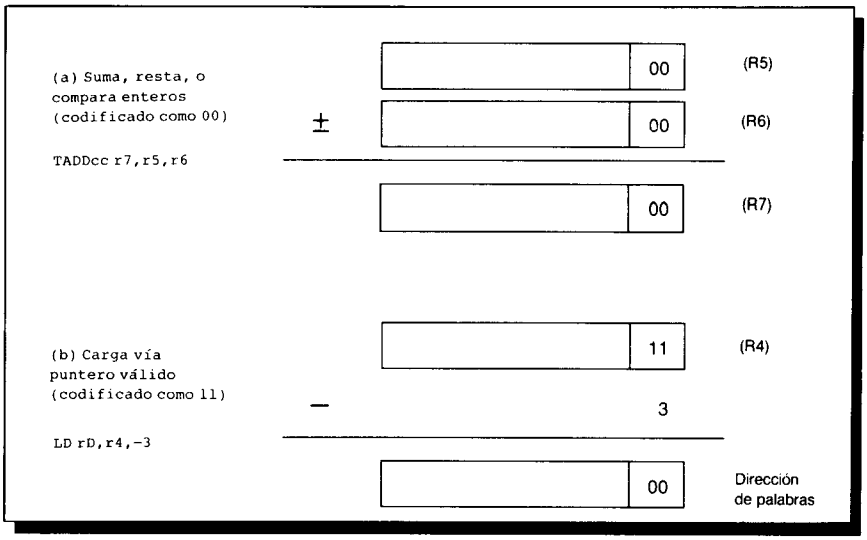


FIGURA E.10 SPARC utiliza los dos bits menos significativos para codificar diferentes tipos de datos para las instrucciones aritméticas etiquetadas. (a) muestra aritmética entera, que emplea un solo ciclo, siempre que los operandos y el resultado sean enteros; (b) muestra que el trap de mal alineado se puede utilizar para detectar accesos inválidos a memoria, como tratar de utilizar un entero como puntero. Para lenguajes con datos emparejados como LISP, se puede utilizar un desplazamiento de −3 para acceder a la palabra par de un par (CAR) y de +1 para la palabra impar de un par (CDR).

ser un acceso inválido. La Figura E.10 muestra ambos tipos de soporte para etiquetas.

Operaciones solapadas enteras y de punto flotante

SPARC permite que la ejecución de las instrucciones de punto flotante se solape con la de las instrucciones enteras. Para recuperarse de una interrupción durante esa situación, SPARC tiene una cola de instrucciones de punto flotante pendientes y sus direcciones. `STDFQ` permite que el procesador vacíe la cola. La segunda característica de punto flotante es la inclusión de instrucciones para el cálculo de la raíz cuadrada en punto flotante, `FSQRTS` y `FSQRTD`.

Instrucciones restantes

Las restantes características distintivas de SPARC son:

- *JMPL* utiliza Rd para especificar el registro con la dirección de retorno; así, especificar `r31` es similar a `JALR` en DLX y especificar `r0` hace lo mismo que `JR`.
- *LDSTUB* carga el valor del byte en Rd y después almacena FF_{16} en el byte direccionado. Esta instrucción se puede utilizar para implementar un semáforo.

- *LDDC* y *STDC* proporcionan carga doble y almacenamiento doble para el coprocesador.
- *UNIMP* provoca una interrupción de instrucción no implementada. Muchnick [1988] explica cómo se utiliza para la ejecución adecuada de procedimientos de retorno agregados en C.

Finalmente, SPARC incluye códigos de operación para instrucciones que son emuladas en software en las primeras implementaciones. Los programas de aplicación de SPARC, generalmente, llaman a rutinas de biblioteca dinámicamente enlazadas para realizar estas operaciones, pero los códigos de operación producirán un trap si se ejecutan. Las instrucciones son:

- *Multiplicación y división entera con signo y sin signo*, estando ambos operandos y el resultado en registros enteros. Los 32 bits extra de un producto y los 32 bits del resto de una división se colocan en el registro Y.
- *Aritmética de punto flotante de cuádruple precisión*, que permite a los registros de punto flotante que actúen como ocho registros de 128 bits.
- *Resultados de punto flotante en múltiple precisión para la multiplicación*, significa que dos operandos en simple precisión pueden originar un producto en doble precisión y dos operandos en doble precisión pueden originar un producto en cuádruple precisión. Estas instrucciones pueden ser útiles en aritmética compleja y en algunos modelos de cálculos de punto flotante.

E.7 Instrucciones únicas del M88000

La característica más distintiva del M88000 es el conjunto de 32 registros compartido por operaciones enteras y de punto flotante. Esto simplifica el repertorio de instrucciones a costa de menos registros para programas en punto flotante.

Instrucciones de bits

La siguiente característica distintiva del M88000 es un conjunto completo de instrucciones de campos de bits, mostradas en la Figura E.11. (Aunque habitualmente numeramos el bit más significativo como 0, en esta tabla seguimos la notación de Motorola, que numera el bit más significativo como 31 y el menos significativo como 0.) Las instrucciones de campos de bits necesitan un operando extra para especificar la anchura del campo además del registro destino, registro fuente y comienzo del campo de bits. Este campo de 5 bits de ancho se localiza a continuación del campo de bits en fuente 2. El M88000 codifica una anchura de 0 para significar el valor completo de 32 bits; por tanto, las instrucciones tradicionales de desplazamiento (SLL,SRL,SRA) son simplemente las instrucciones correspondientes de campo de bits (MAK,EXTU,EXT) cuya anchura de campo es 0.

Nombre	Instrucción	Notación
CLR	Pone a 0 campo de bits	$Rd_{(o+w)..(o+1)} \leftarrow 0^w$
SET	Pone a 1 campo de bits	$Rd_{(o+w)..(o+1)} \leftarrow 1^w$
EXT	Extrae campo de bits con signo	if (w==0) {Rd ← $Rsl_{31}{}^{\circ}$ ## (Rsl >> o) } else {Rd ← ($Rsl_{(o+w)}{}^{\circ}$ ## ($Rsl_{(o+w)..(o+1)}$ >> 0) }
EXTU	Extrae campo de bits sin signo	if (w==0) {Rd←0° ## (Rsl >> o) } else {Rd ← 0° ## ($Rsl_{(o+w)..(o+1)}$ >> o) }
MAK	Crea campo de bits	if (w==0) {Rd ← Rsl << o} else {$Rd_{(o+w)..(o+1)}$ ← $Rsl_{(w-1)..0}$ }
ROT	Desplazamiento circular a la derecha	Rd ← $Rsl_{(o-1)..0}$ ## $Rsl_{31..o}$
FF0	Encuentra primer bit a 0	for (i=31; $Rs2_i$==0‖i<0; i ← i−1); /*bucle hasta = 0*/ if(i<0) {Rd ← 32} else {Rd ← i}
FF1	Encuentra primer bit a 1	for (i=31; $Rs2_i$==1‖i<0; i ← i−1); /* bucle hasta=1*/ if(i<0) {Rd ← 32} else {Rd ← i}

FIGURA E.11 Instrucciones de campo de bits del M88000. El desplazamiento de bits, o, son los cinco bits menos significativos del segundo operando y la anchura de campo de bits, w, son los cinco bits que siguen al desplazamiento. La notación de subíndices especifica un campo de bit, mientras que la notación de superíndices significa replicar ese bit muchas veces. Observar que en esta tabla, el bit 31 referencia al bit más significativo, y 0 al bit menos significativo.

Instrucciones restantes

Las instrucciones finales únicas son cargar dirección (LDA), MASK, redondear al entero más próximo (NINT), trap en límites (TBND) e intercambio del registro de control (XCR):

- LDA carga Rd con la dirección efectiva en lugar del dato de memoria. La única vez que es diferente de ADDU es para el direccionamiento escalado de datos diferentes a bytes.
- MASK es simplemente otro caso de AND lógico inmediato: esta instrucción borra la otra mitad de la palabra mientras que el AND inmediato la deja inalterada. Por ello, ANDI en DLX se puede argumentar que es más parecida a MASK que a AND inmediato en el M88000.
- NINT difiere de INT en que redondea el entero más próximo sin importar el modo de redondeo que se utilice (ver Apéndice A, páginas 657-658).
- TBND produce un trap si Rs1 > Rs2, tratándolos como números sin signo (ver página 256 en el Capítulo 5 para una explicación de cómo una comparación sin signo puede comprobar a la vez dos límites con signo).
- XCR intercambia un registro de control con un registro entero.

Además de las instrucciones, citamos algunas características que distinguen el M88000:

- Las operaciones de doble longitud utilizan Rn y Rn+1 en lugar de un par de registros par-impar. Esto da al M88000 más flexibilidad en la ubicación de registros, que es importante dada la falta de registros de punto flotante.
- La primera implementación, el MC88100, permite que todas las instrucciones multiciclo solapen la ejecución con las instrucciones siguientes, a menos que haya un riesgo de datos (ver páginas 284-285 en el Capítulo 6). También, todas las instrucciones de punto flotante, excepto la división,están segmentadas, empleando exactamente un ciclo para realizar las operaciones en simple precisión y dos ciclos para las de doble precisión. El 88000 proporciona un conjunto de registros de sombra (ver Sección 5.6) para los operandos en punto flotante para ayudar al software a manipular interrupciones precisas e imprecisas (ver Motorola [1988]).
- Hay transferencias especiales de datos, que se identifican al añadir .USR a las instrucciones, que permiten acceder a los datos del usuario desde modo supervisor.

E.8 Instrucciones únicas del i860

El i860 tiene muchas características únicas. Antes de cubrir las extensiones especiales para los gráficos y punto flotante de alto rendimiento, citamos las áreas tradicionales.

Las únicas transferencias de datos son sólo para punto flotante. El i860 proporciona cargas de 128 bits (`FLD.Q`) y almacenamientos (`FST.Q`) de pares de registros de punto flotante de 64 bits. También proporciona un modo de direccionamiento, opcional, en todas las cargas de punto flotante y almacenamientos: la dirección efectiva (suma de Rs1/Const y Rs2) se vuelve a almacenar en Rs2. Una característica única es que al i860 parece que le falten bits de código de operación para las instrucciones de carga, porque utiliza el bit menos significativo para distinguir la carga de media palabra de la carga de una palabra. Esto funciona bien para el formato registro-registro, ya que el bit 0 es una extensión del campo del código de operación en este formato, pero en el formato registro-inmediato éste es el bit menos significativo del campo de constante. Para evitar problemas disparatados de direccionamiento, este bit se pone a 0 cuando se utiliza como dirección. Esto impide tener un valor impar en un registro índice, que se corrige por una dirección de byte impar en el campo de constante, para transferencias de medias palabras y palabras de datos (ver E.10(b) en página 760 para una razón por la que esto es útil).

La única instrucción aritmética-lógica es un desplazamiento lógico a la derecha de doble longitud (`SHRD`). Rs1 y Rs2 son desplazados a la derecha como un par, y después se colocan los 32 bits menos significativos en Rd. Como no hay sitio en la instrucción para especificar el número de desplazamientos, `SHRD` utiliza el número de desplazamientos de la última instrucción `SHR`. Este valor se almacena en el campo SC de 5 bits de la palabra de estado del programa.

Al mismo tiempo, SHRD se puede utilizar para realizar un desplazamiento circular de 32 bits haciendo que Rs1 y Rs2 sean el mismo registro.

Entre las instrucciones de control del i860 se encuentra una instrucción de bucle denominada BLA. Esta instrucción realiza una suma y un salto condicional. Como es probable que otra instrucción del bucle pueda cambiar el código de condición, el i860 tiene un código de condición de bucle especial (LCC) sólo para esta instrucción. BLA realiza Rd ← Rs1 + Rs2 y salta si LCC es igual a 1. Además, BLA pone a 1 LCC para la próxima vuelta del bucle si Rs2 $\geqslant$ −Rs1 y lo pone a 0 en caso contrario. (LCC se inicializa con el valor opuesto con el que ADDS inicializa CC.)

Aunque el i860 no tiene división en punto flotante, tiene una instrucción recíproca de punto flotante (FRCP). Usada con la iteración de Newton-Raphson (páginas 664-666 del Apéndice A), ésta calcula divisiones que difieren del estándar de punto flotante del IEEE (IEEE 754) en los 2 bits menos significativos. Intel ofrece software para producir el resultado correctamente redondeado, utilizando el doble de ciclos. Una instrucción similar, FRSQR, calcula un paso recíproco para la raíz cuadrada. Las instrucciones de punto flotante también incluyen suma y resta enteras de 64 bits (FIADD.DD y FISUB.DD) usando los registros de punto flotante.

Hasta aquí se han tratado las características distintivas de las categorías tradicionales, ahora describamos las nuevas categorías del i860.

Instrucciones gráficas

Las instrucciones gráficas o instrucciones de *pixels* del i860 operan sobre 64 bits de datos a la vez, representando cada palabra varios pixels. Las instrucciones de «pixel» se consideraron para que fuesen útiles en las operaciones de gráficos, tales como la eliminación de superficies ocultas (ver página 566 en el Capítulo 9), interpolación de distancias, y sombreado tridimensional utilizando interpolación de intensidades. Estas instrucciones de propósito especial no son fáciles de comprender, así que los lectores interesados deberán acudir al manual para los detalles.

La visión general de las operaciones es que dos bits de la palabra de estado del programa determinan el tamaño de los pixels en una palabra de 64 bits. Los pixels pueden ser de 8, 16 o 32 bits, conteniendo campos en cada tamaño que representan la intensidad de los colores fundamentales rojo, azul y verde. Algunas instrucciones de pixels funcionan con un acumulador de 64 bits llamado registro MERGE (MEZCLA), útil para recopilar los resultados de una serie de cálculos sobre pixels. Además de las instrucciones de «mezcla» (FADDP y FADDZ), el i860 tiene instrucciones para buffers z (página 566) que comparan dos conjuntos de cuatro de valores de 16 bits (FZCHKS) o dos de 32 bits (FZCHKL), almacenando los valores más pequeños en el registro destino de 64 bits e inicializando los bits para indicar cuál fue más pequeño en la palabra de estado del programa. Las instrucciones de almacenamiento de pixels (PST) utilizan entonces esos bits para almacenar selectivamente sólo aquellos pixels que son más pequeños. Finalmente, la instrucción FORM se utiliza para transferir el registro MERGE a un registro de punto flotante y después borrar MERGE.

Modo segmentado

Para mayor rendimiento, el i860 ofrece versiones segmentadas de todas las instrucciones de pixel y de punto flotante. Un modelo para estas instrucciones es utilizarlas para construir primitivas vectoriales, permitiendo que se escriban procedimientos para implementar operaciones vectoriales (ver Capítulo 7). La esperanza es que los compiladores vectoriales, existentes, puedan invocar estos procedimientos más eficientes. Otro modelo, utilizado por los compiladores actualmente bajo desarrollo por orden de Intel, intenta compilar directamente utilizando estas instrucciones para códigos vectoriales y no vectoriales.

En el *modo segmentado,* cada ciclo se realiza una instrucción, pero de forma distinta a otras máquinas segmentadas, no hay hardware para recordar dónde se van a almacenar los resultados. ¡Básicamente, la instrucción en emisión en el momento en que se completa una operación especifica el destino! Hay cuatro segmentaciones independientes en el i860, y cada segmentación avanza solamente cuando se ejecuta la siguiente instrucción de ese tipo. La Figura E.12 muestra las segmentaciones del i860, su número de etapas y las instrucciones que hacen avanzar cada segmentación. Por tanto, los campos de fuente y código de operación especifican la operación que se va a ejecutar mientras que el campo destino especifica el registro que se va a cargar por una instrucción del mismo tipo que ya está en la etapa final en este ciclo.

Por ejemplo, observar la secuencia que sigue para la segmentación del sumador de punto flotante (suponer que los operandos se especifican con el resultado en la izquierda):

```
PFADD.SS    F4, F2, F3        ;Suma Simple Prec.
PFSUB.DD    F10, F8, F6       ;Resta Doble Prec.
PFMUL.DD    F16, F12, F14     ;Mul Doble Prec.
PFADD.SS    F19, F17, F18     ;Suma Simple Prec.
PFADD.SS    F22, F20, F21     ;Suma Simple Prec.
```

Segmentación	Núm. de etapas	Instrucciones que usan segmentación
Multiplicador FP	3 (operandos simples) 2 (operandos dobles)	PFMUL
Sumador FP	3	PFADD, PFSUB, PFGT, PFLE, PFEQ, PFIX, PFTRUNC
Carga FP	3	PFLD
Gráficos	1	PFIADD, PFISUB, PFZCHKS, PFZCHKL, PFADDP, PFADDZ, PFORM

FIGURA E.12 Segmentaciones del i860, incluyendo el número de etapas de la segmentación. Todas las instrucciones del sumador y multiplicador permiten operandos en simple precisión con resultados en simple precisión (.SS), operandos en simple precisión con resultados en doble precisión (.SD) y operandos en doble precisión con resultados en doble precisión (.DD). Como el número de etapas difiere para la multiplicación dependiendo que sea en simple o doble precisión, Intel recomienda no mezclar precisiones que involucren la multiplicación.

La segmentación del sumador de punto flotante tiene tres etapas; la primera instrucción realiza una suma en punto flotante de F2 y F3, pero F4 se carga con la operación realizada en el sumador tres instrucciones antes. La multiplicación en esta secuencia no hace avanzar la segmentación del sumador; así que la tercera instrucción del sumador que sigue a la primera instrucción (una resta y dos sumas) es la instrucción final de la secuencia, significando que F22←F2+F3.

La carga segmentada tiene una interacción interesante con la cache de datos. Mientras el dato está en la cache, se busca en la cache. En un fallo, el dato se busca en memoria, pero la cache no se actualiza con el nuevo dato. Esta política evita que operaciones sobre grandes estructuras de datos llenen la cache con datos que no son reutilizados y que se pierdan datos que se puedan reutilizar. El programador debe decidir si utilizar o no cargas escalares (FLD) o cargas segmentadas (PFLD), dependiendo que el dato se vaya a reutilizar o no.

Las instrucciones escalares, normalmente, vacían la segmentación. (La excepción es la carga segmentada porque FLD o LD no lo vacían.) Por tanto, antes de ejecutar una instrucción escalar de punto flotante debe haber una secuencia de instrucciones segmentadas ficticias que almacenen el resultado. Por ejemplo, no hay versión segmentada de la instrucción en punto flotante utilizada para la multiplicación entera (FMLOW); así que la segmentación debe ser drenada si se necesita una multiplicación entera durante un cálculo de punto flotante.

Resumiendo el modo segmentado en el i860, las ventajas son:

- El control de la segmentación es sencillo (básicamente se hace en software).
- No necesita muchos registros, ya que no están reservados durante la operación.

Las desventajas son:

- Deben realizarse operaciones para vaciar la segmentación.
- El mecanismo de interrupciones es complicado, empleando mucho tiempo en recuperar el estado.
- A veces es difícil utilizar la segmentación.
- El tamaño del código puede crecer rápidamente (esto todavía no se ha cuantificado).

Suma/resta y multiplicación

Para lograr aún más rendimiento de la unidad de punto flotante, el i860 tiene instrucciones segmentadas que realizan simultáneamente una suma y una multiplicación (PFAM y PFMAM) o una resta y multiplicación (PFSM y PFMSM), avanzando las segmentaciones de ambas unidades de suma y multiplicación. Como cada instrucción necesita 4 fuentes y 2 destinos, el i860 tiene tres registros que también se pueden utilizar además de los tres registros de punto flotante especificados en la instrucción. Los registros KI y KR, opcionalmente

cargados desde Rs1, pueden ser fuentes para el multiplicador, y el registro T puede ser un destino del multiplicador o una fuente para el sumador. La etapa final de la segmentación del sumador y del multiplicador también pueden ser fuentes. Cuatro bits de cada instrucción especifican una serie de combinaciones de los operandos y las operaciones.

Modo dual de instrucción

Finalmente, el i860 permite que se busquen y ejecuten simultáneamente una instrucción entera y otra de punto flotante. Esta palabra larga de instrucción o forma superescalar de operación (páginas 341-346 del Capítulo 6) se denomina *modo dual de instrucción* en el i860. La ejecución simultánea ocurre en este modo cuando la instrucción superior de una doble palabra alineada es una instrucción entera y la inferior es una instrucción en punto flotante con el bit «D» a 1 (bit 9 = 1). La entrada o salida del modo está retardada: cuando el i860 encuentra una instrucción con el bit D a 1, ejecuta una instrucción más, antes de entrar en el modo de instrucción dual; y, análogamente, cuando el i860 está en el modo de instrucción dual y encuentra un bit D no a 1, ejecuta un par más, antes de ir a la ejecución secuencial.

Claramente, el mayor rendimiento se obtiene cuando el i860 está en ambos modos, segmentado y de instrucción dual.

E.9 Observaciones finales

Este apéndice cubre los modos de direccionamiento, formatos de instrucción y todas las instrucciones encontradas en cuatro arquitecturas recientes. Aunque las últimas secciones se concentran en las diferencias, no habría sido posible cubrir cuatro arquitecturas en estas pocas páginas si no hubiese muchas analogías. En efecto, se puede intuir que más del 90 por 100 de las instrucciones ejecutadas para cualquiera de estas arquitecturas se encuentra en la Figura E.3. Para ilustrar esta homogeneidad, la Figura E.13 da un resumen para cuatro arquitecturas de los años setenta, análogo a la Figura E.1. (Imagínense intentar escribir un solo apéndice de este estilo para esas arquitecturas.) En la historia de la computación, no ha habido nunca un acuerdo tan amplio sobre arquitectura de computadores.

Sin embargo, este estilo de arquitectura no puede permanecer estático. Una lección difícil es que el espacio de direcciones debe crecer, ya que el tamaño de 32 bits de todas estas arquitecturas se debe expandir para sobrevivir. En términos de su implementación, esperamos que todas ofrezcan ejecución superescalar de 2 a 4 instrucciones por ciclo. La tecnología del hardware irá más allá de los actuales CMOS VLSI y ECL a BiCMOS, y posiblemente a arseniuro de galio. Nuestra intuición es que todas crecerán más allá del mercado actual de las estaciones de trabajo y controladores de periféricos para llegar a minicomputadores, grandes computadores e, incluso, supercomputadores, con números crecientes de procesadores por clase de computador.

	IBM 360/370	Intel 8086	Motorola 68000	DEC VAX
Fecha anunciada	1964/1970	1978	1980	1977
Tamaño de instrucción (bits)	16, 32, 48	8, 16, 24, 32, 40, 48	16, 32, 48, 64, 80	8, 16, 24, 32,..., 432
Direccionamiento (tamaño, modelo)	24 bits, plano	4+16 bits, segmentado	24 bits, plano	32 bits, plano
¿Datos alineados?	Sí 360/No 370	No	16-bit alineado	No
Modos de direccionamiento de datos	4	5	9	$\geqslant 14$
Protección	Página	Ninguna	Opcional	Página
Tamaño de página	4 KB	—	0,25 a 32 KB	0,5 KB
E/S	Código de Op	Código de Op	Mapeadas en memoria	Mapeadas en memoria
Registros enteros (tamaño, modelo, número)	16 GPR x 32 bits	8 datos dedicados x 16 bits	8 datos & 8 direcciones x 32 bits	15 GPR x 32 bits
Registros separados en punto flotante	4 x 64 bits	Opcional: 8 x 80 bits	Opcional: 8 x 80 bits	0
Formato en punto flotante	IBM	IEEE 754 simple, doble, extendido	IEEE 754 simple, doble, extendido	DEC

FIGURA E.13 Resumen de cuatro arquitecturas de los años setenta. De forma distinta a las arquitecturas de la Figura E.1 (página 744), hay poco acuerdo entre estas arquitecturas en cualquier categoría. (Ver Capítulo 4 para más detalles sobre 370, 8086 y VAX.)

E.10 Referencias

INTEL [1989]. *i860 64-Bit Microprocessor Programmer's Reference Manual.*

KANE, G. [1988]. *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, N. J.

MOTOROLA [1988]. *MC88100 RISC Microprocessor User's Manual.*

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS AND D. ZURAS [1988]. «Integer multiplication and division on the HP Precision Architecture», *IEEE Trans. on Computers*, 37:8, 980-990.

MUCHNICK, S. S. [1988]. «Optimizing compilers for SPARC», *Sun Technology* (Summer) 1:3, 64-77.

SUN MICROSYSTEMS [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, August 25, 1989.

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON AND B. ZORN [1986]. «Evaluation of the SPUR LISP architecture», *Porc. 13th Symposium on Computer Architecture* (June), Tokyo.

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES AND D. PATTERSON [1984]. «Architecture of SOAR: Smalltalk on a RISC», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188-197.

Definiciones de arquitectura de computadores, trivialidades, fórmulas y reglas empíricas

Definiciones

CPI: ciclos de reloj por instrucción (pág. 38).

Frecuencia de aciertos: fracción de referencias a memoria encontradas en la cache, igual a 1 – Tasa de fallos (pág. 435).

Frecuencia de fallos: fracción de referencias a memoria no encontradas en la cache, igual a 1 – Tasa de aciertos (pág. 435).

Frecuencia de reloj: inverso de la duración del ciclo de reloj, habitualmente medida en MHz (pág. 38).

Gran Endian (Big Endian): el byte con la dirección binaria «x...x00» está en la posición más significativa («gran fin») de una palabra de 32 bits (pág. 101).

MIMD: (múltiples flujos de instrucciones, múltiples flujos de datos) un multiprocesador o multicomputador (pág. 616).

$N_{1/2}$: la longitud vectorial necesaria para alcanzar la mitad de R_∞ (pág. 412).

N_v: la longitud vectorial necesaria para que el modo vectorial sea más rápido que el modo escalar (pág. 412).

Localidad espacial: (localidad en el espacio) si se referencia un elemento, los elementos próximos tenderán a ser referenciados pronto (pág. 433).

Localidad temporal: (localidad en el tiempo) si un elemento es referenciado, tenderá a ser referenciado, de nuevo, pronto (pág. 433).

Penalización de fallo: tiempo para sustituir un bloque en el nivel superior de un sistema de cache con el correspondiente bloque del nivel inferior (pág. 435).

Pequeño Endian (Little Endian): el byte con la dirección binaria «x...x00» está en la posición menos significativa («pequeño fin») de una palabra de 32 bits (pág. 101).

R_∞: la velocidad en megaflops para vectores de longitud infinita (pág. 412).

Recuento de instrucciones: número de instrucciones ejecutadas mientras corre un programa (pág. 38).

Riesgo de datos RAW: (lectura después de escritura) la instrucción intenta leer un dato fuente antes que una instrucción anterior lo escriba, tomando así incorrectamente el valor antiguo (pág. 284).

Riesgo de datos WAR: (escritura después de lectura) la instrucción trata de escribir un destino antes de que sea leído por una instrucción anterior; por tanto, la instrucción anterior toma incorrectamente el nuevo valor (pág. 284).

Riesgo de datos WAW: (escritura después de escritura) la instrucción intenta escribir un operando antes de que sea escrito por una instrucción anterior. Las escrituras son realizadas en orden erróneo, dejando incorrectamente el valor de la instrucción anterior en el destino (pág. 284).

SIMD: (único flujo de instrucciones, múltiples flujos de datos) un procesador en array (pág. 616).

SISD: (único flujo de instrucciones, único flujo de datos) un uniprocesador (pág. 616).

Tiempo de acierto: tiempo de acceso a memoria para un acierto de cache, incluyendo el tiempo para determinar si hay fallo o acierto (pág. 435).

Trivialidades

Orden de byte de las máquinas (pág. 102)

Gran «Endian»: IBM 360, MIPS, Motorola, SPARC, DLX
Pequeño «Endian»: DEC VAX, DEC RISC, Intel 80x86
Año y tamaño de direcciones de usuario de Generaciones de las familias de Computadores IBM e Intel

Año	Modelo	Tamaño de direcciones de usuario	Año	Modelo	Tamaño de direcciones de usuario
1964	IBM 360	24	1978	Intel 8086	4+16
1971	IBM 370	24	1981	Intel 80186	4+16
1983	IBM 370-XA	31	1982	Intel 80286	16+16
1986	IBM ESA/370	16+31	1985	Intel 80386	16+32 o 32
			1989	Intel 80486	16+32 o 32

Fórmulas

1. *Aceleración de la segmentación* =

$$= \frac{\text{Tiempo de ciclo de reloj}_{\text{no segmentado}}}{\text{Tiempo de ciclo de reloj}_{\text{segmentado}}} \cdot \frac{\text{CPI ideal} \cdot \text{Profundidad de la segmentación}}{\text{CPI ideal} + \text{Ciclos de detención por instrucción}}$$

donde Ciclos de detención contabiliza los ciclos de reloj perdidos debidos a los riesgos de la segmentación (pág. 277).

2. *Coste de un circuito integrado* =

$$= \frac{\text{Coste del dado} + \text{Coste del test del dado} + \text{Coste del encapsulamiento}}{\text{Productividad de prueba final}}$$ (pág. 59)

3. *Ley de Amdahl*: aceleración =

$$= \frac{1}{(1 - \text{Fracción}_{\text{mejorada}}) + \frac{\text{Fracción}_{\text{mejorada}}}{\text{Aceleración}_{\text{mejorada}}}}$$ (pág. 9)

4. *Medias*—aritméticas(AM), aritmética ponderada(WAM), armónica(HM) y armónica ponderada(WHM):

$$\text{AM} = \frac{1}{n}\sum_{i=1}^{n}\text{Tiempo}_i,\ \text{WAM} = \sum_{i=1}^{n}\text{Peso}_i \cdot \text{Tiempo}_i,\ \text{HM} = \frac{n}{\sum_{i=1}^{n}\frac{1}{\text{Frecuencia}_i}},\ \text{VHM} = \frac{n}{\sum_{i=1}^{n}\frac{\text{Peso}_i}{\text{Frecuencia}_i}}$$

donde Tiempo_i es el tiempo de ejecución para el programa *i*ésimo de un total de *n* de la carga de trabajo. Peso_i es el peso del programa *i*ésimo de la carga de trabajo, y Frecuencia_i es una función de $1/\text{Tiempo}_i$ (pág. 54).

5. *Productividad del dado* =

$$= \text{Productividad de la oblea} \cdot \left\{1 + \frac{\text{Defectos por unidad de área} \cdot \text{Area del dado}}{\alpha}\right\}^{-\alpha}$$

donde Productividad de la oblea se refiere a las obleas que están tan mal que no necesitan ser examinadas y α corresponde al número de niveles de enmascaramiento críticos para la productividad del dado (habitualmente $\alpha \geqslant 2{,}0$, pág. 63).

6. *Rendimiento del sistema*:

$$\text{Tiempo}_{\text{carga trabajo}} = \frac{\text{Tiempo}_{\text{CPU}}}{\text{Velocidad}_{\text{CPU}}} + \frac{\text{Tiempo}_{\text{E/S}}}{\text{Velocidad}_{\text{E/S}}} - \frac{\text{Tiempo}_{\text{solapamiento}}}{\text{Máximo}(\text{Velocidad}_{\text{CPU}}, \text{Velocidad}_{\text{E/S}})}$$

donde $\text{Tiempo}_{\text{CPU}}$ significa el tiempo que la CPU está ocupada, $\text{Tiempo}_{\text{E/S}}$ significa el tiempo que el sistema de E/S está ocupado, y $\text{Tiempo}_{\text{solapamiento}}$ significa el tiempo que ambos están ocupados. Esta fórmula supone que el solapamiento escala linealmente con la aceleración (pág. 544).

7. *Tiempo de CPU* = Recuento de instrucciones · Ciclos de reloj por instrucción · Duración del ciclo del reloj (pág. 38).

8. *Tiempo medio de acceso a memoria* = Tiempo de acierto + Frecuencia de fallos · Penalización de fallos (pág. 435).

Reglas empíricas

1. *Regla de Amdahl/Case.* Un sistema computador equilibrado necesita aproximadamente 1 megabyte de capacidad de memoria principal y 1 megabit por segundo de ancho de banda de E/S por MIPS de rendimiento de CPU (pág. 19).
2. *Regla de localidad 90/10.* Un programa ejecuta aproximadamente el 90 por 100 de sus instrucciones en el 10 por 100 de su código (pág. 12).
3. *Regla de crecimiento de las DRAM.* La densidad incrementa aproximadamente el 60 por 100 por año, cuadruplicándose en tres años (pág. 17).
4. *Regla de crecimiento de los discos.* La densidad aumenta aproximadamente el 25 por 100 por año, duplicándose en tres años (pág. 18).
5. *Regla de consumo de direcciones.* La memoria necesaria por un programa medio crece aproximadamente en un factor de 1,5 a 2 por año; por tanto, consume entre 1/2 y 1 bit de dirección por año (pág. 17).
6. *Regla de saltos efectivos 90/50.* Aproximadamente, son efectivos el 90 por 100 de los saltos hacia atrás, mientras que, aproximadamente, son efectivos el 50 por 100 de los saltos hacia adelante (pág. 116).
7. *Regla cache 2:1.* La frecuencia de fallos de una cache de correspondencia directa de tamaño X es aproximadamente la misma que la de una cache asociativa por conjuntos de 2 vías de tamaño X/2 (pág. 453).

Notación de descripción hardware (y algunos operadores C estándares)

Notación	Significado	Ejemplo	Significado
←	Transferencia de datos. La longitud de la transferencia se da por la longitud del destino; la longitud se especifica cuando no es evidente.	`R1←R2;`	Transfiere el contenido de `R2` a `R1`. Los registros tienen una longitud fija, por tanto las transferencias más cortas que el tamaño del registro deben indicar los bits que se utilizan.
`M`	Array de memoria accedida en bytes. La dirección de comienzo para una transferencia se indica como el índice al array de memoria.	`R1←M[x];`	Coloca el contenido de la posición de memoria `x` en `R1`. Si una transferencia comienza en `M[i]` y requiere 4 bytes, los bytes transferidos son `M[i]`, `M[i+1]`, `M[i+2]` y `M[i+3]`.
$\leftarrow_n$	Transferir un campo de *n* bits se usa siempre que la longitud de la transferencia no es evidente.	$M[y] \leftarrow_{16} M[x];$	Transfiere 16 bits comenzando en la posición de memoria `x` a la posición de memoria `y`. Debe coincidir la longitud de las dos posiciones.
X_n	Subíndice selecciona un bit.	$R1_0 \leftarrow 0;$	Cambia el bit de signo de `R1` a 0. (Los bits están enumerados, comenzando en 0 el MSB.)
$X_{m..n}$	Subíndice selecciona un campo de bits.	$R3_{24..31} \leftarrow M[x];$	Transfiere el contenido de la posición de memoria `x` al byte de orden inferior de `R3`.
X^n	Superíndice replica un campo.	$R3_{0..23} \leftarrow 0^{24};$	Pone a 0 los tres bytes de orden superior de `R3`.
`##`	Concatena dos campos.	$R3 \leftarrow 0^{24}$ `##` `M[x]`	Transfiere al contenido de la posición `x` al byte inferior de `R3`; pone a cero los tres bytes superiores.
		$F2 \#\# F3 \leftarrow_{64} M[x];$	Transfiere 64 bits de memoria comenzando en la posición `x`; los 32 primeros bits van a `F2`; los 32 segundos a `F3`.
`*, &`	Desreferencia un puntero; obtiene la dirección de una variable.	`P*←&x;`	Asigna al objeto señalado por la dirección `p` de la variable `x`.
`<<, >>`	Desplazamientos lógicos de C (izquierda, derecha).	`R1 << 5`	Desplaza `R1` 5 bits a la izquierda.
`==, !=, >, <, >=, <=`	Operadores relacionales de C: igual, no igual, mayor, menor, mayor o igual, menor o igual	`(R1==R2) & (R3!=R4)`	Verdadero si el contenido de `R1` es igual al contenido de `R2` y el contenido de `R3` no es igual al contenido de `R4`.
`&, |, ^, !`	Operaciones lógicas bit a bit de C: and, or, or exclusiva y complementación.	`R1 & (R2 | R3)`	and bit-a-bit de `R1` con la or bit-a-bit de `R2` y `R3`.

Estructura de la segmentación del DLX

Etapa	Instrucción ALU	Instrucción de carga o almacenamiento	Instrucción de salto
IF	IR←Mem[PC]; PC←PC+4;	IR←Mem[PC]; PC←PC+4;	IR←Mem[PC] PC←PC+4;
ID	A←Rs1; B←Rs2; PC1←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR
EX	ALUoutput←A *op* B; o ALUoutput←A *op* $((IR1_{16})^{16}\#\#IR1_{16..31})$;	DMAR←A+ $((IR1_{16})^{16}\#\#IR1_{16..31})$; SMDR←B;	ALUoutput←PC1 + $((IR1_{16})^{16}\#\#IR1_{16..31})$; cond←(Rs1 *op* 0);
MEM	ALUoutput1←ALUoutput	LMDR←Mem[DMAR]; o Mem[DMAR]←SMDR;	if (cond) PC←ALUoutput;
WB	Rd←ALUoutput1;	Rd←LMDR;	

Repertorio de instrucciones del DLX, lenguaje de descripción y segmentación de DLX

Repertorio de instrucciones estándares de DLX

Tipo de instrucción/Código op.	Significado de la instrucción
Transferencias de datos	**Transfiere datos entre registros y memoria, o entre registros de enteros y registros de FP o registros especiales; sólo el modo de direccionamiento de memoria es un desplazamiento de 16 bits + el contenido de un registro de enteros**
LB, LBU, SB	Carga byte, carga byte sin signo, almacena byte
LH, LHU, SH	Carga media palabra, carga media palabra sin signo, almacena media palabra
LW, SW	Carga palabra, almacena palabra (a/desde registros de enteros)
LF, LD, SF, SD	Carga flotante SP, carga flotante DP, almacena flotante SP, almacena flotante DP
MOVI2S, MOVS2I	Transfiere desde/a registro entero a/desde un registro especial
MOVF, MOVD	Copia un registro de punto flotante o un par en DP en otro registro o par
MOVFP2I, MOVI2FP	Transfiere 32 bits desde/a registros de FP a/desde registros enteros
Aritmética, lógica	**Operaciones sobre datos enteros o lógicos en registros de enteros; las instrucciones aritméticas con signo causan un trap en caso de desbordamiento**
ADD, ADDI, ADDU, ADDUI	Suma, suma de inmediatos (todos los inmediatos son de 16 bits); con signo y sin signo

Tipo de instrucción/Código op.	Significado de la instrucción
SUB, SUBI, SUBU, SUBUI	Resta, resta de inmediatos; con signo y sin signo
MULT, MULTU, DIV, DIVU	Multiplicación y división, con signo y sin signo; los operandos deben ser registros de punto flotante; todas las operaciones toman valores de 32 bits
AND, ANDI	And, and de inmediatos
OR, ORI, XOR, XORI	Or, or de inmediatos, or exclusiva, or exclusiva de inmediatos
LHI	Cargo inmediato más significativo—carga la mitad superior del registro con inmediato
SLL, SRL, SRA, SLLI, SRLI, SRAI	Desplazamientos: ambos inmediatos (S__I) y forma variable (S__); los desplazamientos son lógicos a la izquierda, lógicos a la derecha, aritméticos a la derecha
S___, S___I	Inicialización (puesta a 1) condicional: «__» puede ser EQ, NE, LT, GT, LE, GE
Control	**Saltos y bifurcaciones condicionales; relativas al PC o a través de registros**
BEQZ, BNEZ	Salto si registro de enteros igual/no igual cero; desplazamiento de 16 bits desde el PC
BFPT, BFPF	Comprobación del Bit de comparación en el registro de estado FP y salto; desplazamiento de 16 bits desde el PC
J, JR	Bifurcaciones: desplazamiento de 26 bits desde el PC (J) (o destino) en registro (JR)
JAL, JALR	*Bifurcación y enlace:* almacena PC+4 en R31, el destino se desplaza 26 bits desde el PC (JAL) o desde un registro (JALR)
TRAP	Transfiere control al sistema operativo a una dirección vectorizada (ver Capítulo 5)
RFE	Vuelta al código de usuario desde una excepción; restaura modo usuario (ver Capítulo 5)
Punto flotante	**Operaciones de punto flotante en formatos DP y SP**
ADDD, ADDF	Suma números DP, SP
SUBD, SUBF	Resta números DP, SP
MULTD, MULTF	Multiplica punto flotante DP, SP
DIVD, DIVF	Divide punto flotante DP, SP
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convierte instrucciones: CVTx2y convierte del tipo x al tipo y, donde x e y pueden ser I (entero), D (doble precisión) o F (simple precisión); ambos operandos están en los registros FP
____D, ____F	Comparaciones DP y SP: «__» puede ser EQ, NE, LT, GT, LE, GE; modifica el bit de comparación en el registro de estado FP

Referencias

A continuación presentamos una recopilación de todas las referencias listadas en las secciones de referencias de cada capítulo. El número de la página donde aparece cada referencia en el libro se indica entre paréntesis al final de cada referencia.

ADAMS, T. AND R. ZIMMERMAN [1989]. «An analysis of 8086 instruction set usage in MS DOS programs», *Proc. Third Symposium on Architectural Support for Programming Languages and Systems* (April), Boston, 152-161. (p. 202)

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph. D. Thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May). (p. 526)

AGARWAL, A., R. L. SITES, AND M. HOROWITZ [1986]. «ATUM: A new technique for capturing address traces using microcode», *Proc. 13th Annual Symposium on Computer Architecture* (June 2-5), Tokyo, Japan, 119-127. (p. 525)

AGERWALA, T. AND J. COCKE [1987]. «High performance reduced instruction set processors», IBM Tech. Rep. (March). (p. 365)

ALEXANDER, W. G. AND D. B. WORTMAN [1975]. «Static and dynamic characteristics of XPL programs», *Computer* 8:11 (November) 41-46. (pp. 139, 200)

ALLIANT COMPUTER SYSTEMS CORP. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass. (p. 425)

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, Calif. (p. 635)

AMDAHL, G. M. [1967]. «Validity of the single processor approach to achieving large scale computing capabilities», *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N. J. (April) 483-485. (pp. 28, 634)

AMDAHL, G. M., G. A. BLAAUW , AND F. P. BROOKS, JR. [1964]. «Architecture of the IBM System/360», *IBM J. Research and Development* 8:2 (April) 87-101. (pp. 137, 200)

ANDERSON, D. W., F. J. SPARACIO, AND R. M. TOMASULO [1967]. «The IBM 360 Model 91: Machine philosophy and instruction handling», *IBM J. of Research and Development* 11:1 (January) 8-24. (p. 364)

ANDERSON, S. F., J. G. EARLE, R. E. GOLDSCHMIDT, AND D. M. POWERS [1967]. «The IBM System/360 Model 91: Floating-point execution unit», *IBM J. Research and Development* 11, 34-53. Reprinted in [Swartzlander, 1980]. (p. 702)

ANDREWS, G. R. AND F. B. SCHNEIDER [1983]. «Concept and notations for concurrent programming», *Computing Surveys* 15:1 (March) 3-43. (p. 635)

ANON ET AL. [1985]. «A measure of transaction processing power», Tanden Tech. Rep. TR 85.2. Also appeared in *Datamation*, April 1, 1985. (p. 550)

ARCHIBALD, J. AND J.-L. BAER [1986]. «Cache coherence protocols: Evaluation using a multiprocessor simulation model», *ACM Trans. on Computer Systems* 4:4 (November) 273-298. (p. 526)

ATANASOFF, J. V. [1940]. «Computing machine for the solution of large systems of linear equations», Internal Report, Iowa State University. (p. 26)

ATKINS, D. E. [1968]. «Higher-radix division using estimates of the divisor and partial remainders», *IEEE Trans. on Computers* C-17:10, 925-934. Reprinted in [Swartzlander 1980]. (p. 704).

BAER, J.-L. AND E.-H. WANG [1988]. «On the inclusion property for multi-level cache hierarchies», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, 73-80. (p. 526)

BAKOGLU, H. B., G. F. GROHOSKI, L. E. THATCHER, J. A. KAHLE, C. R. MOORE, D. P. TUTTLE, W. E. MAULE, W. R. HARDELL, D. A. HICKS, M. NGUYEN PHU, R. K. MONTOYE, W. T. GLOVER, AND S. DHAWAN [1989]. «IBM second-generation RISC machine organization», *Proc. Int'l Conf. on Computer Design, IEEE* (October) Rye, NY, 138-142 (p. 365)

BANERJEE, U. [1979]. *Speedup of Ordinary Programs,* Ph. D. Thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (October). (p. 424)

BARTON, R. S. [1961]. «A new approach to the functional design of a computer», *Proc. Western Joint Computer Conf.,* 393-396. (p. 137)

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM's Early Computers,* MIT Press, Cambridge, Mass. (p. 605)

BASHE, C. J., W. BUCHHOLZ, G. V. HAWKINS, J. L. INGRAM, AND N. ROCHESTER [1981]. «The architecture of IBM's early computers», *IBM J. of Research and Development* 25:5 (September) 363-375. (p. 605)

BATCHER, K. E. [1974]. «STARAN parallel processor system hardware», *Proc. AFIPS National Computer Conf.,* 405-410. (p. 635)

BELL, C. G. AND W. D. STRECKER [1976]. «Computer structures: What have we learned from the PDP-11?», *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 1-14. (p. 524)

BELL, C. G. [1984]. «The mini and micro industries», *IEEE Computer* 17:10 (October) 14-30. (p. 29)

BELL, C. G. [1985]. «Multis: A new class of multiprocessor computers», *Science* 228 (April 26) 462-467. (p. 634)

BELL, C. G. [1989]. «The future of high performance computers in science and engineering», *Comm. ACM* 32:9 (September) 1091-1101. (p. 636)

BELL, C. G. AND A. NEWELL [1971]. *Computer Structures: Readings and Examples,* McGraw-Hill, New York. (p. 702)

BELL, C. G., J. C. MUDGE, AND J. E. MCNAMARA [1978]. *A DEC View of Computer Engineering,* Digital Press, Bedford, Mass. (p. 85)

BELL, C. G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. «A new architecture for mini-computers: The DEC PDP-11», *Proc. AFIPS SJCC,* 657-675. (p. 137)

BERRY, M., D. CHEN, P. KOSS, D. KUCK [1988]. «The Perfect Club benchmarks: Effective performance evaluation of supercomputers», CSRD Report No. 827 (November), Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign. (p. 85)

BIRMAN, M., G. CHU, L. HU, J. MCLEOD, N. BEDARD, F. WARE, L. TORBAN, AND C. M. LIM [1988]. «Desing of a high-speed arithmetic datapath», *Proc. ICCD: VLSI Computers and Processor,* 214-216. (p. 696)

BLAKKEN, J. [1983]. «Register windows for SOAR», in *Smalltalk On a RISC: Architectural Investigations,* Proc. of CS 292R (April) 126-140. (p. 487)

BLOCH, E. [1959]. «The engineering design of the Stretch computer», *Proc. Fall Joint Computer Conf.,* 48-59. (p. 363)

BORRILL, P. L. [1986]. «32-bit buses—An objective comparison», *Proc. Buscon 1986 West,* San Jose, Calif., 138-145. (p. 573)

BOUKNIGHT, W. J., S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. «The Illiac IV system», *Proc. IEEE* 60:4, 369-379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), 306-316. (p. 614)

BRADY, J. T. [1986]. «A theory of productivity in the creative process», *IEEE CG & A* (May) 25-34. (p. 604)

BRENT, R. P. AND H. T. KUNG [1982]. «A regular layout for parallel adders», *IEEE Trans. on Computers* C-31, 260-264. (p. 702)

BRODERSEN, R. W. [1989]. «Evolution of VLSI signal-processing circuits», *Proc. Decennial Caltech Conf. on VLSI* (March) 43-46, The MIT Press, Pasadena, Calif. (p. 635)

BUCHER, I. Y. [1983]. «The computational speed of supercomputers», *Proc. SIGMETRICS Conf. on Measuring and Modeling of Computer Systems,* ACM (August) 151-165. (p. 424).

BUCHER, I. Y. AND A. H. HAYES [1980]. «I/O Performance measurement on Cray-1 and CDC 7000 computers», *Proc. Computer Performance Evaluation Users Group, 16th Meeting,* NBS 500-65, 245-254. (p. 606)

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch,* McGraw-Hill, New York. (p. 363)

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN [1946]. «Preliminary discussion of the logical design of an electronic computing instrument», Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann,* W. Aspray and A. Burks, eds., The MIT Press, Cambridge, Mass. and Tomash Publishers, Los Angeles, Calif., 1987, 97-146. (p. 26)

CALLAHAN, D., J. DONGARRA, AND D. LEVINE [1988]. «Vectorizing compilers: A test suite and results», *Supercomputing'88*, ACM/IEEE (November), Orlando, Fla., 98-105. (p. 405)

CASE, R. P. AND A. PADEGS [1978]. «The architecture of the IBM System/370», *Comm. ACM* 21:1, 73-96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830-855. (pp. 200, 523)

CENSIER, L. M. AND P. FEAUTRIER [1978]. «A new solution to the coherence problem in multicache systems», *IEEE Trans. on Computers* C-27:12 (December) 1112-1118. (p. 526)

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1982]. «Register allocation via coloring», *Computer Languages* 6, 47-57. (p. 140)

CHARLESWORTH, A. E. [1981]. «An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family», *Computer* 14:12 (December) 12-30. (p. 365)

CHEN, P. [1989]. *An Evaluation of Redundant Arrays of Inexpensive Disks Using an Amdahl 5890*, M. S. Thesis, Computer Science Division, Tech. Rep. UCB/CSD 89/506. (p. 546)

CHEN, S. [1983]. «Large-scale and high-speed multiprocessor system for scientific applications», *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., «Supercomputers: Design and applications», *IEEE* (August) 1984. (p. 423)

CHEN, T. C. [1980]. «Overlap and parallel processing», in *Introduction to Computer Architecture*, H. Stone, ed., Science Research Associates, Chicago, 427-486. (p. 364)

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph. D. Thesis, Stanford Univ. (December). (p. 140)

CHOW, F. C. AND J. L. HENNESSY [1984]. «Register allocation by priority-based coloring», *Proc. SIGPLAN'84 Compiler Construction* (*ACM SIGPLAN Notices* 19:6, June) 222-232. (p. 140)

CHOW, F., M. HIMELSTEIN, E. KILLIAN, AND L. WEBER [1986]. «Engineering a RISC compiler system», *Proc. COMPCON* (March), San Francisco, 132-137. (p. 210)

CLARK, D. W. [1983]. «Cache performance of the VAX-11/780», *ACM Trans. on Computer Systems* 1:1, 24-37. (p. 525)

CLARK, D. W. [1987]. «Pipelining and performance in the VAX 8800 processor», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 173-177. (p. 293)

CLARK, D. W. AND H. LEVY [1982]. «Measurement and analysis of instruction set use in the VAX-11/780», *Proc. Ninth Symposium on Computer Architecture* (April), Austin, Tex., 9-17. (p. 201)

CLARK, D. W. AND J. S. EMER [1985]. «Performance of the VAX-11/780 translation buffer: Simulation and measurement», *ACM Trans. on Computer Systems* 3:1, 31-62. (p. 525)

CLARK, D. W. AND W. D. STRECKER [1980]. «Comments on "the case for the reduced instruction set computer"», *Computer Architecture News* 8:6 (October) 34-38. (p. 140)

CLARK, D. W., P. J. BANNON, AND J. B. KELLER [1988]. «Measuring VAX 8800 performance with a histogram hardware monitor», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, Hawaii, 176-185. (pp. 228, 525)

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers*, Courant Institute, New York Univ., New York City. (p. 140)

COCKE, J. AND J. MARKSTEIN [1980]. «Measurement of code improvement algorithms», *Information Processing* 80, 221-228. (p. 140)

CODD, E. F. [1962]. «Multiprogramming», in F. L. Alt and M. Rubinoff, *Advances in Computers*, vol. 3, Academic Press, New York, 82. (p. 258)

CODY, W. J. [1988]. «Floating point standards: Theory and practice», in *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, R. E. Moore (ed.), Academic Press, Boston, Mass., 99-107. (p. 653)

CODY, W. J., J. T. COONEN, D. M. GAY, K. HANSON, D. HOUGH, W. KAHAN, R. KARPINSKI, J. PALMER, F. N. RIS, AND D. STEVENSON [1984]. «A proposed radix- and word-lenght-independent standard for floating-point arithmetic», *IEEE Micro* 4:4, 86-100. (p. 653)

COHEN, D. [1981]. «On holy wars and a plea for peace», *Computer* 14:10 (October) 48-54. (p. 102)

COLWELL, R. P., C. Y. HITCHCOCK, III, E. D. JENSEN, H. M. B. SPRUNT, AND C. P. KOLLAR [1985]. «Computers, complexity, and controversy», *Computer* 18:9 (September) 8-19. (p. 135).

COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND B. K. RODMAN [1987]. «A VLIW architecture for a trace scheduling compiler», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 180-192. (p. 365)

CONTI, C., D. H. GIBSON, AND S. H. PITKOWSKY [1968]. «Structural aspects of the System/360 Model 85, part I: General organization», *IBM Systems J.* 7:1, 2-14. (pp. 82, 525)

COONEN, J. [1984]. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, Ph. D. Thesis, Univ. of Calif., Berkeley. (p. 670)

CRAWFORD, J. H. AND P. P. GELSINGER [1987]. *Programming the 80386*, Sybex, Alameda, Calif. (pp. 202, 480)

CURNOW, H. J. AND B. A. WICHMANN [1976]. «A synthetic benchmark», *The Computer J.* 19:1. (p. 82)

DAVIDSON, E. S. [1971]. «The design and control of pipelined function generators», *Proc. Conf. on Systems, Networks, and Computers*, IEEE (January), Oaxtepec, Mexico, 19-21. (p. 364)

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. «Effective control for pipelined processors», *COMPCON, IEEE* (March), San Francisco, 181-184. (p. 364)

DEHNERT, J. C., P. Y.-T.-HSU, AND J. P. BRATT [1989]. «Overlapped loop support on the Cydra 5», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (April), IEEE/ACM, Boston, 26-39. (p. 366)

DEROSA, J., R. GLACKEMEYER, AND T. KNIGHT [1985]. «Design and implementation of the VAX 8600 pipeline», *Computer* 18:5 (May) 38-48. (p. 352)

DEWITT, D. J., R. FINKEL, AND M. SOLOMON [1984]. «The CRYSTAL multicomputer: Design and implementation experience», Computer Sciences Tech. Rep. No. 553, University of Wisconsin-Madison, September. (p. 635)

DIGITAL EQUIPMENT CORPORATION [1987]. *Digital Technical J.* 4 (March), Hudson, Mass. (This entire issue is devoted to the VAX 8800 processor.) (p. 367)

DITZEL, D. R. [1981]. «Reflections on the high-level language Symbol computer system», *Computer* 14:7 (July) 55-66. (p. 139)

DITZEL, D. R. AND D. A. PATTERSON [1980]. «Retrospective on high-level language computer architecture», in *Proc. Seventh Annual Symposium on Computer Architecture*, La Baule, France (June) 97-104. (p. 141)

DITZEL, D. R. AND H. R. MCLELLAN [1987]. «Branch folding in the CRISP microprocessor: Reducing the branch delay to zero», *Proc. 14th Symposium on Computer Architecture* (June), Pittsburgh, 2-7. (p. 365)

DITZEL, D. R., AND H. R. MCLELLAN [1982]. «Register allocation for free: The C machine stack cache», *Simposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 48-56. (p. 526)

DOHERTY, W. J. AND R. P. KELISKY [1979]. «Managing VM/CMS systems for user effectiveness», *IBM Systems J.* 18:1, 143-166. (p. 604)

DONGARRA. J. J. [1986]. «A survey of high performance computers», *COMPCON, IEEE* (March) 8-11. (p. 423)

EARLE, J. G. [1965]. «Latched carry-save adder», *IBM Technical Disclosure Bull.* 7 (March) 909-910. (p. 273)

EGGERS, S. [1989]. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*, Ph. D. Thesis, Univ. of California, Berkeley, Computer Science División Tech. Rep. UCB/CSD 89/501 (April). (p. 526)

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL. K. P. MCAULIFFE, L. RANDOLPH. M. SNIR, P. TELLER, AND J. WILSON [1985]. «Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach», *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 126-135. (p. 634)

ELLIS, J. R., J. A. FISHER, J. C. RUTTENBERG, AND NICHOLAU [1984]. «Parallel processing: A smart compiler and a dumb machine», *Proc. SIGPLAN Conf. on Compiler Construction* (June), Montreal, Canada, 37-47. (p. 365)

ELSHOFF, J. L. [1976]. «An analysis of some commercial PL/I programs», *IEEE Trans. on Sofware Engineering* SE-2 2 (June) 113-120. (p. 139)

EMER, J. S. AND D. W. CLARK [1984]. «A characterization of processor performance in the VAX-11/780», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301-310. (pp. 203, 228, 367, 525)

E·SUN MICROSYSTEMS [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, August 25, 1989.

FABRY, R. S. [1974]. «Capability based addressing», *Comm. ACM* 17:7 (July) 403-412. (p. 523)

FAZIO, D. [1987]. «It's really much more fun building a supercomputer than it is simply inventing one», *COMPCON, IEEE* (February) 102-105. (p. 423)

FEIERBACK, G. AND D. STEVENSON [1979]. «The Illiac-IV» in *Infotech State of the Art Report on Supercomputers*, Maidenhead, England. This data also appears in D. P. Siewiorek, C. G. Bell,

and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 268-269. (p. 599)

FISHER, J. A. [1983]. «Very long instruction word architectures and ELI-512», *Proc. Temth Symposium on Computer Architecture* (June), Stockholm, Sweden. (p. 365)

FLEMMING, P. J. AND J. J. WALLACE [1986]. «How not to lie with statistics: The correct way to summarize benchmarks results», *Comm. ACM* 29:3 (March) 218-221. (p. 83)

FLYNN, M. J. [1966]. «Very high-speed computing systems», *Proc. IEEE* 54:12 (December) 1901-1909. (pp. 377, 636)

FOLEY, J. D. AND A. VAN DAM [1982]. *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass. (p. 605)

FOSTER, C. C. AND E. M. RISEMAN [1972]. «Percolation of code to enhance parallel dispatching and execution», *IEEE Trans. on Computers* C-21:12 (December) 1411:1415. (p. 365)

FOSTER, C. C., R. H. GONTER, AND E. M. RISEMAN [1971]. «Measures of opcode utilization», *IEEE Trans. on Computers* 13:5 (May) 582-584. (p. 139)

FRANK, P. D. [1987]. «Advances in Head Technology», presentation at *Challenges in Winchester Technology* (December 15), Santa Clara Univ. (p. 605)

FRANK, S. J. [1984]. «Tightly coupled multiprocessor systems speed memory access times», *Electronics* 57:1 (January) 164-169. (p. 526)

FREIMAN, C. V. [1961]. «Statistical analysis of certain binary division algorithms», *Proc. IRE* 49:1, 91-103. (p. 702)

FRIESENBORG, S. E. AND R. J. WICKS [1985]. «DASD expectations: The 3380, 3380-23, and MVS/XA», Thec. Bulletin GG22-9363-02 (July 10), Washington Systems Center. (p. 596)

FULLER, S. H. [1976]. «Price/performance comparison of C.mmp and the PDP-11», *Proc. Third Annual Symposium on Computer Architecture* (Texas, January 19-21), 197-202. (p. 85)

FULLER, S. H. AND W. E. BURR [1977]. «Measurement and evaluation of alternative computer architectures», *Computer* 10:10 (October) 24-35. (p. 84)

GAGLIARDI, U. O. [1973]. «Report of workshop 4—software-related advances in computer hardware», *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99-120. (p. 139)

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. «CEDAR—A large scale multiprocessor», *Proc Int'l Conf. on Parallel Processing* (August) 524-529. (p. 634)

GARNER, R., A. AGARWAL, F. BRIGGS, E. BROWN, D. HOUGH, B. JOY, S. KLEIMAN, S. MUNCHNIK, M. NAMJOO, D. PATTERSON, J. PENDLETON, AND R. TUCK [1988]. «Scaleable processor architecture (SPARC)», *COMPCON, IEEE* (March), San Francisco, 278-283. (p. 203)

GEHRINGER, E. F., D. P. SIEWIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass. (p. 636)

GIBSON, D. H. [1967]. «Considerations in block-oriented systems design», *AFIPS Conf. Proc.* 30, SJCC, 75-80. (p. 525)

GIBSON, J. C. [1970]. «The Gibson mix», Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N. Y. (Research done in 1959.) (p. 82)

GOLDBERG, D. [1989]. «Floating-point and computer systems», *Xerox Tech. Rep.* CSL-89-9. A version of this paper will appear in *Computing Surveys*. (p. 670)

GOLDBERG, I. B. [1967]. «27 bits are not enough for 8-digit accuracy», *Comm. ACM* 10:2, 105-106. (p. 703)

GOLDSTEIN, S. [1987]. «Storage performance—an eight year outlook», Tech. Rep. TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif. (p. 605)

GOLDSTINE, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, N. J. (p. 27)

GOODMAN, J. R. [1983]. «Using cache memory to reduce processor memory traffic», *Proc. Tenth Annual Symposium on Computer Architecture* (June 5-7), Stockholm, Sweden, 124-131. (p. 526)

GOODMAN, J. R. [1987]. «Coherency for multiprocessor virtual address caches», *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 71-81. (p. 525)

GOODMAN, J. R. AND M.-C. CHIANG [1984]. «The use of static column RAM as a memory hierarchy», *Proc. 11th Annual Symposium on Computer Architecture* (June 5-7), Ann Arbor. Mich., 167-174. (p. 525)

GOSLING, J. B. [1980]. *Design of Arithmetic Units for Digital Computers*, Springer-Verlag New York, Inc., New York. (p. 641)

GRAY, W. P. [1989]. Memorandum of Decision, No. C-84-20799-WPG, U.S. District Court for the Northen District of California (February 7, 1989). (p. 261)

GROSS, T. R. [1983]. *Code Optimization of Pipeline Constraints*, Ph. D. Thesis (December), Computer Systems Lab., Stanford Univ. (p. 364)

HALBERT. D. C. AND P. B. KESSLER [1980]. «Windows of overlapping register frames», *CS 292R Final Reports* (June) 82-100. (p. 487)

HAMACHER, V. C., Z. G. VRANESIC, AND S. G. ZAKY [1984]. *Computer Organization*, 2nd ed., McGraw-Hill, New York. (p. 641)

HAUCK, E. A., AND B. A. DENT [1968]. «Burroughs' B6500/B7500 stack mechanism», *Proc. AFIPS SJCC*, 245-251. (p. 137)

HENLY, M. AND B. MCNUTT [1989]. «DASD I/O characteristics: A comparison of MVS to VM», Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif. (pp. 83, 606)

HENNESSY, J. [1985]. «VLSI RISC processors», *VLSI Systems Design* VI:10 (October) 22-32. (p. 203)

HENNESSY, J. L. AND T. R. GROSS [1983]. «Postpass code optimization of pipeline constraints», *ACM Trans. on Programming Languages and Systems* 5:3 (July) 422-448. (p. 364)

HENNESSY, J. L., N. JOUPPI, F. BASKETT, T. R. GROSSS, AND J. GILL [1982]. «Hardware/software tradeoffs for increased performance», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), 2-11. (p. 140)

HENNESSY, J., N. JOUPPI, F. BASKETT, AND J. GILL [1981]. «MIPS: A VLSI processor architecture», *Proc. CMU Conf. on VLSI Systems and Computations* (October), Computer Science Press, Rockville, Md. (p. 203)

HENNESSY, J. [1984]. «VLSI processor architecture», IEEE *Trans. on Computers* C-33:11 (December) 1221-1246. (p. 203)

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance*, Ph. D. Thesis, Univ. of California at Berkeley Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November. (p. 526)

HILL, M. D. [1988]. «A case for direct mapped caches», *Computer* 21:12 (December) 25-40. (p. 519)

HILLIS, W. D. [1985]. *The Connection Machine*, The MIT Press, Cambridge, Mass. (p. 634)

HINTZ, R. G. AND D. P. TATE [1972]. «Control data STAR-100 processor design», *COMPCON, IEEE* (September) 1-4. (p. 422)

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms*, Adam Hilger Ltd., Bristol, England and Philadelphia. (p. 635)

HOLLAND, J. H. [1959]. «A universal computer capable of executing and arbitrary number of subprograms simultaneously», *Proc. East Joint Computer Conf.* 16, 108-113. (p. 634)

HOLLINGSWORTH, W., H. SACHS AND A. J. SMITH [1989]. «The Clipper processor: Instruction set architecture and implementation», *Comm. ACM* 32:2 (February), 200-219. (p. 76)

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md. (p. 634)

HOWARD, J. H. ET AL. [1988]. «Scale and performance in a distributed file system», *ACM Trans. on Computer Systems* 6:1, 51-81. (p. 551)

HUGUET, M. AND T. LANG [1985]. «A reduced register file for RISC architectures», *Computer Architecture News* 13:4 (September) 22-31. (p. 527)

HWANG, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York. (p. 704)

HWU, W.-M. AND Y. PATT [1986]. «HPSm, a high performance restricted data flow architecture having minimum functionality», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 297-307. (p. 364)

IBM [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0 White Plains, N. Y., 11-82. (p. 604)

IEEE [1985]. «IEEE standard for binary floating-point arithmetic», *SIGPLAN Notices* 22:2, 9-25. (p. 553)

IMPRIMIS [1989]. «Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB», Document No. 64402302 (May). (p. 601)

INTEL [1989]. *i860 64-Bit Microprocessor Programmer's Reference Manual.* (768)

JORDAN, K. E. [1987]. «Performance comparison of large-scale scientific computers: Scalar mainframes, mainframes with vector facilities, and supercomputers», *Computer* 20:3 (March) 10-23. (p. 424)

JOUPPI, N. P. AND D. W. WALL [1989]. «Avalilabel instruction-level parallelism for superscalar and superpipelined machines», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272-282. (p. 366)

KAHAN, W. [1968]. «7094-II system support for numerical analysis», *SHARE Secretarial Distribution* SSD-159. (p. 703)

KAHANER, D. K. [1988]. «Benchmarks for "real" programs», *SIAM News* (November). (p. 700)

KAHN, R. E. [1972]. «Resource-sharing computer communication networks», *Proc. IEEE* 60:11 (November) 1397-1407. (p. 605)

KANE, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N. J. (p. 204)

KANE, G. [1988]. *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, N. J. (p. 768)

KATZ, R. H., D. A. PATTERSON, AND G. A. GIBSON [1990]. «Disk system architectures for high performance computing», *Proc. IEEE* 78:2 (February). (p. 605)

KATZ, R. H., S. EGGERS, D. A. WOOD, C. PERKINS, AND R. G. SHELDON [1985]. «Implementing a cache consistency protocol», *Proc. 12th Symposium on Computer Architecture*, 276-283. (p. 526)

KELLER, R. M. [1975]. «Look-ahead processors», *ACM Computing Surveys* 7:4 (December) 177-195. (p. 364)

KELLY, E. [1988]. «"SCRAM Cache" in Sun-4/110 beats traditional caches», *Sun Technology* 1:3 (Summer) 19-21. (p. 525)

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, F. H. SUMNER [1962]. «One-level storage system», *IRE Transactions on Electronic Computers* EC-11 (April) 223-235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135-148. (pp. 29)

KIM, M. Y. [1986]. «Synchronized disk interleaving», *IEEE Trans. on Computers* C-35:11 (November). (p. 605)

KNUTH, D. [1981]. *The Art of Computer Programming*, vol. II, 2nd ed., Addison-Wesley, Reading, Mass. (p. 705)

KNUTH, D. E. [1971]. «An empirical study of FORTRAN programs», *Software Practice and Experience*, vol. 1, 105-133. (p. 29)

KOGGE, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York. (pp. 688)

KOHN, L. AND S.-W. FU [1989]. «A 1,000,000 transistor microprocessor», *IEEE Int'l Solid-State Circuits Conf.*, 54-55. (p. 661)

KROFT, D. [1981]. «Lockup-free instruction fetch/prefetch cache organization», *Proc. Eighth Annual Symposium on Computer Architecture* (May 12-14), Minneapolis, Minn., 81-87. (p. 526)

KUCK, D., P. P. BUDNIK, S.-C. CHEN, D. H. LAWRIE, R. A. TOWLE, R. E. STREBENDT, E. W. DAVIS, JR, J. HAN, P. W. KRASKA, Y. MURAOKA [1974]. «Measurements of parallelism in ordinary FORTRAN programs», *Computer* 7:1 (January) 37-46. (p. 424)

KUHN, R. H. AND D. A. PADUA, EDS. [1981]. *Tutorial on Parallel Processing*, IEEE. (p. 635)

KUNG, H. T. [1982]. «Why systolic architectures?», *IEEE Computer* 15:1, 37-46. (p. 635)

KUNKEL, S. R. AND J. E. SMITH [1986]. «Optimal pipelining in supercomputers», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404-414. (p. 364)

LAM, M. [1988]. «Software pipeling: An effective scheduling technique for VLIW machines», *SIGPLAN Conf. on Programming Language Design and Implementation*, ACM (June), Atlanta, Ga., 318-328. (p. 365)

LAMPSON, B. W. [1982]. «Fast procedure calls», *Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 66-75. (p. 526)

LARSON, JUDGE E. R. [1973]. «Findings of Fact, Conclusions of Law, and Order for Judgment», File No. 4-67, Civ. 138, *Honeywell v. Sperry Rand and Illinois Scientific Development*, U. S. District Court for the District of Minnesota, Fourth Division (October 19). (p. 26)

LEE, R. [1989]. «Precision architecture», *Computer* 22:1 (January) 78-91. (p. 204).

LEINER, A. L. [1954]. «System specifications for the DYSEAC», *J. ACM* 1:2 (April) 57-81. (p. 605)

LEINER, A. L. AND S. N. ALEXANDER [1954]. «System organization of the DYSEAC», *IRE Trans. of Electronic Computers* EC-3:1 (March) 1-10. (p. 605)

LEVY, H. M. AND R. H. ECKHOUSE, JR. [1989]. *Computer Programming and Architecture: The VAX*, 2nd ed., Digital Press, Bedford, Mass. 358-372. (pp. 201, 260)

LEVY, J. V. [1978]. «Buses: The skeleton of computer structures», in *Computer Engineering: A DEC View of Hardware Systems Design*, C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Digital Press Bedford, Mass. (p. 605)

LINCOLN, N. R. [1982]. «Technology and design tradeoffs in the creation of a modern supercomputer», *IEEE Trans. on Computers* C-31:5 (May) 363-376. (p. 422)

LIPOVSKI, A. G. AND A. TRIPATHI [1977]. «A reconfigurable varistructure array processor», *Proc. 1977 Int'l Conf. of Parallel Processing* (August), 165-174. (p. 635)

LIPTAY, J. S. [1968]. «Structural aspects of the System/360 Model 85, part II: The cache», *IBM Systems J.* 7:1, 15-21. (p. 525)

LOVETT, T. AND S. THAKKAR [1988]. «The Symmetry multiprocessor system», *Proc. 1988 Int'l Conf. of Parallel Processing*, University Park, Pennsylvania, 303-310. (p. 634)

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. «A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2», *Computer* 18:12 (December) 10-24. (pp. 80, 424)

LUNDE, A. [1977]. «Empirical evaluation of some features of instruction set processor architecture», *Comm. ACM* 20:3 (March) 143-152. (p. 139)

MABERLY, N. C. [1966]. *Mastering Speed Reading*, New American Library, Inc., New York . (p. 553)

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS AND D. ZURAS [1988]. «Integer multiplication adn division on the HP Precision Architecture», *IEEE Trans. on Computers*, 37:8, 980-990. (p. 752)

MAGENHEIMER, D. J., L. PETERS, K. W. PETTIS, AND D. ZURAS [1988]. «Integer multiplication and division on the HP Precision Architecture», *IEEE Trans. on Computers* 37:8, 980-990. (p. 652)

MCCALL, K. [1983]. «The Smalltalk-80 benchmarks», *Smalltalk 80: Bits of History, Words of Advice*, G. Krasner, ed., Addison-Wesley, Reading, Mass., 153-174. (p. 487)

MCCREIGHT, E. [1984]. «The Dragon computer system: An early overview», Thech. Rep. Xerox Corp. (September). (p. 526)

MCFARLING, S. [1989]. «Program optimization for instruction caches», *Proc. Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (April 3-6), Boston, Mass., 183-191. (p. 535)

MCFARLING, S. AND J. HENNESSY [1986]. «Reducing the cost of branches», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396-403. (p. 365)

MCKEEMAN, W. M. [1967]. «Language directed computer design», *Proc. 1967 Fall Joint Computer Conf.*, Washington, D. C., 413-417. (p. 138)

MCKEVITT, J., ET AL. [1977]. *8086 Design Report*, internal memorandum. (p. 244)

MCMAHON, F. M. [1986]. «The Livermore FORTRAN kernels: A computer test of numerical performance range», Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, Calif. (December). (p. 83)

MEAD, C. AND L. CONWAY [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (p. 702)

MENABREA, L. F. [1842]. «Sketch of the analytical engine invented by Charles Babbage», Bibiothèque Universelle de Genève (October). (p. 634)

METCALFE, R. M. AND D. R. BOGGS [1976]. «Ethernet: Distributed packet switching for local computer networks», *Comm. ACM* 19:7 (July) 395-404. (p. 603)

MEYERS, G. J. [1978]. «The evaluation of expressions in a storage-to-storage architecture», *Computer Architecture News* 7:3 (October), 20-23. (p. 136)

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley. N. Y. . (p. 138)

MIRANKER, G. S., J. RUBENSTEIN, AND J. SANGUINETTI [1988]. «Squeezing a Cray-class supercomputer into a single-user package», *COMPCON, IEEE* (March) 452-456. (p. 424)

MITCHELL, D. [1989]. «The Transputer: The time is now», *Computer Design*, RISC supplement, 40-41 (November). (p. 614)

MIURA, K. AND K. UCHIDA [1983]. «FACOM vector processing system: VP100/200», *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K, Hwang, ed., «Supercomputers: Design and applications», *IEEE* (August 1984) 59-73. (p. 423)

MOORE, B., A. PADEGS, R. SMITH, AND W. BUCHOLZ [1987]. «Concepts of the System/370 vector architecture», *Proc. 14th Symposium on Computer Architecture* (June), ACM/IEEE, Pittsburgh, Pa., 282-292. (p. 425)

MORSE, S., B. RAVENAL, S. MAZOR, AND W. POHLMAN [1980]. «Intel Microprocessors—8008 to 8086», *Computer* 13:10 (October). (p. 201)

MOTOROLA [1988]. *MC88100 RISC Microprocessor User's Manual.* (p. 763)

MOUSSOURIS, J., L. CRUDELE, D. FREITAS, C. HANSEN, E. HUDSON, S. PRZYBYLSKI, T. RIORDAN, AND C. ROWEN [1986]. «A CMOS RISC processor with integrated system functions», *Proc. COMPCON, IEEE* (March), San Francisco. (p. 203)

MUCHNICK, S. S. [1988]. «Optimizing compilers for SPARC», *Sun Technology* (Summer) 1:3, 64-77. (p. 752)

NEWMAN, W. N. AND R. F. SPROULL [1979]. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York. (p. 605)

NGAI, T.-F. AND M. J. IRWIN [1985]. «Regular, area-time efficient carry-lookahead adders», *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 9-15. (p. 702)

NICHOLAU, A. AND J. A. FISHER [1984]. «Measuring the parallelism available for very long instruction word architectures», *IEEE Trans. on Computers* C-33:11 (November) 968-976. (p. 365)

OUSTERHOUT, J. K. ET AL. [1985]. «A trace-driven analysis of the UNIX 4.2 BSD file system», *Proc. Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, Wash., 15-24. (p. 580)

PADUA, D. AND M. WOLFE [1986]. «Advanced compiler optimizations for supercomputers», *Comm. ACM* 29:12 (December) 1184-1201. (p. 424)

PAPAMARCOS, M. AND J. PATEL [1984]. «A low coherence solution for multiprocessors with private cache memories», *Proc. of the 11th Annual Symposium on Computer Architecture* (June), Ann Arbor, Mich., 348-354. (p. 526)

PATTERSON, D. A. [1983]. «Microprogramming», *Scientific American* 248:3 (March), 36-43. (p. 261)

PATTERSON, D. A. [1985]. «Reduced Instruction Set Computers», *Comm. ACM* 28:1 (January) 8-21. (p. 203)

PATTERSON, D. A. AND C. H. SEQUIN [1981]. «Lockup-free instruction fetch/prefetch cache organization», *Proc. Eighth Annual Symposium on Computer Architecture* (May 12-14), Minneapolis, Minn., 443-458. (p. 526)

PATTERSON, D. A. AND D. R. DITZEL [1980]. «The case for the reduced instruction set computer», *Computer Architecture News* 8:6 (October), 25-33. (pp. 140, 202)

PATTERSON, D. A., G. A. GIBSON, AND R. H. KATZ [1987]. «A case for redundant arrays of inexpensive disks (RAID)», Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.*, Chicago, Illinois, June 1-3, 1988, 109-116. (p. 605)

PENG, V., S. SAMUDRALA, AND M. GAVRIELOV [1987]. «On the implementation of shifters, multipliers, and dividers in VLSI floating point units», *Proc. Eighth IEEE Symposium on Computer Arithmetic*, 95-102. (p. 705)

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. «The IBM research parallel processor prototype (RP3): Introduction and architecture», *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, Mass., 764-771. (p. 634)

PHISTER, M., JR. [1979]. *Data Processing Technology and Economics*, 2nd ed., Digital Press and Santa Monica Publishing Company. (p. 85)

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Mateo, Calif. (p. 526)

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. «Performance tradeoffs in cache design», *Proc. 15th Annual Symposium on Computer Architecture* (May-June), Honolulu, Hawaii, 290-298. (p. 520)

RADIN, G. [1982]. «The 801 minicomputer», *Proc. Symposium Architectural Support for Programming Languages and Operating Systems* (March), Palo Alto, Calif. 39-47. (p. 202)

RAMAMOORTHY, C. V. AND H. F. LI [1977]. «Pipeline architecture», *ACM Computing Surveys* 9:1 (March) 61-102. (p. 364)

REDMOND, K. C. AND T. M. SMITH [1980]. *Project Whirlwind—The History of a Pioneer Computer*, Digital Press, Boston, Mass. (p. 26)

REIGEL, E. W., U. FABER, AND D. A. FISCHER [1972]. «The Interpreter—a microprogrammable building block system», *Proc. AFIPS 1972 Spring Joint Computer Conf.* 40, 705-723. (p. 262)

ROBERTS, D., G. TAYLOR, AND T. LAYMAN [1990]. «An ECL RISC microprocessor designed for two-level cache», *IEEE COMPCON* (February). (p. 525)

ROBINSON, B. AND L. BLOUNT [1986]. «The VM/HPO 3880-23 performance results», IBM Tech. Bulletin, GG66-0247-00 (April), Washington Systems Center, Gathersburg, Md. (p. 596)

ROWEN, C., M. JOHNSON, AND P. RIES [1988]. «The MIPS R3010 floating-point coprocessor», *IEEE Micro* 53-62 (June). (p. 696)

RUSSELL, R. M. [1978]. «The CRAY-1 computer system», *Comm. ACM* 21:1 (January) 63-72. (pp. 393, 635)

RYMARCZYK, J. [1982]. «Coding guidelines for pipelined processors», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 12-19. (p. 364)

SALEM, K. AND H. GARCIA-MOLINA [1986]. «Disk striping», *IEEE 1986 Int'l Conf. on Data Engineering*. (p. 605)

SAMPLES, A. D. AND P. N. HILFINGER [1988]. «Code reorganization for instruction caches», Tech. Rep. UCB/CSD 88/447 (October), Univ. of Calif., Berkeley. (p. 535)

SANTORO, M. R., G. BEWICK, AND M. A. HOROWITZ [1989]. «Rounding algorithms for IEEE multipliers», *Proc. Ninth IEEE Symposium on Computer Arithmetic*, 176-183. (p. 662)

SCHNECK, P. B. [1987]. *Supercomputer Architecture*, Kluwer Academic Publishers, Norwell, Mass. (p. 423)

SCOTT, N. R. [1985]. *Computer Number Systems and Arithmetic*, Prentice-Hall, Englewood Cliffs, N. J. (p. 641)

SCRANTON, R. A., D. A. THOMPSON, AND D. W. HUNTER [1983]. «The access time myth», Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N. Y. (p. 605)

SEITZ, C. [1985]. «The Cosmic Cube», *Comm. ACM* 28:1 (January) 22-31. (p. 635)

SHURKIN, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York. (p. 27)

SHUSTEK, L. J. [1978]. «Analysis and performance of computer instruction sets», Ph. d. Thesis (May), Stanford Univ., Stanford, Calif. (p. 200)

SITES, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego. (p. 364)

SITES, R. L. [1979]. «How to use 1000 registers», *Caltech Conf. on VLSI* (January). (p. 526)

SLATER, R. [1987]. *Portraits in Silicon*, The MIT Press, Cambridge, Mass. (p. 27)

SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. «The Solomon computer», *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97-107. (p. 634)

SMITH, A. AND J. LEE [1984]. «Branch prediction strategies and branch target buffer design», *Computer* 17:1 (January) 6-22. (p. 365)

SMITH, A. J. [1982]. «Cache memories», *Computing Surveys* 14:3 (September) 473-530. (p. 525)

SMITH, A. J. [1985]. «Disk cache—miss ratio analysis and design considerations», *ACM Trans. on Computer Systems* 3:3 (August) 161-203. (p. 580)

SMITH, A. J. [1986]. «Bibliography and readings on CPU cache memories and related topics», *Computer Architecture News* (January) 22-42. (p. 525)

SMITH, B. J. [1981]. «Architecture and applications of the HEP multiprocessor system», *Real-Time Signal Processing IV* 298 (August) 241-248. (p. 424)

SMITH, J. E. [1981]. «A study of branch prediction strategies», *Proc. Eighth Symposium on Computer Architecture* (May), Minneapolis, 135-148. (p. 365)

SMITH, J. E. [1984]. «Decoupled access/execute computer architectures», *ACM Trans. on Computer Systems* 2:4 (November), 289-308. (p. 365)

SMITH, J. E. [1988]. «Characterizing computer performance with a single number», *Comm. ACM* 31:10 (October 1202-1206. (p. 83)

SMITH, J. E. [1989]. «Dynamic instruction scheduling and the Astronautics ZS-1», *Computer* 22:7 (July) 21-35. (p. 366)

SMITH, J. E. AND A. R. PLEZKUN [1988]. «Implementing precise interrupts in pipelined processors», *IEEE Trans. on Computers* 37:5 (May) 562-573. (p. 364)

SMITH, J. E. AND J. R. GOODMAN [1983]. «A study of instruction cache organizations and replacement policies». *Proc. Tenth Annual Symposium on Computer Architecture* (June 5-7), Stockholm, Sweden, 132-137. (p. 528)

SMITH, J. E., G. E. DERMER, B. D. VANDERWARN, S. D. KLINGER, C. M. ROZEWSKI, D. L. FOWLER, K. R. SCIDMORE, J. P. LAUDON [1987]. «The ZS-1 central processor», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto Calif., 199-204. (p. 366)

SMITH, M. D., M. JOHNSON, AND M. A. HOROWITZ [1989]. «Limits on multiple instruction issue», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, Mass., 290-302. (p. 366)

SMITH, W. R., R. R. RICE, G. D. CHESLEY, T. A. LALIOTIS, S. F. LUNDSTROM, M. A. CHALHOUN, L. D. GEROULD, AND T. C. COOK [1971]. «SYMBOL: A large experimental system exploring major hardware replacement of software», *Proc. AFIPS Spring Joint Computer Conf.*, 601-616. (p. 138)

SMOTHERMAN, M. [1989]. «A sequencing-based taxonomy of I/O systems and review of historical machines», *Computer Architecture News* 17:5 (September) 5-15. (pp. 258, 605)

SOHI, G. S., AND S. VAJAPEYAM [1989]. «Tradeoffs in instruction format design for horizontal architectures», *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, Mass. 15-25. (p. 366)

SPEC [1989]. «SPEC Benchmark Suite Release 1.0», October 2, 1989. (p. 51)

SPORER, M., F. H. MOSS AND C. J. MATHAIS [1988]. «An introduction to the architecture of the Stellar Graphics supercomputer», *COMPCON, IEEE* (March) 464-467. (p. 424)

STERN, N. [1980]. «Who invented the first electronic digital computer», *Annals of the History of Computing* 2:4 (October) 375-376. (p. 26)

STRAPPER, C. H. [1989]. «Fact and fiction in yield modelling», Special Issue of the *Microelectronics Journal* entitled *Microelectronics into the Nineties*, Oxford, UK; Elsevier (May). (p. 85)

STRAPPER, C. H., F. H. ARMSTRONG, AND K. SAJI [1983]. «Integrated circuit yield statistics», *Proc. IEEE* 71:4 (April) 453-470. (p. 85)

STRECKER, W. D. [1976]. «Cache memories for the PDP-11?», *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, Penn., 155-158. (pp. 200, 486)

STRECKER, W. D. [1978]. «VAX-11/780: A virtual address extension to the PDP-11 family», *Proc. AFIPS National Computer Conf.* 47, 967-980. (138, 200)

STRECKER, W. D. AND C. G. BELL [1976]. «Computer structures: What have we learned from the PDP-11?», *Proc. Third Symposium on Computer Architecture*. (p. 200)

SUTHERLAND, I. E. [1963]. «Sketchpad: A man-machine graphical communication system», *Spring Joint Computer Conf.* 329. (p. 605)

SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. «The implementation of the Cm* multi-microprocessor», *Proc. AFIPS National Computing Conf.*, 645-654. (p. 634)

SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. «Cm*—A modular, multimicroprocessor», *Proc. AFIPS National Computer Conf.* 46, 637-644. (p. 635)

SWARTZ, J. T. [1980]. «Ultracomputers», *ACM Transactions on Programming Languages and Systems* 4:2, 484-521. (p. 637)

SWARTZLANDER, E., ED. [1980]. *Computer Arithmetic*, Dowden, Hutchison and Ross (distributed by Van Nostrand, New York). (p. 702)

TAKAGI, N., H. YASUURA, AND S. YAJIMA [1985]. «High-speed VLSI multiplication algorithm with a redundant binary addition tree», *IEEE Trans. on Computers* C-34:9, 789-796. (p. 702)

TANENBAUM, A. S. [1978]. «Implications of structured programming for machine architecture», *Comm. ACM* 21:3 (March) 237-246. (p. 138)

TANG, C. K. [1976]. «Cache system design in the tightly coupled multiprocessor system», *Proc. 1976 AFIPS National Computer Conf.*, 749-753. (p. 525)

TAYLOR, G. S. [1981]. «Compatible hardware for division and square root», *Proc. Fifth IEEE Symposium on Computer Arithmetic*, 127-134. (p. 705)

TAYLOR, G. S. [1985]. «Radix 16 SRT dividers with overlapped quotient selection stages», *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 64-71. (p. 697)

TAYLOR, G. S., P. N. HILFINGER, J. R. LARUS, D. A. PATTERSON, AND B. G. ZORN [1987]. «Evaluation of the SPUR Lisp architecture», *Proc. 13th Annual Symposium on Computer Architecture* (June 2-5), Tokyo, Japan, 444-452. (pp. 203, 451)

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND ZORN [1986]. «Evaluation of the SPUR LISP architecture», *Proc. 13th Symposium on Computer Architecture* (June), Tokyo. (p. 759)

THACKER, C. P. AND L. C. STEWART [1987]. «Firefly: a multiprocessor workstation», *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., 164-172. (p. 526)

THACKER, C. P., E. M. MCCREIGHT, B. W. LAMPSON, R. F. SPROULL, AND D. R. BOGGS [1982]. «Alto: A personal computer», in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549-572. (p. 603)

THADHANI, A. J. [1981]. «Interactive user productivity», *IBM Systems J.* 20:4, 407-423. (p. 560)

THISQUEN, J. [1988]. «Seek time measurements», *Amdahl Peripheral Products Division Tech. Rep.* (May). (p. 602)

THORLIN, J. F. [1967]. «Code generation for PIE (parallel instruction excecution) computers», *Spring Joint Computer Conf.* (April), Atlantic City, N.J. (p. 364)

THORNTON, J. E. [1964]. «Parallel operation in Control Data 6600», *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33-40. (pp. 138, 364)

THORTON, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, I11. (p. 364)

TJADEN, G. S. AND M. J. FLYNN [1970]. «Detection and parallel execution of independent instructions», *IEEE Trans. on Computers* C-19:10 (October) 889-895. (p. 365)

TOMASULO, R. M. [1967]. «An efficient algorithm for exploring multiple arithmetic units», *IBM J. of Research and Development* 11:1 (January) 25-33. (p. 364)

TRELEAVEN, P. C., D. R. BROWNBRIDGE, AND R. P. HOPKINS [1982]. «Data-driven and demand-driven computer architectures», *Computing Surveys*, 14:1 (March) 93-143. (p. 635)

TROIANI, M., S. S. CHING, N. N. QUAYNOR, J. E. BLOEM, AND F. C. COLON OSORIO [1985]. «The VAX 8600 I Box, a pipelined implementation of the VAX architecture», *Digital Technical J.* 1 (August) 4-19. (p. 352)

TUCKER, S. G. [1967]. «Microprogram control for the System/360», *IBM Systems Journal* 6:4, 222-241. (p. 259)

UNGAR, D. M. [1987]. *The Design of a High Performance Smalltalk System*, The MIT Press Distinguished Dissertation Series, Cambridge, Mass. (p. 487)

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. «Architecture of SOAR: Smalltalk on a RISC», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188-197. (p. 203)

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. «Architecture of SOAR: Smalltalk on a RISC», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188-197. (p. 759)

UNGER, S. H. [1958]. «A computer oriented towards spatial problems». *Proc. Institute of Radio Engineers* 46:10 (October) 1744-1750. (p. 634)

VON NEUMANN, J. [1945]. «First draft of a report on the EDVAC», Reprinted in W. Aspray and A. Burks, eds.; *Papers of John von Neumann on Computing and Computer Theory* (1987), 17-82, The MIT Press, Cambridge, Mass. (p. 637)

WAKERLY, J. [1989]. *Microcomputer Architecture and Programming*, J. Wiley, New York. (p. 202)

WANG, E.-H., J.-L. BAER, AND H. M. LEVY [1989]. «Organization and performance of a two-level virtual-real cache hierarchy», *Proc. 16th Annual Symposium on Computer Architecture* (May 28-June 1), Jerusalem, Israel, 140-148. (p. 525)

WATANABE, T. [1987]. «Architecture and performance of the NEC supercomputer SX system», *Parallel Computing* 5, 247-255. (p. 423)

WATERS, F., ED. [1986]. *IBM RT Personal Computer Technology*, IBM, Austin, Tex., SA 23-1057. (p. 203)

WATSON, W. J. [1972]. «The TI ASC-A highly modular and flexible super computer architecture», *Proc. AFIPS Fall Joint Computer Conf.*, 221-228. (p. 422)

WEICKER, R. P. [1984]. «Dhrystone: A synthetic systems programming benchmark», *Comm. ACM* 27:10 (October) 1013-1030. (p. 50)

WEISS, S. AND J. E. SMITH [1984]. «Instruction issue logic for pipelined supercomputers», *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110-118. (p. 365)

WEISS, S. AND J. E. SMITH [1987]. «A study of scalar compilation techniques for pipelined supercomputers», *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105-109. (p. 365)

WESTE, N. AND K. ESHRAGHIAN [1985]. *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, Mass. (p. 702)

WHITBY-STREVENS C. [1985]. «The transputer», *Proc. 12th Int'l Symposium on Computer Architecture*, Boston, Mass. (June) 292-300. (p. 634)

WICHMANN, B. A. [1973]. *Algol 60 Compilation and Assessment*, Academic Press, New York. (p. 49)

WIECEK, C. [1982]. «A case study of the VAX 11 instruction set usage for compiler execution», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177-184. (p. 201)

WILKES, M. [1965]. «Slave memories and dynamic storage allocation», *IEEE Trans. Electronic Computers* EC.14:2 (April) 270-271. (p. 524)

WILKES, M. V. [1953]. «The best way to design an automatic calculating machine», in *Manchester University Computer Inaugural Conf.*, 1951, Ferranti, Ltd., London. (Not publisched until 1953.) Reprinted in «The Genesis of Microprogramming» in *Annals of the History of Computing* 8:116. (p. 259)

WILKES, M. V. [1982]. «Hardware support for memory protection: Capability implementations», *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1-3), Palo Alto, Calif., 107-116. (pp. 107, 524)

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer*, The MIT Press, Cambridge, Mass. (pp. 27, 259)

WILKES, M. V. AND J. B. STRINGER [1953]. «Microprogramming and the design of the control circuits in an electronic digital computer», *Proc. Cambridge Philosophical Society* 49:230-238. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles*

and Examples (1982), McGraw-Hill, New York, 158-163, and in «The Genesis of Microprogramming» in *Annals of the History of Computing* 8:116. (p. 266)

WILKES, M. V. AND W. RENWICK [1949]. *Report of a Conf. on High Speed Automatic Calculating Machines*, Cambridge, England. (p. 94)

WILKES, M. V., D. J. WHEELER, AND S. GILL [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Press, Cambridge, Mass. (p. 25)

WILLIAMS, T. E., M. HOROWITZ, R. L. ALVERSON, AND T. S. YANG [1987]. «A self-timed chip for division», *Advanced Research in VLSI, Proc. 1987 Stanford Conf.*, The MIT Press, Cambridge, Mass. (p. 689)

WILSON, A. W., JR. [1987]. «Hierarchical cache/bus architecture for shared memory multiprocessors», *Proc. 14th Int'l Symposium on Computer Architecture* (June), Pittsburg, Penn., 244-252. (p. 634)

WULF, W. [1981]. «Compilers and computer architecture», *Computer* 14:7 (July) 41-47. (p. 140)

WULF, W. A., R. LEVIN AND S. P. HARBISON [1981]. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York. (p. 524)

WULF, W. AND C. G. BELL [1972]. «C.mmp—A multi-mini-processor», *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765-777. (p. 635)

WULF, W. AND S. P. HARBISON [1978]. «Reflections in a pool of processors—An experience report on C.mmp/Hydra», *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif. 939-951. (p. 634)

Indice

Los números de página en negrita indican definiciones de términos.

∞ (*ver* infinito)
−∞ (*ver* infinito)
+∞ (*ver* infinito)
10000 (*ver* Apollo DN 10000)
11/780 (*ver* Digital Equipment Corporation, VAX-11/780)
11/785 (*ver* Digital Equipment Corporation, VAX-11/785)
2000 (*ver* MIPS Computer Corporation, 2000; Digital Equipment Corporation, VAXstation)
regla 2:1 de cache, 771
2100 (*ver* Sequent Corporation)
29000 (*ver* AMD 29000)
3000 (*ver* MIPS Computer Corporation, 3000)
3010 (*ver* MIPS Computer Corporation, 3010)
3090 (*ver* International Business Machines Corp.; disco, magnético y subsistema de almacenamiento IBM 3990)
3090-600S (*ver* International Business Machines Corp., IBM 3090-600S)
3100 (*ver* Digital Equipment Corporation, DECstation; Digital Equipment Corporation, VAXstation)
3364 (*ver* Weitek 3364)
360 (*ver* International Business Machines Corp., IBM 360)
360/85 (*ver* International Business Machines Corp., IBM 360/85)
360/91 (*ver* International Business Machines Corp., IBM 360/91)
370 (*ver* International Business Machines Corp., IBM 370)
370/158 (*ver* International Business Machines Corp., IBM 370/158)
370-XA (*ver* International Business Machines Corp., IBM 370-XA)
3990 (*ver* International Business Machines Corp.; disco, magnético y subsistema de almacenamiento IBM 3990)
68000 (*ver* Motorola Corporation, 68000)
6809 (*ver* Motorola Corporation, 6809)
701 (*ver* International Business Machines Corp., IBM 701)
7030 (*ver* International Business Machines Corp., IBM 7030)
704 (*ver* International Business Machines Corp., IBM 704)
7090 (*ver* International Business Machines Corp., IBM 7090)
8000 (*ver* Sequent Corporation)
801 (*ver* International Business Machines Corp, IBM 801)
8012 (*ver* International Business Machines Corp., IBM 8012)
80186 (*ver* Intel Corporation, 80x86, 80186)
80286 (*ver* Intel Corporation, 80x86, 80286)
80386 (*ver* Intel Corporation, 80x86, 80386)
80486 (*ver* Intel Corporation, 80x86, 80486)
80x86 (*ver* Intel Corporation, 80x86)
8080 (*ver* Intel Corporation, 8080)
8086 (*ver* Intel Corporation, 80x86, 8086)
8088 (*ver* Intel Corporation, 8088)
8550 (*ver* Digital Equipment Corporation, VAX)
860 (*ver* Intel Corporation, 860; Intel Corporation, i860)
8600 (*ver* Digital Equipment Corporation, VAX)
8700 (*ver* Digital Equipment Corporation, VAX)
88000 (*ver* Motorola Corporation, 88000)
88100 (*ver* Motorola Corporation, 88100)
88200 (*ver* Motorola Corporation, 88200)
8847 (*ver* Texas Instruments, 8847)
regla 90/10, 771 (*ver también* localidad, principio de)
regla 90/50 de saltos efectivos, 771 (*ver también* salto, efectivo

A

abortos, **231** (*ver también* interrupciones)
acarreo, 642, 652 (*ver también* de entrada; acarreo de salida; sumador con anticipación de acarreo; sumador con propagación de acarreo; sumador

acarreo (*cont.*)
con ahorro de acarreo; sumador con selección de acarreo; sumador con salto de acarreo)
de entrada, **642**, 643 (fig.), 647, 679, 680
de salida, 642, 643 (fig.), 647, 659, 661, 675, 679
acceso directo a memoria (DMA), **576**-577 (*ver también* entrada/salida, DMA y)
aceleración, 9-12, 22, 28, 30, 31
debida a la segmentación, 277-278, 297
definición de, 9
global, 11
lineal, 620, 629-631, 638
mejorada, 11
acierto, **435** (*ver también* jerarquía de memoria, acierto; cache, acierto)
Adams, T., 202
adelantamiento, **281**-285, 289, 307, 364
Agarwal, A., 204
agrupar, **409** (*ver también* procesador vectorial, matrices dispersas)
Aiken, 26
aleatorio, **411** (*ver también* remplazo de bloques, aleatorio)
Alexander, W. G., 139, 200
algoritmo, 16
de Goldschmidt, 666-667, 700
de Tomasulo, 321-329, 364 (*ver también* planificación dinámica)
detección de riesgos y, 324, 326 (*ver también* riesgo, detección)
DLX y, 323 (fig.) 329
frente a planificación estática, 331
alias, **495**
alineación (*ver también* alineación de datos; pila, alineación de)
de acceso (*ver* alineación de datos)
de datos, 101-103
en DLX, 236, 248
interrupciones y, 231
almacenamiento (*ver* memoria; disco; disco, magnético; entrada/salida)
de control escribible (WCS), **256**-258, 265
de operando, 97-99
en memoria, 99-101
de sólo lectura (*ver* memoria de sólo lectura)
de un nivel (*ver* memoria virtual)
de un solo nivel, 466 (*ver también* memoria virtual)
del resultado, 353 (fig.), 355
directo, **444** (*ver también* cache, escritura directa)
inmediato (SI) (*ver* International Business Machines Corp., IBM 360, repertorio de instrucciones)
interno, 96-99
(SS) (*ver* International Business Machines Corp., IBM 360, repertorio de instrucciones)
Alto, 603
ALU (*ver* unidad aritmética lógica)
AMD 29000, 179, 203
Amdahl, G. M., 19, 28, 137, 200, 259, 633-634 (*ver también* Ley de Amdahl)
Amdahl/Case regla (*ver* regla empírica Case/Amdahl)
amos de bus, **570** (*ver también* bus)
ancho de banda, 5, 19, 21 (fig.), 31, 132, 145
E/S (*ver* entrada/salida, rendimiento, productividad)
medidas de rendimiento de memoria principal y, 458
ancho de bus, 461 (*ver también* memoria, organización de)
ancho de memoria (*ver* memoria, ancho)
anillos, **475** (*ver también* memoria virtual, esquemas de protección de)
antes de redondear, 657-658, **663** (*ver también* aritmética, redondeo)
antialias, **496**(*ver también* cache virtual)
antidependencia, **402** (*ver también* procesador vectorial, antidependencia)
de datos (*ver* antidependencia)
AP-120B (*ver* Sistemas de Punto Flotante)
Apollo DN 10000, 365
árbol de Wallace, **690,** 690, 702 (*ver también* multiplicador de arrays; aritmética)
Archibald, J., 508, 526
área de datos, global, **124**
área global de datos (*ver* área de datos, global)
aritmética, 15, 215, 641
con signo, 647, 650, 701
representación de dígitos con signo, **690-691**
representación de logaritmos con signo, 708
de punto flotante, cuádruple precisión, **761**
decimal, 15, 110, 117-118 (*ver también* aritmética, entero; aritmética, punto flotante)
de normales, **655**, 661, 663, 664, 673, 703
división,
entera, 643, 647 (*ver también* aritmética, entera)
no restauración, **645**, 646 (fig.), 682, 683-684, 684-685
restaurar, **645**, 646 (fig.)
punto flotante, 664-665, 668
entera (*ver* aritmética, entera)
entero, 642, 651, 701
acelerar división, 682, 684, 692, 696
con un solo sumador, 692, 696
desplazamiento sobre ceros, **682**, 684
acelerar multiplicación, 682, 692
desplazamiento sobre ceros, **682**
con muchos sumadores, 686, 691
con un solo sumador, 685, 686
acelerar suma, 673, 681
suma con selección de acarreo, **680**-682, 681 (fig.), 697, 709 (*ver también* acarreo)

aritmética (*cont.*)
sumador con anticipación de acarreo (CLA), **674**, 678, 681 (fig.), (*ver también* acarreo)
sumador con salto de acarreo, **678**, 679, 681 (fig.) (*ver también* acarreo)
facilidades de sistemas de, 650, 651-652
multiplicación y división en base 2, 643, 646
números con signo y, 647, 650
suma con propagación de acarreo, **642**, 643, 678 (fig.), 681 (fig.)
suma en múltiple precisión, 651-652
técnicas básicas de, 642, 651-652
estándar del IEEE y, 117, 641, 653, 657, 703, 744, 768
excepciones, 672-673
desbordamiento, **647**, 650, 651 (fig.), 672, 673
entera, 650
multiplicación en punto flotante, 662
suma en punto flotante, 661
desbordamiento a cero, 661, 662, 663, 672, 700, 701
excepción, 673
gradual, **655**, 664, 702, 703, 706
exponentes y, 641, 652, 654, 655, 656
y campo de exponente, **654**, 655, 662
falacias y pifias, 700-701
historia de, 701-703
instrucciones de suma, resta y multiplicación en Intel 860, 766
mantisa, 652, **654**, 659, 664, 671
multiplicación, punto flotante, 662, 664
precisión, 663-665
denormales y, 663-665
multiplicación y división, entero, 643, 647 (*ver también* aritmética, entero)
operaciones, 110
precisión, 663-665, 670-672
doble extendido, 670, 703
suma con múltiple precisión, 652
punto flotante, 652-673, 701, 701, 703 (*ver también* aritmética, estándar IEEE y)
división, 664-668
excepciones, 671-673
multiplicación, 661-665 (*ver también* aritmética, multiplicación, punto flotante)
precisión, 663-665, 669-672
resto, 667-670
suma, 657-662 (*ver también* aritmética, suma, punto flotante)
raíz cuadrada, 666, 668, 670-672, 707
de un número negativo, 652, 654
recodificación de Booth, **648**-650, 662, 682, 685-687, 697-702
modificado, 707
redondeo, 654, 657-658, 659, 660, 661-663, 664, 668 (ver *también* infinito)
antes de redondear, 658, **663**
después de redondear, 662, **663**
doble redondeo, **671**, 707
errores de redondeo, 666, 667
modo de redondeo, **654**, 663
resto, 644-645, 682-685, 692
punto flotante, 667-670
REM, 667-670, 696 (*ver también* aritmética, resto)
sistemas y, 650-652 (*ver también* no un número; infinito)
suma, entero, 642-643 (*ver también* aritmética, entero)
algoritmo para, 659-661
denormales y, 661
punto flotante, 657-662
redondear en, 657-658 (*ver también* infinito)
Armstrong, F. H., 85
ARPANET, 568 (fig.), 605 (*ver también* redes)
arquitectura, **3, 4, 5, 13**, 138 (*ver también* Digital Equipment Corporation, VAX; DLX; HLLCA; Intel Corporation 860; Intel Corporation 80x86; International Business Machines Corp., IBM 360; MIPS Computer Corporation R3000; Motorola Corporation; SPARC)
acumulador, 25, 96-99, 136
almacenamiento (*ver* arquitectura memoria-memoria)
basada en acumulador (*ver* arquitectura, acumulador)
básica de un procesador vectorial (*ver* procesadores vectoriales, arquitectura)
carga/almacenamiento (*ver* arquitectura de carga/almacenamiento)
de acumulador (*ver* arquitectura, acumulador)
de carga/almacenamiento, 41-45, **99, 101**, 131, 363 (*ver también* computador de repertorio reducido de instrucciones; DLX)
de computadores (*ver* arquitectura, computador)
orientada a Lenguajes de Alto Nivel (HLLCA), **138**-140
de etiquetas (*ver* SPARC)
de pila, **97**-99, 137
de registros de propósito general (GPR) (*ver* registro, arquitectura de registros de propósito general)
definiciones,
desacoplada, 344
desacoplado (*ver* arquitectura desacoplada)
espectro de evolución, revolución, **632**-634
evaluación del rendimiento de, 83-85
evolución frente a revolución, 632-634
fórmulas, contraportada anterior
Harvard, 26

arquitectura (*cont.*)
memoria-memoria, **99**-101, 131-134, 137-139, 144
instrucción registro-memoria, 41-43
orientada al lenguaje (*ver* lenguaje de alto nivel)
pila (*ver* arquitectura de pila)
registro-memoria, **99**-101, 137
registro, de propósito general (*ver* arquitecturas de registros de propósito general)
-almacenamiento (*ver* arquitectura memoria-registro; International Business Machines Corp., IBM 360, repertorio de instrucciones)
-memoria (*ver* arquitectura registro-memoria)
-registro, **99**-101 (*ver también* arquitectura de carga/almacenamiento)
reglas empíricas (*ver* reglas empíricas)
repertorio de instrucciones (*ver* repertorio de instrucciones, arquitectura)
revolucionaria, 638
Simposio Internacional Anual sobre, 85
simulador, 51
sistólica, **624**, 635
sistólico (*ver* arquitectura sistólica)
tendencias de, 17
trivialidades, 770
vectorial (*ver* procesador vectorial, arquitectura) (*ver* procesador vectorial, arquitectura de)
array, 687-689 (*ver también* arrays sistólicos)
de discos, 560-562
disponibilidad de, 560-562
fiabilidad de, 560-562
de rejilla de patillas (PGA), 64, 89 (*ver también* paquete)
lógico programado (PLA), **220**-221, 247, 249
sistólico, **624**, 635 (*ver también* array)
arrays redundantes de discos baratos (*ver* array de disco)
ASCII, 117
asociativa por conjuntos (*ver también* cache, asociativa por conjuntos)
de *n* vías (*ver* cache)
asociatividad, 454 (*ver también* cache, completamente asociativa; cache asociativa por conjuntos)
ASP (*ver* coste, precio de venta medio)
ASPLOS (Soporte Arquitectónico para Lenguajes de Programación y Sistemas Operativos) conferencia, 139-140
Atanasoff, J. V., 26
atómica, **509** (*ver también* cache, coherencia, sincronización)
Auslander, M. A., 141
autodecremento, **105** (*ver también* modo de direccionamiento)
autorización de acceso (*ver* memoria virtual, Intel 80286/80386 y; memorial virtual, esquemas de protección de)

B

B5000 (*ver* Burroughs)
B5500 (*ver* Burroughs)
B6500 (*ver* Burroughs)
Baer, J.-L., 508, 525, 526
nivel inferior, **434** (*ver* jerarquía de memoria, cache; memoria; memoria virtual)
Barton, R. S., 137
basado en directorio, **504**-505 (*ver* cache, coherencia)
base, **474** (*ver también* memoria virtual, esquemas de protección de; memoria virtual, Intel 80286/80386 y)
Baskett, F., 204
BCD (*ver* decimal codificado en binario)
Bell, C. G., 85, 137, 524
apuesta con Hillis, 635
W. D. Strecker y, 524, 528
benchmark, 45, 46, 48-52, 56, 77, 79, 87, 88, 90-92 (*ver también* disco, magnético, benchmarks de E/S para; entrada/salida, rendimiento)
de E/S del sistema de ficheros (*ver* benchmark; entrada/salida, rendimiento; disco, magnético, benchmarks de E/S para)
TP-1, 550, 551 (fig.), 609
del Perfect Club (*ver* Penchmark)
E/S en el sistema de ficheros, 550 (*ver también* entrada/salida, benchmarks)
injustos, **528**
Linpack (*ver* procesador vectorial, benchmarks Linpack)
núcleos, 48
Perfect Club, 79, 84-86
vectorización y, 403
perspectiva histórica, 82-86
procesamiento de transacciones de E/S, 550-552 (*ver también* transacciones de, entrada/salida)
reducidos, 48
sintético, 48-52, 77-79, 91
Dhrystone, 48, 50, 77-79, 90, 91
Whetstone, 48-50, 77-79, 82, 87, 88, 91
SPEC (Cooperativa de Evaluación de Rendimientos de Sistemas), 51, 76-78, 84, 86, 88
supercomputador E/S, 548-551 (*ver también* entrada/salida, benchmarks)
benchmarks de E/S de procesamiento de transacciones (*ver también* disco, magnético, benchmarks de E/S para)
de supercomputadores (*ver* entrada/salida, supercomputadores y; disco, magnético, E/S para)
injustos, **528** (*ver también* benchmark)
Berry, M. D., 85
Bigelow, Julian, 26

bit, de acceso, **481** (*ver también* memoria virtual, tabla de página)
de error, **575** (*ver también* entrada/salida)
de modificación, **477** (*ver también* memoria virtual, tabla de páginas; memoria virtual, bit de modificación)
de modificación, **444** (*ver también* cache, escritura; memoria virtual, bits de modificación y)
de presente, **481** (*ver también* memoria virtual, tabla de página; memoria virtual, Intel 80286/80386 y)
de hecho, **575** (*ver también* entrada/salida)
de referencia, **470** (*ver también* reemplazo de bloque; menos recientemente usado)
de uso, **470** (*ver también* reemplazo de bloques, menos recientemente usado)
de validez, **441, 477** (*ver también* cache, bloques y; memoria virtual, tabla de páginas)
retenedor, **658**, 659, 665, 703
bits de blts, **563** (*ver también* pantallas gráficas)
Blaauw, G. A., 137, 200
Blau, R., 205
bloque, **434** (*ver también* jerarquía de memoria, bloques y; cache, bloques y; memoria virtual, página; memoria virtual, segmento)
básico, **122**
de control de E/S, **576-577, 592** (*ver también* interfaz de entrada/salida a la CPU; disco, magnético, subsistema de almacenamiento IBM 3990 y)
bloqueo circular, 510 (*ver también* cache, coherencia, sincronización)
Boggs, D., 603, 606
recodificación de Booth, **648** (*ver también* aritmética, recodificación de Booth)
Brady, J., 547, 604, 605
brazo, **555** (*ver también* disco, magnético)
Briggs, F., 204
Brooks, F. P., 137, 200, 480
Brown, E., 204
bucle, 122-125 (*ver también* desenrollamiento de bucle)
salto (*ver* salto, bucle)
segmentación software (*ver* segmentación, bucle con segmentación software)
segmentado-software (*ver* segmentacion, bucle segmentado-software)
bucle de espera, 510 (*ver también* cache, coherencia, sincronización)
bucles desenrollados (*ver* desenrollar bucles)
buffer, de adaptación de velocidad, **582, 591** (*ver también* entrada/salida)
de destinos de saltos, **333**-335, 364-366
de dirección del buffer de instrucciones físicas (PIBA), **514** (*ver también* jerarquía de memoria, VAX-11/780 y)
de escritura (*ver* cache, buffer de escritura; cache, escrituras y)
de encuadre, **562** (*ver también* pantallas gráficas)
de prebúsqueda de instrucciones, **484**-486, 523 (fig.)
resumen de, 523 (fig.)
VAX-11/780 y, 485 (fig.)
de predicción de saltos, **331**-333
de refresco de exploración, **562** (*ver también* pantallas gráficas)
de resultado, 282
de traducción anticipada (TLB), **472** (*ver también* memoria virtual, buffer de traducción anticipada)
z, **566** (*ver también* pantallas gráficas)
buffers de carga y almacenamiento, 323-326, 330 (fig.)
burbuja, **285** (*ver también* detención de la segmentación)
Burks, A. W., 26
Burr, W. E., 84
Burroughs
B5000, 136-137
B5500, 76
B6500, 137, 141
bus, **569**-574, 604
asíncrono, **572**
IBM PC-AT, 574
común de datos (CDB), **323**-331
de conexión/desconexión, **571**
de E/S, **570**
DLX, 214 (fig.), 215
estándares para, 572-574
comparación de cinco buses estándares, 573 (fig.)
FutureBus, 574, 573 (fig.)
instrucciones en DLX, 225, 246 (fig.)
interfaz, de periféricos inteligentes (IPI), **574**, 573 (fig.), 604
de sistemas de pequeños computadores (SCSI), **573** (fig.), 603-**604**
Multibus II, 574, 573 (fig.)
NuBus, 15 (fig.), **604**
opciones para, 571 (fig.), 572 (fig.)
segmentado, **571**
síncrono, **571**
transacciones, 570
Unibus PDP-11, 574, 604
VME, 574, 573 (fig.)
búsqueda, **555** (*ver también* disco, magnético, búsquedas)
búsqueda, de OP (Opfetch) (*ver* Digital Equipment Corporation, Búsqueda de Op)

búsqueda, de OP (*cont.*)
en escritura, **445** (*ver también* cache, fallo de escritura)
fuera de orden, **494** (*ver también* cache, fallo)
byte, 234
alineación de en DLX, 236, 248 (fig.)
cargar byte en DLX, 249 (fig.), 250

C

caballos de Troya, **480** (*ver también* memoria virtual, esquemas de protección de)
cabeza, de cadena, **592** (*ver también* disco, magnético; subsistema de memoria IBM 3990 y)
de lectura/escritura, **555** (*ver también* disco, magnético)
cache, **20**-22, 26, 29, 255, **438**-459, 489-512, 519, 521, 523 (fig.), 524-526 (*ver también* memoria; jerarquía de memoria; memoria virtual; identificación de bloques; ubicación de bloques; reemplazo de bloques; estrategia de escritura)
acierto, 443-445, 445, 495
frecuencia de, 442
lectura, 444
aleatorio (*ver* sustitución de bloque, aleatorio)
asociativa por conjuntos, **439**, 440 (fig.), 441 (fig.), 490, 519
campo de desplazamiento de bloque, **441**, 442 (fig.)
campo índice, **441**, 442 (fig.)
de n-vías (*ver* cache asociativa por conjuntos de n-vías)
fallos debidos a conflictos y, 454
n vías, 439, 452-455
bit de validez, **441**
bloques, identificación (*ver* identificación de bloques, caches y)
reemplazo (*ver* reemplazo de bloques, caches y)
subbloques, 491-493
tamaño, 454, 456, 490, 506
frente a tiempo de acceso a memoria, 454-456 (figs.)
ubicación (*ver* ubicación de bloques, caches y)
VAX-11/780, 445
y, 439, 454, 458, 490
y etiqueta de dirección, 440
buffer de escritura, **445**, 493, 520-522 (*ver también* cache, escrituras y)
detenciones de escrituras y, 493-495
falacia de, 520-522
VAX-11/780 y, 445, 447, 514-515, 521
cache mezclada (*ver* cache, unificada)
caches de dos niveles, 495-502, 523 (fig.), 525-526
coherencia, 505 (*ver también* cache, coherencia)
parámetros, típicos para, 500 (fig.) (*ver también* parámetros, rangos típicos de)
resumen de, 523 (fig.)
tamaño de, 498
tiempo de acceso a memoria para, 497
tiempo relativo de ejecución, 500 (fig.), 502 (fig.)
campo de etiqueta, **441**, 442 (fig.)
coherencia, 502, 502-512, 525
aciertos de escritura y, 506
aciertos de lectura y, 508
caches multinivel y, 505
consistencia débil, 512
consistencia secuencial, 512
ejemplo, 510 (fig.)
fallos de escritura y, 506-508
fallos de lectura y, 506-508
problema de coherencia cache, **503** (fig.), 505
protocolos de coherencia, cache fallos y, 504-509
cache espionaje, resumen de, 508 (fig.)
cache, **502**-511
aciertos y, 506
basado en directorio, **504**-505
difusión de escritura, **505**-507 (fig.)
ejemplo de, 505-508
espionaje, **504**-512
invalidación de escritura, **505**-506, 507 (fig.)
sincronización, 508-512
desbloquear, 509 (fig.)
variable de bloqueo, **508**, 509 (fig.), 511
tamaño de bloque y, 506
correspondencia directa, **439**, 440 (fig.), 441 (fig.), 450-455, 492, 519, 524
fallos debidos a conflictos y, 452
partes de dirección de, 441 (fig.)
regla de cache 2:1,
datos obsoletos y (*ver* datos obsoletos)
de disco, 580 (*ver también* entrada/salida, interfaz con un sistema operativo)
de dos niveles (*ver* cache, caches de dos niveles)
de espionaje (*ver* cache, coherencia)
de ficheros, 580 (*ver también* entrada/salida, interfaz a un sistema operativo)
de sólo datos (*ver* cache, sólo datos)
de sólo instrucciones (*ver* cache, sólo instrucción)
diferencias entre memoria virtual y, 472
dirección de bloque, 441, 443 (fig.), 445
disco, 471, **609** (*ver también* entrada/salida, interfaz para un sistema operativo)
escritura directa, **444**-447, 447, 493, 514
escrituras y, 444-447, 448 (*ver también* cache, fallos de escritura)

cache (*cont.*)
hacer escrituras más rápidas, 490-493
y multiprocesador, 504
estrategia de escritura (*ver* estrategia de escritura, caches y)
E/S y, 502-504
fallo, **20**, 443, 445, 450, 451-455, 463, 495 (*ver también* cache, frecuencia de fallos; cache, fallo de escritura)
capacidad, 452, 453 (fig.), 455 (fig.)
conflicto, 452, 453 (fig.), 455 (fig.)
forzoso, 452, 453 (fig.), 455 (fig.)
reducción de penalización debida a fallos, 493-495 (*ver también* cache, caches de dos niveles)
«tres C» (capacidad, compulsorio forzoso, conflicto), 452, 453 (fig.), 455 (fig.), 522
fallos de escritura, 444-447, 448, 493-495
hacer más rápidas, 493-495 (*ver también* ubicación de subbloques)
multiprocesadores y, 504-505
no ubicación en escritura, **445**-477, 447
ubicación en escritura, **445**-447
fichero, 579 (fig.), 580 (*ver también* entrada/salida, interfaz para un sistema operativo)
frecuencia, de comparado con fallos por instrucción, 448
de fallos, 448, 450-452, 519 (*ver también* cache, fallo; cache, fallo de escritura)
de datos frente a instrucciones, 457 (fig.), 458
en DLX, 520 (fig.)
en el VAX-11/780, 520 (fig.)
multiprocesadores y, 504-505 (*ver* cache, coherencia)
para aleatorio frente a reemplazo del bloque menos recientemente utilizado, 443
reducción, frente tamaño cache utilización de una etiqueta de identificación de proceso (PID), 496 (fig.)
frente tamaño cache, 491 (fig.)
para caches de dos niveles, 499 (fig.)
por reducción de limpiezas de cache, 502-504
regla cache 2:1,
lecturas y, 443, 448 (*ver también* cache, fallos)
frecuencia de fallos de lectura, 448
máquinas de segmentación y, 468
memoria virtual y, 468, 473
mixta, **456**
multinivel (*ver* cache, caches de dos niveles)
multiprocesadores y (*ver* cache, coherencia)
parámetros, típico, 439 (fig.) (*ver* también parámetros, rangos típicos de)
postescritura, **444**-447, 462, 506
bit de modificación, **444**
limpio, **444**
modificado, **444**
reducción más rápida de aciertos de cache con caches direccionadas virtualmente, 495-497
registro frente, velocidad de, 521-522
regla de cache 2:1,
relación entre SRAM, 459
rendimiento, 447-452, 489-512, 521-522
resumen de, 523 (fig.)
sincronización (*ver* cache, coherencia)
sólo datos, 454-458
sólo instrucción, 454-458
subbloques, **492**-493, 531
sustitución del bloque menos recientemente usado
tiempo de acceso y, 454
cache virtual, 495-497
tamaño de bloque frente, 456 (fig.), 457 (fig.)
tiempo medio de acceso a memoria, 451, 489
totalmente asociativa, **439**, 440 (fig.), 441 (fig.), 450-455, 490
fallos y, 452
sustitución de bloques y, 442, 454
ubicación de bloques y, 441 (fig.)
unificada, **456**
VAX-11/780 y, 445-448 (*ver también* jerarquía de memoria, VAX-11/780 y; memoria virtual, VAX-11/780 y)
y frecuencia de fallos para, 520 (fig.)
virtual, **495**
antialias, **496**
caches compartidas (*ver* cache, coherencia)
cadenas de caracteres, 117
Cady, R., 141
cambio de contexto, **473** (*ver también* memoria virtual, procesos y)
caches virtuales y, 495, 497 (fig.)
camino de datos, **215**
arquitectura DLX y, 236
control y, 242
datos desde, 219, 220 (fig.)
diseño, 218, 221
microinstrucciones y, 222-224, 225, 229
campo, base, **481** (*ver también* memoria virtual, tabla de página)
de atributo, **481** (*ver también* memoria virtual, tabla de página; memoria virtual; Intel 80286/80386 y)
de desplazamiento de bloque, **441** (*ver también* cache, bloques y)
de etiqueta, **441**
de exponente, **654** (*ver también* aritmética, exponentes y, campo de exponente)
de formato, **226**
de índices, **441** (*ver también* cache, asociativa por conjuntos)

campo (*cont.*)
de límite, **481** (*ver también* memoria virtual, tabla de páginas; memoria virtual, Intel 80286/80386 y)
campos, **223**
canal, **590** (*ver también* disco, magnético, subsistema de memoria IBM 3990 y)
capacidad
de DRM (*ver* memoria dinámica de acceso aleatorio)
de SRAM (*ver* estática de acceso aleatorio)
capacidades, **475**, 523 (*ver también* memoria virtual, esquemas de protección de)
carga, de trabajo, 48
retardada, **287**, 364
CAS (*ver* strobe de acceso de columna)
Case, R., 19, 200
caso común
importancia en diseño, 8
castigada, **454** (*ver también* jerarquía de memoria; cache)
CC (*ver* salto, código de condición)
CD (*ver* disco, óptico)
CDB (*ver* bus común de datos)
CDC (*ver* Control Data Corporation)
CD-ROM, **559** (*ver* disco, óptico)
cerrojo Earle (*ver* cerrojos)
cerrojos, 271-274, 364
cerrojo Earle, 272-274
sobrecarga de cerrojo, 361
Chaitin, G. J., 140
Chandra, A. K., 141
choque, semántico, **133**
Chow, F. C., 123-124, 126, 140
ciclo de reloj, 31, **38**-41, 80, 82, 84, 86, 144, 215, 239, 243 (*ver también* duración de ciclo de reloj; ciclos de reloj por instrucción)
ALU y, 239-242, 253
caches y, 447
segmentación y, 298 (fig.), 377
control y, 218
detenciones y, 226-229, 239
DLX y, 239, 252, 254-256
fichero de registros y, 215
microinstrucciones y, 226-227
por microinstrucción (CPI) (*ver* ciclo de reloj por instrucción)
reducir, 221
saltos y, 239-241, 254-255
ciclos de reloj por instrucción (CPI), 38-44, 75-77, 82, 87, 100, 142, 144, 213, 239
ALU y, 239-242, 252
CPF frente a, 421
DLX y, 239, 240 (fig.), 252, 255
reducción con microcódigo para casos especiales, 228
reducción, 221
por adición de control cableado, 227-229
por paralelismo, 228, 337-351
por segmentación, 270, 277
rendimiento y, 224
segmentación segmentada y, 270, 277, 377
ciclos de reloj por operación de punto flotante (CPF), 386-388, 406, 421
cilindro, **555** (*ver también* disco, magnético)
circuito integrado, coste de, 59-60
(IC), 3, 5, 13, 18, 28
producción de, 63, 86
CISC (computador de repertorio complejo de instrucciones) (*ver* computador de repertorio reducido de instrucciones; Digital Equipment Corporation, VAX)
CLA (*ver* sumador con anticipación de acarreo)
Clark, D. W., 140, 184, 201, 203, 525, 526
CM (*ver* Connection Machine
Cm* multiprocesador, 634 (*ver también* multiprocesador)
COBOL, 16
Cocke, J., 140, 202, 365
codificación **224**-226, 252 (*ver también* modo de direccionamiento)
codificado máximamente, **226** (*ver también* microcódigo, vertical)
codificado mínimamente, **226** (*ver también* microcódigo, horizontal)
código (programa), 48, 51
automodificable, 359
condición (*ver* salto, código de condición)
de usuario (*ver* código, usuario)
fuente, 46
no optimizado, 44-45, 78
optimizado, 44-45, 52, 78
sistema, 37
tamaño, 74-76, 78, 83-85, 110, 129-130, 145, 347
usuario, 38
coherencia (*ver* cache, coherencia)
colas, 344, 366
colorear, grafo, 121-122, 140
columna DRAM estática, **465** (*ver también* memoria dinámica de acceso aleatorio, columna estática; memoria, DRAM)
¿Cómo se encuentra un bloque? (*ver* identificación de bloques)
compactación de trazas, **350**
Comparabilidad, de repertorio de instrucciones (*ver* compatibilidad código-objeto)
comparación (*ver* comparar)

comparaciones de punto flotante, 114-115
enteras, 114-115
comparadores
para detección de riesgos, 288 (*ver también* riesgos, detección)
comparar, 108, 110, 114-115
en arquitecturas RISC, 750, 752
mejora del macrocódigo de, en el VAX-11/780, 256
compartición falsa, **506** (*ver también* cache, coherencia)
compatibilidad de código objeto, 4
compilador, 5, 18, 21, 23, 30, 98-101, 118-132
GnuC, 71, 73-75, 84, 90
Ultrix, 72
complejidad de, 119, 128-130
direcciones futuras para, 625-627
estructura de, 119-124
optimizar, 44, 50, 77-79, 86, 119-130, 135, 140, 146
procesador vectorial y (*ver* procesador vectorial, compiladores y)
rendimiento y, 39, 45-52, 75-77, 85
complemento, a dos, 645, **647**-650, 659-661
a uno, **647** (*ver también* aritmética, con signo)
comportamiento, **551** (*ver también* entrada/salida, dispositivos)
comprobación de límites (*ver* memoria virtual, Intel 80286/80386 y)
compromisos (*ver* equilibrio)
computador, Atlas, 28, 524
de flujo de datos múltiples-flujo de instrucciones simple (SIMD), **616**-619, 622, 634, 637, 638
de flujo de datos simple-flujo de instrucciones múltiple (MISD), 617, 625
de flujo de instrucciones múltiple-flujo de datos múltiple (MIMD) MIMD fuertemente acoplado (*ver* multiprocesador)
de datos múltiple (MIMD) MIMD débilmente acoplado (*ver* multicomputador)
de datos múltiple (MIMD), 618-621, 622, 633, 636, 637, 638
de programa almacenado, 24-27
de repertorio, complejo de instrucciones (CISC) (*ver* computador de repertorio reducido de instrucciones)
reducido de instrucciones (RISC) arquitectura, visión de instrucciones de control de flujo, 748-749
visión de transferencia de datos, 747-748
visión de instrucciones aritméticas y lógicas, 747-748
reducido de instrucciones (RISC), 140, 141, **202**-204, 362, 364-366 (*ver también* International Business Machines Corp., IBM 801; MIPS Computer Corporation)
arquitectura, visión de mejora de rendimiento de, 202
visión de saltos condicionales de RISC, 752
visión de instrucciones de punto flotante, 748-749
visión de formato de instrucción, 745
visión de extensión constante, 746
visión de modo de direccionamiento, 744
visión, 743-767
de Berkeley, 202
de multiplicación y división entera, 751-753
grande, 3-4
frente a minicomputador, 537
MIMD (*ver* computador de flujo múltiple de instrucciones-flujo múltiple de datos)
MISD (*ver* computador de flujo múltiple de instrucciones-flujo simple de datos)
Pegasus, 136
personal (PC), 604 (*ver también* Intel Corporation, 80x86; Intel Corporation, 8o88; International business Machines Corp., IBM PC)
frente a estación de trabajo, 538
SIMD (*ver* computador de flujo de datos múltiples-flujo simple de instrucciones)
Computer Museum, 27
comunicación, 617, 618-619, 636-637, 639
explícita, 622
implícita, 622-624
sobrecarga, 619, 625-627
congelar el procesador segmentado, 293, 359
Connection Machine
CM, 634-635, 636
CM-2, 617, 621, 638
consistencia, máxima, 511-512 (*ver también* cache, coherencia)
secuencial, 511-512 (*ver también* cache, coherencia)
contador de programa (PC), **113**
direccionamientos relativos al PC (contador de programa), **104**-105, 111-114
saltos relativos al PC (contador de programa) (*ver* salto)
VAX 8600 y, 355
Conti, C. J., 85
control, 213, 215 (fig.)
cableado, 218-222, 224
mejorar el rendimiento de DLX cuando el control es cableado, 241-244
reducir costes hardware de control cableado, 219-221, 226-229
reducir el CPI añadiendo control cableado, 226-229
rendimiento de, 221, 239-241, 254
Data Corporation (CDC), 379, 423

control (*cont.*)
CDC 6600, 76, 137, 141, 314, 318, 321-322
ETA-10, 423
DLX y, 235-240, 243-252 (*ver también* DLX, repertorio de instrucciones, instrucciones de control de flujo)
flujo (*ver también* instrucciones de control de flujo)
interrupciones y, 231-234
mediante máscara vectorial, **407** (*ver también* procesador vectorial, control de máscara vectorial)
microcodificado (*ver* control microprogramado/microcodificado)
microprogramado (*ver* control, microprogramado/microcodificado)
microprogramado/microcodificado, control microcodificado para el DLX, 243-252
ABC de la microprogramación, 218-225
memoria de control escribible (WCS) y, 255-257
microprogramado/microcodificado, 222-229, 255-261
reducir costos de hardware con formatos múltiples de microinstrucción, 225-227
microcódigo «para casos» especiales, 228
reducir costes y mejorar el rendimiento del DLX cuando el control es microcodificado, 250-255
reducir costes de hardware codificando líneas de control, 224-226
rendimiento de, 255, 257-259
rendimiento del control microcodificado para el DLX, 252
controlador, disco (*ver* disco, magnético)
de disco, **556** (*ver también* disco, magnético)
controladores, de canal, **576** (*ver también* entrada/salida, interfaz a la CPU)
de E/S, **576** (*ver también* entrada/salida, interfaz a la CPU)
conversiones implícitas (*ver* instrucciones de punto flotante)
coprocesador, **624**
correspondencia directa, **439** (*ver* también cache, correspondencia directa)
cortocircuito, **281** (*ver también* adelantar)
coste, 36, 57-58, 85 (*ver también* dado; circuito integrado; encapsulado; oblea; estación de trabajo)
comparar precio de los medios frente al precio del sistema empaquetado, 600-601
descuento medio, 68-70, 89-91
directo, 68-72, 90
disco magnético, coste, 600-601
coste frente a tiempo de acceso para SRAM, DRAM, y discos magnéticos, 559 (fig.)
DRAM, 600-601
frente a precio, 65-69, 69 (fig.), 70 (fig.), 89-91
indirecto, 69
precio, de lista, 68-73, 89-91
medio de venta (ASP), 69, 70, 90
/rendimiento, 12, 17, 23, 81 (*ver también* rendimiento)
diseño, **36**
falacias, 74
optimización, 16
precio/rendimiento, 50, 71-75, 85
CPA (*ver* sumador con propagación de acarreo)
CPF (*ver* ciclos de reloj para operación en punto flotante)
CPI (*ver* ciclos de reloj por instrucción)
CPU (*ver* unidad central de proceso)
Crawford, J., 202
Cray, Seymour, 76
Criba de Eratóstenes (*ver* benchmark, reducido)
CRT (*ver* tubo de rayos catódicos; pantallas gráficas)
Crudele, L., 204
CSA, 685 (*ver también* sumador con ahorro de acarreo)
encapsulamiento plano cuadrado de plástico (PQFO), **64** (*ver también* paquete)
cuello de botella de Flynn, **377**, 378 (*ver también* procesador vectorial)
Curnow, H. J., 85
curva de evolución, 57, 58 (*ver también* producción)
Cydra 5 (*ver* Cydrome Cydra 5)
Cydrome Cydra 5, 366
Cypress Corporation
Cypress CY7C601 microprocesador, 89, 531

D

dado, 58-62 (*ver también* oblea)
área, 63-64, 65
coste de, 59, 63-64, 65, 66, 89, 90
fotografías de, 62
rendimiento, 63-66, 66, 85
test, 64
coste de, 59, 64, 66, 89
DASD, **554** (*ver también* dispositivo de memoria de acceso directo disco, magnético; entrada/salida)
Nova de Data General, 604
datos obsoletos, **502**, 576-579 (*ver también* cache; memoria virtual; entrada/salida)
DAXPY (*ver* procesador vectorial, benchmark Linpack)
DEC (*ver* Digital Equipment Corporation)
decimal codificado en binario (BCD), **117**
desempaquetado, **117**
empaquetado, **117**
decodificación de campo fijo, **216**
punto fijo, **652**, 701 (*ver también* aritmética, entera)

defectos por unidad de área, 63-64
definiciones, 769
DeLagi, B., 141
denormal, **655** (*ver también* aritmética, denormales y)
densidad, de bits constante, **654** (*ver también* disco, magnético)
área del disco, **558** (*ver también* máxima densidad de área; disco, magnético)
Densidad máxima de área de discos (MAD), 519, **558**, 605 (*ver también* disco, magnético)
desnivel semántico, 133, **138**
Dent, B. A., 137
dependencia, de datos verdadera, **402** (*ver también* procesador vectorial, dependencias de datos)
de salida, **402** (*ver también* procesador vectorial, dependencia de salida)
dependencias 283, 289 (fig.), 309 (*ver también* riesgos; procesador vectorial, dependencias de datos)
anti- (*ver* procesador vectorial, antidependencia)
entre iteraciones del bucle, **400** (*ver también* procesador vectorial, dependencias de datos)
dato verdadero (*ver* procesador vectorial,- dependencias de datos
de control (*ver* riesgos, saltos)
de datos (*ver* riesgos, datos)
tratamiento vectorial y, 403 (*ver también* procesador vectorial, dependencias de datos)
procesamiento vectorial y (*ver* procesador vectorial dependencias de datos)
salida (*ver* procesador vectorial, dependencia de salida)
desbloquear, **509** (fig.) (*ver también* cache, coherencia, sincronización)
desbordamiento, **647** (*ver también* aritmética, excepción, desbordamiento)
a cero (*ver* aritmética, excepciones, desbordamiento a cero)
gradual (*ver* aritmética, excepciones, desbordamiento)
ventana (*ver* registros de ventana)
aritmético (*ver* interrupciones, desbordamiento aritmético y)
de ventana, **486** (*ver también* ventanas de registro)
de desbordamiento a cero **486** (*ver también* ventanas de registro)
entero (*ver* aritmética, excepción, desbordamiento)
ventana (*ver* ventanas de registro)
descriptor de segmentos, **481** (*ver también* memoria virtual, tabla de página
descuento (*ver* coste)
desempaquetado, **655** (*ver también* decimal codificado binario, desempaquetado)
desenrollar el bucle, **339**, 348-350, 365
bucle desenrollado, 339-342, 343, 350
DLX superescalar y, 342-344
paralelismo incrementado a nivel de instrucción, 338-341
desplazamiento (*ver* jerarquía de memoria, bloque)
dentro de bloque, **435** (*ver también* jerarquía de memoria, bloques y)
sobre ceros, **682** (*ver también* aritmética, entero, aceleración de división, desplazamiento sobre ceros)
después de redondear, 661, **663** (*ver también* aritmético, redondean y)
desvío, **281**, 282, 315 (*ver también* adelantamiento)
detección, de riesgos de la segmentación (*ver* riesgo, detección)
dinámica de riesgos de memoria (*ver* riesgos, memoria, detección dinámica de)
detención, 226-**228** (*ver también* ciclos de detención de memoria)
segmentación
de escritura, **445** (*ver también* cache, escrituras y; cache, buffer de escritura; detenciones de escritura y)
en la segmentacion, 275-278, **285**-286, 298-299, 306, 312 (fig.)
máquinas vectoriales y, 378, 384-386
retardo de salto y, 293-299
riesgo de control y, 289-292, 290 (fig.)
detenciones de punto flotante, 312
DG (*ver* Data General)
Dhrystone, 30, 48 (*ver también* benchmarks, sintético)
diagrama de estados finitos **218**, 221
interrupciones y, 231, 232
para el DLX, 219, 235
diferencia de tiempo de acceso, 559 (fig.), **558**
dificultades en implementar la segmentación (*ver* segmentación, dificultades de implementación)
difusión en escritura, **505** (*ver también* cache, coherencia)
Digital Equipment Corporation (DEC), 15
DECstation, 21
3100, 73, 179, 204, 739-741
PDP-10, 100
PDP-11 Unibus y (*ver* bus, Unibus)
bus de, 574
PDP-11, 100, 112, 136-138, 141-142, 152, 201, 574, 604
PDP-8, 97
VAX, 27, 97, 100, 103, 108-110, 111-112, 131, 138-139, 150, 158, 181-185, 200-202
8550, 30
8600 FBox, 352-356
8600, 14, 30, 352-353, 361 (*ver también* segmentación, VAX 8600 y)
IBox, 352-356, 358

Digital Equipment Corporation (*cont.*)
IFetch, 354-363, 354
MBox, 352-355, 357
Opfetch, 353-359
EBox, 352-356
8700 frecuencia de conmutación de procesos en, 473 (fig.)
aritmética de punto flotante en, 703
códigos de condición, 158
especificadores de operando (*ver* Digital Equipment Corporation, VAX, modos de direccionamiento)
interrupciones, 230 (fig.), 233 (fig.), 234
mezclas de instrucciones, 184-185
modos de direccionamiento, 154-158, 156 (fig.), 182 (fig.)
uso, 181-184, 183
operaciones en el, 158
registros, 153-155
resumen de, 768
repertorio de instrucciones de usuario, 712-716
instrucciones aritméticas y lógicas enteras y de punto flotante, 711-712
instrucciones decimales y de cadena, 714-716
instrucciones de campo de bits de longitud variable, 716
instrucciones de cola, 716
instrucciones de salto, bifurcación y de llamada a procedimiento, 713-715
repertorio de instrucciones, 152-155, 156 (fig.), 158 (fig.) (*ver también* Digital Equipment Corporation, VAX, repertorio de instrucciones de usuario)
formato, 151, 154-156, 158
longitud de instrucción, 155 (fig.), 158 (fig.)
medida de uso, 150, 180, 181, 184, 199 (fig.), 728
tipos de datos, 153
VAX-11/780, 14, 21, 30, 31, **152,** 200-202
buffers de escritura en, 445, 447, 514-515, 521
buffer de prebúsqueda de instrucciones en, 485
buffer de traducción anticipada (*ver* memoria virtual, buffer de traducción anticipada)
cache en, 445-448 (*ver también* cache, VAX-11/780 y)
distribuciones de tiempo en, 734-735
entrada de tabla de páginas de (*ver* entrada de tabla de páginas)
espacio de direcciones, 475
jerarquía de memoria de (*ver* jerarquía de memoria, VAX-11/780 y)
memoria virtual en (*ver* memoria virtual, VAX-11/780 y)
VAX-11/785, 14
VAXstation 3100, 72
2000, 72
dirección (*ver también* modo de direccionamiento)
compartida frente a múltiple, 622-623
consumo de, 17, 771
de bloque, **435** (*ver también* jerarquía de memoria, bloques y; cache, dirección de estructura de bloque de)
de estructura (*ver* jerarquía de memoria, bloque)
del buffer de instrucciones virtuales (VIBA), **514** (*ver también* jerarquía de memoria, VAX-117780 y)
efectiva, **103**-105
espacio, 17, 20 (*ver también* cache; memoria; jerarquía de memoria; memoria virtual; memoria virtual, procesos y)
consecuencias de muy pequeño, 518-520
en el Intel 80286, 480-481
en el VAX-11/780, 475
extensiones de, 521
especificador, **109** (*ver también* modo de direccionamiento)
fallo, **467** (*ver también* memoria virtual, fallo de página)
memoria, 12, 19, 99, 101-110, 122-125, 144
traducción, **477** (*ver también* memoria virtual, traducción de direcciones)
direccionamiento, absoluto (*ver* modo de direccionamiento, directo)
diferido (*ver* modo de direccionamiento, indirecto de memoria)
directo (absoluto), **105** (*ver también* modo de direccionamiento)
escalado (índice), **105** (*ver también* modo de direccionamiento)
indirecto de memoria (memoria diferida), **105** (*ver también* modo de direccionamiento)
direcciones, físicas, **467** (*ver también* memoria virtual, traducción de direcciones)
virtuales **467** (*ver también* memoria virtual, traducción de direcciones)
director de almacenamiento, **592** (*ver también* disco, magnético, subsistema de almacenamiento IBM 3990 y)
disco, 6, 20, 21, 32 (*ver también* disco, magnético; disco, óptico)
almacenamiento, 3, 21
duro (*ver* disco, magnético)
magnético, 553-561, 605
array de (*ver* array de discos)
benchmarks de E/S, 548-551 (*ver también* entrada/salida, rendimiento)
para procesamiento de transacciones, 550-551
para sistema de ficheros, 550

disco (*cont.*)
para supercomputador, 548-550
para TP-1, 550, 551 (fig.)
búsquedas y, **555**, 600-603
fórmulas para, 600-601, 651 (fig.)
frente a distancia de búsqueda, 651 (fig.)
medidas de distancia de búsqueda, 602 (fig.)
tiempo medio de búsqueda, 555, 600, 607
capacidad de, 557 (fig.), 558, 590 (*ver también* máxima densidad de área)
características de, 554-556, 557 (fig.)
comparación de cuatro fabricantes, 557 (fig.)
coste de, 598-601
frente a tiempo de acceso, 559 (fig.)
diferencia de tiempo de acceso y, 559 (fig.), 558
discos de estado sólido (SSD), **558** (*ver también* memoria dinámica de acceso aleatorio)
frecuencia de datos de, 553 (fig.), 557 (fig.)
futuro de, 558-559, 605
memoria expandida (ES), **558**
organización de, 555 (fig.)
subsistema de almacenamiento IBM 3990, 589-598, 611
y buffers de coincidencia de velocidad de, 592
y cambios en tiempo de respuesta con mejoras en 3380D, 596 (fig.)
y director de almacenamiento de, 592
y IOCB de, 592
y seguir la pista de una lectura de disco, 592, 597
y programa de canal para, **592-593**
y canales y, 590, 591, 597
y resumen de, 596-598
y jerarquía de control, 590-592
y DLS y, 595-597
y cabeza de cadena, **592,** 595
y RPS, 595, 597
y DPR y, 595, 597
óptico, 558-561
compacto, **559** (*ver también* disco, óptico)
de una sola escritura, **560** (*ver también* disco, óptico)
una vez escrito, 560
regla de crecimiento, 19, 771
tecnología, 18
discos de estado sólido (SSD), **558** (*ver también* memoria dinámica de acceso aleatorio)
diseñador, computador (*ver* arquitecto, computador)
diseño
de alto rendimiento (*ver* diseño, alto rendimiento)
de bajo coste (*ver* diseño, bajo coste)
computador, 8, 13
alto rendimiento, 34
ayudado por computador, **624**
bajo coste, 36
complejidad y tiempo 15, 17
intercambios, 8, 16
tendencias y, 17-19
dispersar (scatter), **409** (*ver también* procesador vectorial, matrices dispersas y)
agrupar, **409** (*ver también* procesador vectorial, matrices dispersas y)
disponibilidad, **560** (*ver también* entrada/salida, fiabilidad)
dispositivos, de almacenamiento de acceso directo, **554** (*ver también* disco, magnético; entrada/salida)
inteligentes, **604** (*ver también* bus)
Ditzel, D. R., 139, 140, 202
división (*ver* aritmética, división, punto flotante; aritmética, división, entero)
con restauración (*ver* aritmética, división, entero, restaurar)
no restauración, **645** (*ver también* aritmética, no restauración)
SRT, **682** 683, 684, 693, 696, 697, 702 (*ver también* aritmética, entera, acelerar división, desplazamiento sobre ceros)
DLS (*ver* selección a nivel de dispositivo)
DLX, 126, 171-179, 192-197, 202
alineamiento, 236, 248
beneficios de ventanas de registros en, 489 (fig.)
bus, 214 (fig.), 215, 225, 246
camino de datos, 239
carga de byte, 246, 250
control (*ver* control, DLX y)
distribución en el tiempo de, 740-741
estados, 219, 236 (figs.), 239 (fig.), 240-242
frecuencias de fallos para, 520 (fig.)
máquinas relacionadas con, 180
mezclas de instrucciones, 193-197
procesamiento de vectores y (*ver* procesador vectorial, DLXV y)
registros, 172-174
repertorio de instrucciones, 172-180, 177 (fig.), 746-749
extensiones comunes a, 752-756
formato, 178 (fig.)
instrucciones aritmético-lógicas, 174-175
instrucciones de bifurcación, 237-241
instrucciones de carga y almacenamiento en, 172-175, 217
instrucciones de control de flujo, 175, 176 (fig.), 196
instrucciones de salto, 217, 239 (fig.), 247 (fig.), 247-255
medida de uso, 192-197, 194 (fig.), 199 (fig.), 732
resumen de, 744
segmentación (*ver* encauzada, DLX y)

DLX (*cont.*)
utilización de los modos de direccionamiento, 192-194
DLXV, 779 (*ver también* procesador vectorial, DLXV)
DMA (*ver* acceso directo a memoria)
virtual, **574** (*ver también* entrada/salida, DMA y)
doblamiento recursivo, **411** (*ver también* procesador vectorial, reducción vectorial)
doble, palabra, **101**
precisión (*ver* aritmética, precisión)
redondeo, **671** (*ver también* aritmética, redondeo)
Doherty, W., 604, 606
¿Dónde se puede ubicar un bloque? (*ver* ubicación de bloques)
DPR (*ver* reconocimiento dinámico de caminos)
DRAM (*ver* memoria dinámica de acceso aleatorio; memoria, DRAM)
DRAM de vídeo, **565** (*ver también* pantallas gráficas)
de DRAM (*ver* memoria de acceso aleatorio dinámica)
duración de ciclo (*ver también* duración de ciclo de reloj)
de reloj, 5, 38-44, 86, 213, 215, 243 (*ver también* frecuencia de reloj)
caches y, 448, 519
control y, 224, 242, 258
interrupciones y, **229**
máquinas segmentadas y, 269-274
de SRAM (*ver* memoria estática de acceso aleatorio)

E

EBCDIC, 117
EBox (*ver* Digital Equipment Corporation, VAX 8600)
Eckert, J. P., 25, 27, 258
Eckert-Mauchly Computer Corporation, 27
Eckhouse, R., 201
EDSAC (Electronic Delay Storage Automatic Calculator), 26, 258-260
EDVAC (Electronic Discrete Variable Automatic Computer), 25-26
efectividad de la planificación, 288, 296, 299
Eggers, S., 508, 526, 527
ejecución, 270, 316, 324, 353
fuera de orden, **341**-315, 321-323, 364 (*ver también* marcador; algoritmo de Tomasulo)
modo de, 8, 10, 31
segmentada, 270, 306, 316, 354
simulación, 311
ejemplo de coherencia en cache, 505 (fig.) (*ver también* cache, coherencia)
eliminación
de subexpresiones comunes, 123 (*ver también* optimización)
global, **120**, 123
de superficies ocultas, **566** (*ver también* pantallas gráficas)
de variables de inducción, **123** (*ver también* optimización)
Emer, J., 189
emisión (*ver* emisión de instrucciones)
de instrucciones múltiples, 341-344, 344-349, 365
planificación dinámica, 344-346
de más de una instrucción (*ver* emisión de instrucciones; superescalar)
emisiones, de frecuencia (*ver* repertorio de instrucciones, medidas)
de tiempo (*ver* repertorio de instrucciones, medidas)
emitida **286** (*ver también* emisión de instrucciones)
emitir en pareja (pares), 345, 365
emitir instrucción en orden, 313
empaquetado (*ver también* decimal codificado binario, empaquetado)
emulación, **259**
encadenamiento, **406** (*ver también* procesador vectorial, encadenamiento y)
encuesta, **575** (*ver también* entrada/salida, interfaz a la CPU)
Engelbart, D., 604
ENIAC (Electronic Numerical Integrator And Calculator), 25-26
entrada de tabla de páginas (PTE), **477** (*ver también* memoria virtual, tabla de páginas)
en el Intel 80286/80386, 479
en el VAX-11/780, 478, 512
entrada/salida (E/S), 6, 12, 15 (fig.), 19, 24, 537-539, 597-605 (*ver también* disco, magnético; pantallas gráficas; redes; bus)
ancho de banda (*ver* entrada/salida, productividad)
benchmarks (*ver* entrada/salida, rendimiento; disco, magnético, benchmarks de E/S para)
diseñar un sistema para, 581-589
dispositivos, 550-554, 603-605 (*ver también* disco magnético; pantallas gráficas; redes; bus)
categorizado por comportamiento, compañero, y frecuencia de datos, 552 (fig.)
ejemplos de, 552 (fig.)
frecuencia de datos, **550, 552**, 553
teclados, 552
DMA, 575-577, 579, 605
y virtual, 579, 579 (fig.)
falacias y pifias de, 597-603
fiabilidad, 560-562
gente, 546-548, 552, 604
y frecuencias máximas de E/S para, 552 (fig.)
y transacciones por hora frente a tiempo de respuesta del computador, 549 (fig.)
historia de, 603-605

entrada/salida (*cont.*)
IBM y, 589
importancia de (*ver* entrada/salida, rendimiento del sistema y)
interfaz a la CPU, 574-577 (*ver también* entrada/salida, DMA y)
falacia de transferir funciones desde la CPU a E/S, 598-600
relegar responsabilidades de E/S desde la CPU, 575-577
interfaz a un sistema operativo, 577-580
cache de disco, 577-580
efectividad de, 580 (fig.)
caches ayudando con, 578-580
caches provocando problemas con, 577-580
datos obsoletos y, 577-578
memoria virtual y, 579
latencia (*ver* entrada/salida, tiempo de respuestas)
productividad (anchura de banda), 545, 546, 586
array de discos y, 560
bus y, 574
disco magnético y, 546 (fig.)
frente a tiempo de respuesta, 546 (fig.), 545-548
pantallas gráficas y, 562-565
redes y, 569
rendimiento, 545-552, 581-589, 598-600 (*ver también* entrada/salida, tiempo de respuesta; entrada/salida, productividad; benchmark; disco, magnético, benchmarks de E/S para)
coste/rendimiento, 581, 598
del sistema, 539-545, 598-600
coste/rendimiento, 598
y ejecución solapada de E/S, 541 (fig.), 540-545
y fórmulas de tiempo para, 540-545
y Ley de Amdahl y, 538, 598, 603
modelo productor-servidor del tiempo de respuesta y productividad, 545 (fig.), 547 (fig.)
sistemas operativos y, 577 (*ver también* entrada/salida, interfaz a un sistema operativo)
solapamiento (*ver* entrada/salida, rendimiento del sistema y)
subsistema de memoria del IBM 3990 (*ver* disco, magnético, subsistema de almacenamiento del IBM 3990 y)
supercomputadores y, 570, 608
tiempo de CPU y, 537
tiempo de inactividad y, 538-539 (*ver también* entrada/salida, gente y)
tiempo de respuesta (latencia), 545, 546, 547 (fig.), 604
array de disco y, 560
disco magnético y, 546 (fig.)
frente a productividad, 546 (fig.), 545-548
frente a transacciones por hora, 549 (fig.)
IBM 3380D y, 596 (fig.)
pantallas gráficas y, 562-565
redes y, 569
tiempo de transacción y, 548
tipos (*ver* dispositivos de entrada/salida)
transacciones y, **548**, 547 (fig.)
transacción de usuario, 547 (fig.)
por hora frente a tiempo de respuesta, 549 (fig.)
y procesamiento de transacciones (TP), **550**-551
y tiempo de transacción, **546**
tiempo de asimilación (thinking), **548**, 547 (fig.), 604
tiempo de entrada, **548**, 547 (fig.), 604
tiempo de respuesta del sistema, **508** (*ver también* entrada/salida, rendimiento, tiempo de respuesta)
entrelazado, de memoria (*ver* memoria, entrelazada)
específico de DRAM para mejorar el rendimiento de la memoria principal, 464-466 (*ver también* memoria, entrelazada)
Equilibrio (*ver* Sequent Corporation)
(compromiso), 130, 145, 150-152, 236 (*ver también* diseño, computador; regla empírica de Case Amdahl)
segmentación
equilibrio entre etapas, 270
equilibrio en distribución, 343
software y hardware, 15-18, 22-23, 31
error de bus, 230 (fig.)
ES (*ver* memoria extendida)
ESA/370 (*ver* International Business Machines, IBM ESA/370)
escalabilidad, **619**, 630
escritura,
del resultado en un procesador segmentado, 316, 317 (fig.), 321 (fig.), 324, 326 (fig.), 327 (fig.), 329 (fig.), 330 (fig.), 357, 372
después de escritura (EDE), **284** (*ver también* riesgo, EDE)
después de lectura (WAR), **284** (*ver también* riesgo, WAR)
directa, **444** (*ver también* cache, escritura directa)
no ubicada en escritura, **445** (*ver también* cache, fallo de escritura)
Escuela Moore de la Universidad de Pennsylvania, 24-26
espacio, de direcciones local, **481** (*ver también* memoria virtual, procesos y)
global de direcciones, **481** (*ver también* memoria virtual, procesos y)
espacios de direcciones privadas múltiples, **622**
especificador de operando, 354-358
(*ver* modo de direccionamiento)
espiar, **505** (*ver también* cache, coherencia)

espionaje, **504** (*ver también* cache, coherencia)
esquemas de predicción,
de saltos, 293-298, 331-338, 364-366 (*ver también* predicción dinámica de saltos por hardware; penalización por predicción errónea)
precisión de la predicción, 311-333, 336
predecir no efectivo, **293**, 298 (fig.), 332, 334-336
predicción de efectivo, 293-296, 298 (fig.), 331-332, 334-336, 355-356
reducción de las penalizaciones de salto mediante predicción dinámica por hardware, 328-338
esquemas para predicción de saltos (*ver* esquemas de predicción de saltos)
estación de trabajo, 537-538, 604
computador personal frente, 538
coste de, 66, 67, 92
DECstation 3100 (*ver* Digital Equipment Corporation, DECstation 3100)
minicomputador frente, 537
servidor de ficheros frente, 538
SPARCstation I (*ver* SPARC)
VAXstation 2000 (*ver* Digital Equipment Corporation, VAXstation 2000)
VAXstation 3100 (*ver* Digital Equipment Corporation, VAXstation 3100)
estaciones de reserva, **322**-331, 344
estados, **215**, 218-221 (*ver también* diagrama de estados finitos)
ciclos de reloj y, 239-241, 243
de espera, **239**
DLX y, 219, 236 (fig.), 237 (fig.), 238 (fig.), 239 (fig.), 241
interrupciones y, 231, 233-235
PLA y, 220-221
estándar de punto flotante, 117
estándares
bus (*ver* bus, estándares)
estrategia de escritura, **438**, 523
caches y, 443-447, 505 (*ver también* cache, escrituras y)
memoria virtual y, 470
estrategia para escrituras (*ver* estrategia de escritura)
ETA-10 (*ver* Control Data Corporation, ETA-10)
Etapa de segmentación, **269-271**, 273-275, 306
Ethernet **567** (*ver también* redes)
etiqueta de identificación de proceso (PID), **495** (*ver también* cache)
evaluación de rendimiento vectorial (*ver* rendimiento vectorial, analizar)
excepción de desbordamiento a cero (*ver* aritmética, expresiones, desbordamiento a cero)
excepción de inexactitud, **672** (*ver también* aritmética, excepciones)
excepciones, **231** (*ver también* aritmética, excepciones; interrupciones)
expansión escalar, **411** (*ver también* procesador vectorial, reducción vectorial y)
exponente, **654** (*ver* aritmética, exponentes y)
polarizado, 654, 656 (*ver también* aritmética, exponentes y)
exponentes no polarizados, 654 (*ver también* aritmética, exponentes y)
extensión constante de arquitecturas RISC (*ver* computador de repertorio reducido de instrucciones)
E/S (*ver* entrada/salida)
mapeada en memoria, **574** (*ver también* entrada/salida, interfaz a la CPU)
controlada por interrupciones, **575** (*ver también* entrada/salida, interfaz a la CPU)

F

Fabry, R., 523, 527
factor de entrelazado, **463** (*ver también* memoria entrelazada)
fallo, **435** (*ver también* cache, fallo; memoria virtual, fallo de página; jerarquía de memoria, fallo)
de capacidad, **452** (*ver también* cache, fallos, capacidad)
de conflicto, **452** (*ver también* cache, fallo, conflicto)
de página, 20, **467** (*ver también* memoria virtual, fallo de página; interrupciones, fallo de página y)
segmentación y, 299-304
RPS **595** (*ver también* disco, magnético, subsistema de almacenamiento IBM 3990 y)
fallos, **231** (*ver también* interrupciones)
de arranque frío, **452** (*ver también* cache, fallo, compulsorio (forzoso)
de colisión, **452** (*ver también* cache, fallo, conflicto)
de primera referencia, **452** (*ver también* cache, fallo, forzoso)
por instrucción, **448** (*ver también* cache, fallo)
fase, **120** (*ver también* paso)
FBox (*ver* Digital Equipment Corporation, VAX 8600)
fiabilidad, **560** (*ver también* entrada/salida, fiabilidad)
fichero, de futuro, **310** (*ver también* completitud fuera de orden)
de historia, **310** (*ver también* terminación fuera de orden)
FIFO (*ver* reemplazo de bloques, primero en entrar primero en salir)
firmware (*ver* microprogramación)
Fisher, J., 365

Flemming, P. J., 83
Floating-Point Systems AP-120B, 365
FLOPS (*ver* punto flotante, operaciones de punto flotante por segundo)
Foley, P., 205
forma aproximada, **663**-664
formato, de dos operandos, 99
  de instrucciones (*ver* sintaxis de instrucciones)
  de microinstrucción, 222-223
    en DLX (*ver* control, DLX y)
    reducir costes de hardware con formatos múltiples de microinstrucción, 225-226
  de punto flotante (*ver* aritmética, estándar del IEE y)
  de tres operandos, 99-101
FORTRAN, 128, 140
  Absoft System V88 2.0a compilador, 88
  compilador F77, 135-136
  FORTRAN 77, 526
  FORTRAN 8X, 526
forzoso, **552** (*ver también* cache, fallo, compulsorio (forzoso))
Foster, C. C., 139
FP (*ver* punto flotante)
FPD (*ver* primera parte hecha) en VAX
fracción, tiempo de computación, 10
  mejorado, 10
fragmentación, interna, **471** (*ver* memoria virtual, tamaño de página y)
  y reensamblamiento, **568** (*ver también* redes)
frecuencia, de aciertos, **435** (*ver también* jerarquía de memoria, frecuencia de aciertos; cache, acierto)
  de datos, **550** (*ver también* entrada/salida, dispositivos)
  de E/S, **552** (*ver también* entrada/salida, transiciones y)
  de fallos, **435** (*ver también* cache, fallos)
    de escritura, 448 (*ver también* cache, fallo de escritura)
    de lectura, 448 (*ver también* cache, lecturas y; cache, fallo)
    locales, **497** (*ver* cache, fallo; cache, cache de dos niveles)
  de refresco, **561** (*ver también* pantallas gráficas)
  de reloj, 38-40, 44, 72, 145, 244 (*ver también* duración de ciclo de reloj), 76, 89
  global de fallos, **497** (*ver* cache, fallo; cache, caches de dos niveles)
  sostenida (*ver* procesador vectorial, frecuencia sostenida)
Freitas, D., 204
fuera de orden
  terminación (*ver* terminación fuera de orden)
  ejecución (*ver* terminación fuera de orden)
  interrupciones, 301-304
Fuller, S. F., 84, 85
FutureBus, 573, 573 (fig.) (*ver también* bus)

G

Gagliardi, U. O., 138
Gajksi, D., 634
Garner, R., 204
GCD (*ver* máximo común divisor)
Gelsinger, P., 202
generación, computador, 27
generar, **674** (*ver también* aritmética; acarreo)
Gibson, D. H., 82
Gibson, J. C., 82, 85
gigaflop (*ver* punto flotante, millones de operaciones de punto flotante por segundo)
Gill, J., 141
Gill, S., 25
Goldstine, H. H., 25-27
Gonter, R. H., 141
Goodman, J., 526, 527
Gottlieb, A., 634
GPR (*ver* registro, arquitectura de registros de propósito general)
grafo de interferencias **121**
Gran Endian, contraportada anterior, **101**
Gross, T. R., 360, 364
guardar registros, en llamador, **116**-117
  en rutina llamada, **116**-117, 133-135

H

Hansen, C., 204
hardware, **14** (*ver también* equilibrio, software y hardware)
  crecimiento industrial y, 23
  «lo más pequeño es más rápido», 19
Hauck, E. A., 137
Henly, M., 85
Hennessy, J. L., 140, 203
Hewlett-Packard
  Precisión, 179, 203
Hilfinger, P., 204
Hill, M., 453, 457, 519, 524-526, 527
Hillis, D., 621, 634, 635
  con Bell, 635
historia, computador, 24-29
Hitachi S810/20, 79
Hopkins, M. E., 141
Hough, D., 204
HP (*ver* Hewlett-Packard)
Hudson, E., 204

hueco de retardo, **287** (*ver también* hueco de retardo de salto; hueco de retardo de carga)
huecos (*ver* huecos de retardo de salto; retardo de carga)
de retardo de salto, **295**-297, 300, 359
lleno, 296
planificación, 295 (fig.), 297 (fig.), 370
vacío, 296
llenos (*ver* huecos de retardo de salto)
útiles, 296 (*ver también* huecos de retardo de salto)
vacíos (*ver* huecos de retardo)

I

i860 (*ver* Intel Corporation i860)
IAS (Institute for Advanced Study) (*ver* Universidad de Princeton)
IBM (*ver* International Business Machines Corporation)
IBox (*ver* Digital Equipment Corporation, VAX 8600)
IC (*ver* circuito integrado)
identificación, bloque (*ver* identificación de bloques)
de bloque, **438**, 523
caches virtuales y, 495-497
caches y, 440-442
memoria virtual y, 469-471
identificación de datos, 331, 364
IEEE (*ver* aritmética, estándar del IEEE y)
Ifetch (*ver* Digital Equipment Corporation, VAX 8600)
Illiac IV, 598, 599 (fig.), 617, 634, 636
IMP (*ver* procesador de mensajes de interfaz; redes)
implementación, 13
evaluación de rendimiento y, 83-85
hardware, 16, 23
software, 16
tecnología de, 17
incremento de consumo de dirección, contraportada anterior, 17, 518-520
independencia de posición, **113**
índice, 50, 105
infinito, 654, 655, 660 (fig.), 663, 672, 703 (*ver también* aritmética, redondeo y; no un número)
negativo (*ver* infinito)
positivo (*ver* infinito)
precisión infinita, 664
Instituto de Estudios Avanzados de la Universidad de Princeton (IAS), 26
Instituto para Estudios Avanzados (IAS) (*ver* Princeton University Institute for Advanced Study)
instrucción (*ver también* repertorio de instrucciones)
arquitectura (*ver* repertorio de instrucciones, arquitectura)
Calls, **131**, 132-135, 146-147, 228
codificación, 100, 109-111
comprometida, **301**
control (*ver* control; instrucciones de control de flujo)
de cargas interbloqueadas (*ver* interbloqueo de carga)
de comparación y salto, 113-**114**
de intercambio atómico (*ver* transferencia de datos)
de salto probable, **758** (*ver también* salto retardado)
de TLB, **756**
densidad, **100**
emisión, **186**, 307-311, 314-319, 322-328, 364-365 (*ver también* emisión en pareja, emisión de múltiples instrucciones)
detenciones y, 306
de múltiples instrucciones con clasificación dinámica, 344-349
de más de una instrucción, 341-344
máquinas superescalares y, 341-344
marcador y, 314-319, 314 (fig.)
estado, 318, 317 (fig.), 321 (fig.), 325, 328 (fig.), 330 (fig.)
estática, 13
formato (*ver* repertorio de instrucciones)
de arquitecturas RISC, 745
frecuencia de búsqueda y decodificación, 377
frecuencias (*ver* repertorio de instrucciones, medidas)
interrupción y rearranque, 300-304, 309-311, 356
longitud de camino, 38 (*ver también* cuenta de instrucciones)
medidas (*ver* repertorio de instrucciones, medidas)
mezcla, 41, 48, 78, 82
paralelismo, 337-352, 365-366 (*ver también* procesador vectorial)
incrementar con desenrollamiento de bucle, 338-342
incrementar con segmentación software y planificación de trazas, 348-352
paso de multiplicación, **652** (*ver también* aritmética)
planificación, **287**-288, 293-299
recuento, 38-45, 77-78, 106, 133
optimización y, 127-130
referencia, 131
repertorio (*ver* repertorio de instrucciones)
sintaxis, 151 (*ver también* repertorio de instrucciones, arquitectura)
sobre campos de bits en Motorola 88000, 761-762
tamaño, 110
tiempo medio de ejecución, 82
instrucciones,
aritméticas y lógicas, 92
en arquitecturas RISC, 747-748

instrucciones (*cont.*)
operaciones del coprocesador, 754, **755**
de control de flujo, 111-114, 131 (*ver también* DLX, repertorio de instrucciones, instrucciones de control de flujo)
en arquitecturas RISC, 749
de punto flotante (*ver* operaciones de punto flotante)
enteras y de punto flotante solapadas, 277
gráficas en Intel 860, 764
sobre pixels en el Intel 8600 (*ver* instrucciones gráficas)
integración de procedimiento, **120**, 122-124
integridad de datos, **560** (*ver también* entrada/salida, fiabilidad)
Intel Corporation
Intel 4004 y 8008, 201
Intel 432, 135
Intel 8080, 164, 201
Intel 8088, 202
Intel 80x86, contraportada anterior, 165, 202, 484
Intel 80186, 165, 202
Intel 80286 protección sobre (*ver* memoria virtual, Intel 80286/80386 y)
Intel 80286 memoria virtual en (*ver* memoria virtual, Intel 80286/80386 y)
Intel 80286 puertas de llamada en (*ver* puerta de llamada)
Intel 80286, 165, 202, 480-481, 483-485
Intel 80286 tabla de descriptores, **481** (*ver también* memoria virtual, tabla de páginas; memoria virtual, Intel 80286/80386 y)
Intel 80386 memoria virtual en(*ver* memoria virtual, Intel 80286/80386 y)
Intel 80386 protección sobre (*ver* memoria virtual, Intel 80286/80386 y)
Intel 80386, 165, 202
Intel 80486, 60, 62, 89, 165, 202
Intel 8086, 97, 103, 112, 151, 164-172, 189-193, 201, 480
codificación postbyte, 171 (fig.)
defectos, 197
distribución de tiempo en, 737-739
espacio de direcciones, 165
interrupciones, 230 (fig.)
mezcla de instrucciones, 189-192
modo de compatibilidad, 165
modos de direccionamiento, 165-168
registros, 164-167, 166 (fig.)
repertorio de instrucciones, 164-172, 169 (fig.), 721-725
formatos, 151, 171, 169 (fig.), 171 (fig.)
instrucciones aritméticas y lógicas, 722
instrucciones de control, 723-724
instrucciones de transferencia de datos, 724-725
medidas de uso, 168 (fig.), 180, 189-193, 199 (fig.), 731
modos de direccionamiento utilización, 190 (fig.)
operaciones sobre, 168-172
resumen de, 768
Intel 860, 89, 204, 365, 531, 744
repertorio de instrucciones, 747-749
extensiones comunes a instrucciones DLX, 753-755
único, 763-768
Intel i860, 531 (*ver también* Intel Corporation, Intel 860)
Intel multicomputador Intel Hypercube, 634
interbloqueo (*ver* interbloqueo en la segmentación, riesgos, datos; interbloqueo de carga)
de carga, 287, 289
en la arquitectura MIPS II, **758**
de la segmentación, **285**-287, 364 (*ver también* interbloqueo de carga)
DLX y, 287-288
interfaces de periféricos inteligentes (IPI), **574**, 573 (fig.), 604 (*ver también* bus)
interfaz de sistemas de pequeños computadores (SCSI), 15, 573 (fig.), **604**-605 (*ver también* bus)
International Business Machines Corp. (IBM), 15, 27, 83, 85
definición de MIPS y, 78
Extensión (*ver* International Business Machines Corp., IBM 7030)
IBM PC-AT, 574
IBM 370-XA, 159, 200-201
IBM 360, 17, 27, 82, 97, 100, 112, 136-138, 159-164, 184-189, 199-201, 259, 523
defectos, 196-198
interrupciones, 230 (fig.), 234-236
mezclas de instrucciones, 188-190
operaciones sobre, 162-164
registros, 151, 160-162, 187, 190-191
repertorio de instrucciones, 159-162, 162 (fig.), 163-164 (fig.), 717-721
almacenamiento inmediato (SI), 162, 187 (fig.)
instrucciones de formato RS y SI, 720
formatos, 161-164
utilización, 187 (fig.)
medida de utilización, 184-189, 198 (fig.), 199 (fig.), 729-731
tesis de Shustek sobre, 184-186, 198, 200
memoria-memoria (SS), 162, 186, 190
instrucciones de formato SS, 721
modos de direccionamiento, 160-162
utilización, 185-187
registro-almacenamiento (RS), 160-162, 187

International Business Machines (*cont.*)
instrucciones de formato RS y SI, 720
registro-indexado (RX), 160-162, 187
saltos y cargas y almacenamientos especiales formato RX, 720
instrucciones enteras/lógicas y de punto flotante formato RX, 718-719
registro-registro (RR), 160-162, 187
instrucciones R-R, de inicialización de status y saltos, 718-719
y enteras/lógicas y de punto flotante, 717-718
resumen de 768
IBM 370 sistema de punto flotante sobre, 702
IBM 7030, 111, 363
IBM 7090, 139, 259
subsistema de almacenamiento IBM 3990 (*ver* disco, magnético, subsistema de almacenamiento IBM 3990)
IBM 360/85, 28, 85, 525
IBM 360 IBM 360/85 (*ver* International Business Machines Corp., IBM 360/85)
IBM Sistema/360 (*ver* International Business Machines Corp., IBM 360)
IBM (7030), 82
multiprocesador IBM RP3, 634 (*ver también* multiprocesador)
IBM 704, 363
IBM 360/IBM 370 (*ver* International Business Machines Corp., IBM 360; International Business Machines Corp., IBM 370)
IBM 3090, 590
almacenamiento (*ver* disco, magnético, subsistema de almacenamiento IBM 3990 y)
IBM 3090-600S, 80
IBM Sistema/370 (*ver* international Business Machines Corp., IBM 370)
IBM 801, 202, 204
IBM 370, 159, 200, 424, 522-523 (*ver también* International Business Machines Corp., IBM 360)
IBM 370/168, 734-738
IBM PC, 36, 189, 197, 202, 738-739
bus de, 574
IBM 370/158, 83
compilador IBM PL.8, 140
IBM RP-PC, 100, 203
IBM 701, 27, 28
IBM ESA/370, 159
IBM 360/91, 321-323, 364
interrupción imprecisa (*ver* interrupciones, imprecisas)
interrupciones, **229**-235
8600 y, 356-359
cómo controlar comprobaciones para interrupciones, 232-233
comparación en cuatro computadores, 230 (fig.)
DLX y, 245, 251, 254
desbordamiento aritmético y, 229-230, 232 (fig.), 233 (fig.), 258
excepción de invalidez, **672** (*ver también* aritmética, excepciones)
fallos de página y, 230, 232 (fig.), 233 (fig.)
historia de, 258
imprecisas, 308-310
memoria virtual y, 474
precisas, **301**, 359, 364
qué es difícil sobre las interrupciones, 233-235
segmentación y, 280, 297, 299-304 (*ver también* interrupciones, imprecisas; interrupciones, precisas)
invalidación en escritura, **505** (*ver también* cache, coherencia)
IOCB (*ver* bloque de control de E/S)
Iowa State University, 26
IPI (*ver* interfaz de periférico inteligente; bus)
ISP (procesador del repertorio de instrucciones) (*ver* repertorio de instrucciones, arquitectura)
iteraciones de bucles solapadas, 330
iteración de Newton, 664-666, 666-668

J

jerarquía, de almacenamiento (*ver* jerarquía de memoria)
de memoria, **20**, 19-22, 24, 31, 32, 432, 433-438, 532 (fig.) (*ver también* cache; cache, caches de dos niveles; memoria; memoria virtual; memoria virtual, buffer de traducción anticipada; identificación de bloque; ubicación de bloque; reemplazo de bloque; estrategia de escritura; buffer de prebúsqueda de instrucciones; ventanas de registros)
jerarquía de memoria, 436 (fig.), 456 (fig.)
acierto, **435** (*ver también* cache, acierto)
bloques y, **434**-438 (*ver también* cache, bloques y; memoria, bloque; memoria virtual, paginada; bloque)
castigos y, 454
desplazamiento dentro de bloque, **435**
dirección de bloque, **435**
falacias y pifias de, 518-522
fallo, **435** (*ver también* cache, fallo; cache, fallo de escritura; memoria virtual, fallo de página)
frecuencia de aciertos, **435** (*ver también* cache, acierto)
historia de, 523-526
implicaciones de, a CPU, 437

jerarquía de memoria (*cont.*)
nivel inferior, **435** (*ver también* cache; memoria virtual; memoria)
nivel superior, **435**-438 (*ver también* memoria; cache; memoria virtual)
niveles (*ver* jerarquía de memoria; nivel inferior; cache; memoria; memoria virtual)
penalización de fallo y tamaño de bloque
principio de localidad y, **433**-435, 522 (*ver también* localidad)
localidad espacial, **433**, 437, 525 (*ver también* localidad)
datos compartidos y, 505-506
tamaño de bloque de cache y, 455, 494, 502 (*ver también* cache, bloques y)
localidad temporal, **433**, 437, 525 (*ver también* localidad)
datos compartidos y, 505-506
menos recientemente usado y, **442** (*ver también* reemplazo del bloque menos recientemente usado)
traducción de direcciones y, 472 (*ver también* memoria virtual, buffer de traducción anticipada)
relación de cache a, 439 (*ver también* cache)
relación de memoria principal a, 458 (*ver también* memoria)
relación de memoria virtual con, 466 (*ver también* memoria virtual)
rendimiento, 435-438, 524 (*ver* cache, rendimiento; memoria virtual, rendimiento)
resumen de ejemplos de, 523 (fig.)
tamaño de bloque fijo, 434, 437, 468
tamaño de bloque variable, 434, 468 (*ver también* memoria virtual, segmentada)
tiempo de acceso, **435**-437, 458-460 (*ver también* cache, tiempo de acceso)
tiempo medio de acceso a memoria, 436, 437 (*ver también* cache, tiempo de acceso; cache, caches de dos niveles)
VAX-11/780, 512-518 (*ver también* cache, VAX-11/780 y; memoria virtual, VAX-11/780 y)
y buffers de escritura, **445**, 447, 514, 521
y dirección del buffer de instrucciones virtuales (VIBA), **514**
y dirección del buffer de instrucciones físicas (PIBA), **514**
y fallos por cien instrucciones para el TLB del VAX-11/780, 517 (fig.)
y frecuencia de fallos para, frente a DLX, 520 (fig.)
el TLB de la VAX-11/780, 517 (fig.)
y número medio de ciclos de reloj por instrucción del 780, 515
y visión global del, 513 (fig.)
Jouppi, N., 141
Joy, B., 204

K

Kahn, R., 605, 606
Kane, G., 204
Katz, R., 526, 527
Kelisky, R. P., 604
Kilburn, T., 29, 466, 522, 527
Kleiman, S., 204
Knuth, D. E., 29
Kuck, D., 636
Kung, H. T., 635

L

LAN (*ver* red de área local; redes)
Lanigan, M. J., 29
Larus, J., 204
latencia, **5**, 19 (*ver también* tiempo de ejecución, rendimiento)
E/S (*ver* entrada/salida, rendimiento, tiempo de respuesta)
de acceso, **435** (*ver también* jerarquía de memoria, tiempo de acceso; cache, tiempo de acceso)
de E/S (*ver* entrada/salida, rendimiento, tiempo de respuesta)
de rotación, **556** (*ver también* disco, magnético)
duración de ciclo de, 458-459
latencia de E/S (*ver* entrada/salida, rendimiento, tiempo de respuesta)
medida de rendimiento de memoria principal y, 487
productividad y, 8
tiempo de acceso, 458-459
latidos (*ver* ciclos de reloj)
LDE (*ver* lectura después de escritura)
LDL (*ver* lectura después de lectura)
lectura–, después de escritura (LDE), **284** (*ver también* riesgo, RAW)
después de lectura (RAR), **284**
lecturas de palabra, no alineada, **757**
Lee, R., 204
lenguaje
alto nivel (*ver* lenguaje de alto nivel)
de alto nivel, 18, **118**, 122-125, 130, 133, 139, 141, 144
de descripción, 151-152, contraportada posterior
ensamblador, 18
programación, 18
Levy, H., 184, 201

Ley de Amdahl, 8-12, 23, 28, 31, 619-621, 631 (*ver también* regla empírica Case/Amdahl)
diferencia de rendimiento CPU-DRAM y, 459, 460 (fig.), 465
E/S y, 538, 598, 603
límite, **474** (*ver también* memoria virtual, esquemas de protección de)
limpio, **444** (*ver también* cache, escritura)
línea, **439** (*ver también* cache, bloques y)
Linpack (*ver* procesador vectorial, benchmark de Linpack; benchmarks) 30, 48
LISP, soporte para en SPARC, 758-760
LIW (*ver* palabra larga de instrucción)
llamada (*ver* llamada/vuelta de procedimiento)
llamadas de seguridad del usuario a puertas OS, 483 (*ver también* memoria virtual, Intel 80286/80386 y)
llamada/retornos de procedimiento, 86, 110, 113, 116-117, 123, 124, 147
falacias y pifias, 133-135
llamador, 133-135
localidad (*ver también* jerarquía de memoria, principio de localidad y)
de referencia, 11-13, 19, 20
espacial, 13 (fig.), 31, **433**
(*ver también* localidad, jerarquía de memoria, principio de localidad y; localidad)
principio de (regla de localidad 90/10), contraportada anterior, 11-13, **433**
programa, 29
temporal, 12, **433** (*ver también* localidad; jerarquía de memoria, principio de localidad y)
lógica
operaciones, 15
tecnología, 18 (fig.)
longitud de vector (*ver* procesador vectorial, longitud de vector)
longitud máxima del vector (MVL), **392** (*ver también* procesador vectorial, longitud de vector)
longitud vectorial (*ver* procesador vectorial, longitud vectorial)
LRU (*ver* reemplazo de bloque, menos recientemente usado)
Lunde, A., 139

M

M680x0 (*ver* Motorola Corporation)
M88000 (*ver* Motorola Corporation, 88000)
macro-, 222
MAD (*ver* máxima densidad de área)
Manchester, Universidad de (*ver* Universidad de Manchester)
mapa de bits, **562** (*ver también* pantallas gráficas)
mapa de colores, **564** (*ver también* pantallas graficas)
máquina, cache, 359 (*ver también* cache)
de registros vectoriales, **379** (*ver también* procesador vectorial, máquinas vectoriales)
direccionada por bytes, 101
Multiflujo, 365
vectorial memoria-memoria, **379** (*ver también* procesador vectorial, máquinas vectoriales)
máquinas,
de Cray Research, 36, 379, 419, 422
aritmética en, 703
CRAY X-MP, 79-80, 86, 379, 404-405, 420, 421, 532
CRAY Y-MP, 379, 420-422, 323
CRAY-1, 379, 405, 419, 421, 422
CRAY-2, 46, 379, 405 (fig.)
de segmentación, 379
superescalares, **341** (*ver también* superescalar)
marcador 313-322, 371, 427-428
componentes de, 320 (fig.)
detección de riesgos, 316 (*ver también* riesgo, detección)
emisión de instrucciones, 317 (fig.)
planificación dinámica en torno a riesgos con un marcador, 420-428
tablas, 317 (fig.), 321 (fig.)
marcaje, **315** (*ver también* marcador)
margen, bruto, 68-71, 81, 90
Mark-I, -II, -III, -IV (Universidad de Harvard), 26-27
Markstein, J., 140
Markstein, P. W., 141
Mark I (Universidad de Manchester), 26
más infinito (*ver* infinito)
matrices vacías (*ver* procesador vectorial, matrices vacías y)
Mauchly, J., 25-27, 258
máximo común divisor (GCD), **401** (*ver también* procesador vectorial, dependencias de datos)
Mazor, S., 204
MBox (*ver* Digital Equipment Corporation, VAX 8600)
McFarland, H., 141
McKeeman, W. M., 138
McMahon, F. M., 83, 86
McNamara, J. E., 85
McNutt, B., 83
media
aritmética, 53, 56, 73, 83
ponderada, 54
armónica (*ver* media, armónica)
geométrica (*ver* media, geométrica)
medias, palabras, **101**

nedias (*cont.*)
ponderadas (*ver* media)
dinámicas, **149** (*ver también* repertorio de instrucciones, medidas)
estáticas, **149** (*ver también* repertorio de instrucciones, medidas)
nedidas, de utilización del repertorio de instrucciones (*ver* repertorio de instrucciones, medidas)
detalladas (*ver* repertorio de instrucciones, medidas)
dinámicas (*ver* repertorio de instrucciones, medidas, dinámicas)
estáticas (*ver* repertorio de instrucciones, medidas, estáticas)
megaFLOPS (*ver* MFLOPS)
megaherzio (*ver* frecuencia de reloj)
mejorar rendimiento de procesadores vectoriales (*ver* procesador vectorial, mejorar rendimiento)
nemoria, 5, 14, 15 (*ver también* ancho de banda cache; memoria dinámica de acceso aleatorio; memoria estática de acceso aleatorio, jerarquía de memoria; memoria virtual; identificación de bloques; ubicación de bloques; reemplazo de bloques; estrategia de escritura)
principal (*ver* memoria, principal)
ancho de banda, 275, 279, 347, 353 (*ver también* memoria, organización de)
en máquinas vectoriales, 387-391, 421
arquitectura, memoria-memoria (*ver* arquitectura memoria-memoria)
registro-memoria (*ver* arquitectura registro-memoria)
bancos, **388**-391 (*ver también* memoria, entrelazada)
bus, 14, 15, 20 (fig.), 31
celda, 19
centralizada, **622**
frente a distribuida, 622-624
compartida (*ver* memoria virtual, compartida; memoria virtual, Intel 80286/80386 y)
consistencia, 511 (*ver también* cache, coherencia)
débil, 512
secuencial, 511
correspondencia, **467** (*ver* memoria virtual, traducción de direcciones; memoria virtual, Intel 80286/80386 y)
de control, 223, **224**, 226-228, 250, 256 (*ver también* memoria escribible de control)
de sólo lectura (ROM), **220**, 223, 256, 258-260
futuro de la microprogramación y, 258
programable (PROM), 67
detenión de ciclo, **239**
caches y, 447-448
de reloj, **239**
dinámica de acceso aleatorio (DRAM), 17, 18, 31, 458-461, 464-466 (*ver también* memoria estática de acceso aleatorio; memoria, DRAM; memoria virtual)
de acceso aleatorio (DRAM), 460 (figs.)
capacidad de, 459, 464
columna estática, **465**, 525
coste de, 598-601
frente a tiempo de acceso para, 559 (fig.)
disco de estado sólido y, **558**, 608
entrelazado y, 464-466
regla de crecimiento, 18
tiempo de, 460 (fig.)
de ciclo de, 459, 466 (fig.)
vídeo, 564 (*ver también* pantallas gráficas)
direccionamiento indirecto diferido de memoria, **104** (*ver también* modo de direccionamiento)
DRAM y, 17, 18, 31, 458-461 (*ver también* memoria dinámica de acceso aleatorio)
coste de refresco, 459
entrelazado, 464-466 (*ver también* memoria, entrelazada)
entrelazada, 462-465
desventaja de, 464
DRAM específica, 464-466
factor de entrelazado, **463**
estática de acceso aleatorio, 459, 465 (*ver también* memoria dinámica de acceso aleatorio; memoria)
capacidad de, 459
coste frente a tiempo de acceso de, 559 (fig.)
duración del ciclo de, 459
expandida (ES), **558**
E/S mapeada en memoria, **574** (*ver también* entrada/salida, interfaz a la CPU)
jerarquía (*ver* jerarquía de memoria)
latencia, duración de ciclo, de, **458**-459 (*ver también* latencia)
tiempo de acceso a, **458**-459 (*ver también* latencia; jerarquía de memoria, tiempo de acceso)
máquina vectorial memoria-memoria, **379** (*ver también* procesador vectorial, máquinas vectoriales)
más ancha, 461-463
modo de direccionamiento diferido (*ver* modo de direccionamiento, indirecto de memoria)
núcleo magnético, 27, 458
organización de, 461, 461 (fig.) (*ver también* memoria, entrelazada; memoria, más ancha)
principal, 20-22, 26, 458-466
ancho de banda, 458 (*ver también* ancho de banda)
latencia, 458 (*ver también* latencia)
más ancha (*ver* memoria, más ancha)
referencia, 99-104, 118, 124-129, 131, 139, 144, 279, 283

memoria (*cont.*)
CDC 6600 y, 315
calculada, **125**
guarda/restaura, **125**, 125-129
IBM 360/91, 323-326
rendimiento, 522
diferencia de rendimiento CPU-DRAM, 459, 460 (fig.), 466
incrementar con entrelazado específico de DRAM, 464-466
riesgos (*ver* riesgo)
segmentación, 278 (*ver también* segmentación; retardo de carga; buffers de carga y almacenamiento)
software y, 17
sólo lectura (ROM), **220**, 223, 256, 258-260
futuro de la microprogramación y, 258
virtual, 14, 20, 28, 137, 139, **466**-535, 523 (fig.) (*ver también* cache; memoria; jerarquía de memoria; identificación de bloque; ubicación de bloque; reemplazo de bloque; estrategia de escritura)
bits de modificación y, 471, 472
bloque (*ver* memoria virtual, página; memoria virtual, segmento)
buffer de traducción anticipada (TLB), **472**-473, 523 (fig.)
en la VAX11/780, 478 (fig.), 478-480, 512
fallos por cien instrucciones en el VAX-11/780, 517
frecuencia de fallos del TLB, debido al flujo de instrucciones, 516
frecuencias de fallos para el TLB del VAX-11/780, 517
parámetros típicos de, 472 (fig.) (*ver también* parámetros, rangos típicos de)
resumen de, 523 (fig.)
caches y, 468, 472
compartida, 467, 479-481
datos obsoletos y (*ver* datos obsoletos)
diferencias entre caches y, 468
DMA y, 579
escrituras y (*ver* estrategia de escritura, memoria virtual y)
esquemas de protección, **466**-467, 473-476, 477, 480-485 (*ver también* memoria virtual, Intel 80286/80386 y, protección en)
de anillos de niveles de seguridad, **475**
de caballos de Troya y, **480**, 482
de protección de sólo lectura, 473-474
de registro base, **474**
de registro límite, **474**
fallo de página, **467**, 468, 470 (*ver también* cache, fallo «tres C»)
frente a segmentación, 468, 469 (fig.), 475
información de bloque (*ver* información de bloque, memoria virtual)
Intel 80286/80386 y, 479-485
y campo de atributos, **481**
y comprobación de límites sobre, 481
y compartición de, 481-482
y correspondencia de memoria, 480
y descriptor de segmento de, 482 (fig.)
y entrada de tabla de páginas de, 481
y llamadas seguras de usuarios a puertas del OS, 483
y protección en, 480, 483-485
página o segmento, **467, 468**
fragmentación interna y, **471**
tamaño de página, 471
parámetros típicos, 478 (fig.) (*ver también* parámetros, rangos típicos de)
penalización de fallos, 468
procesos, 472-474 (*ver también* memoria virtual, esquemas de protección)
y espacio de direcciones, 466
y usuario, procesos núcleo y supervisor, **474**
recubrimientos, **467**
reemplazo de bloque (*ver* reubicación de bloque, memoria virtual)
resumen de, 523 (fig.)
reubicación y, **467**
segmento o página, **467, 468**
falacia de, 521
frente a paginación, 468, 469 (fig.), 475-576
tabla de páginas, **469**, 471
conservación de memoria con, 476-478
entrada de tabla de páginas (PTE) en la VAX-11/780, 477, 512
entrada de tabla de páginas/descriptor de segmentos del Intel 80286/80386, 481
traducción de direcciones, **467**, 469-470 (fig.), 475, 476-478, 495 (*ver también* memoria virtual, buffer de traducción anticipada)
en el VAX-11/780, 476-478
técnicas para traducción rápida de direcciones, 471-473
ubicación de bloque (*ver* ubicación de bloque, memoria virtual)
VAX-11/780, 475-480, 483-485 (*ver también* cache, VAX-11/780 y; jerarquía de memoria, VAX-11/780 y)
y área P0, 475
y área P1, 475
y entrada de tabla de páginas (PTE) de la VAX-11/780, 477, 512
y fallos por cien instrucciones en la VAX11/780, 517

memoria (*cont.*)
y frecuencia de fallos para el TLB de la VAX-11/780, 517
y funcionamiento del TLB del VAX11/780, 479 (fig.)
y parámetros típicos de, 478 (fig.) (*ver también* parámetros, rangos típicos de)
y segmentos de proceso de, 475
menos, infinito (*ver* infinito)
recientemente usado (LRU), **442** (*ver también* reemplazo de bloques, menos recientemente usado)
mercado (*ver* mercado, computador)
computador
efecto sobre el diseño, 5, 13-14, 15
Metcalfe, R., 603, 606
método, de conmutación de paquetes, **568** (*ver también* redes)
de marcador (*ver* marcador)
métrica, computador, 16, 19
mezcla de Gibson, 82, 85
MFLOPS (*ver* punto flotante, millones de operaciones en punto flotante por segundo)
Mhz (megaherzio) (*ver* frecuencia de reloj)
micro-, 222
microarquitectura (*ver* organización)
microcódigo, 222, 228 (*ver también* control, microprogramado/microcodificado; microprograma)
comparado a macrocódigo, 255
horizontal, **226**, 229, 262
estado legal de, como programa, 260
vertical, **226**, 262
microcomputador, 3-4
microinstrucción, 222, 244 (*ver también* microcódigo; cotrol, microprogramado/microcodificado)
horizontal (*ver* microcódigo, horizontal)
vertical (*ver* microcódigo, vertical)
microprocesador, 3-4, 17, 80 (*ver también* Cypress Corporation; Intel Corporation; MIPS Computer Corporation; Motorola Corporation; National 32032 microprocesador)
comparación de, 201-204
Intel 80x86 (*ver* Intel Corporation, 80x86)
MIPS R2000 (*ver* MIPS Computer Corporation)
MIPS R3000 (*ver* MIPS Computer Corporation)
Motorola 680x0 (*ver* Motorola Corporation)
Motorola 88000 (*ver* Motorola Corporation)
National 32032, 583
SPARC (*ver* SPARC)
«super-», 538
microprograma, 222, 226 (*ver también* microcódigo; control, microprogramado/microcodificado)
contador, 244
estructura de, 222
horizontal, **226**, 229, 262
memoria de microprograma (*ver* almacenamiento de control)
microprograma DLX (*ver* control, DLX y)
estado legal de, como programa, 260
vertical, **226**, 262
microprogramación, **222**, 222-223
ABC de la microprogramación, 223-225 (*ver también* control, microprogramado/microcodificado)
millones de instrucciones por segundo (MIPS), 19, 43-45, 47, 71
nativas, 45, 76, 85
relativos, 45, 76, 82-84
millones de operaciones en punto flotante por segundo (MFLOPS) (*ver* punto flotante, millones de operaciones en punto flotante por segundo)
MIMD, débilmente acoplado (*ver* multicomputador)
fuertemente acoplado (*ver* multiprocesador)
minicomputador, 3-4
frente a computador grande, 537
frente a estación de trabajo, 537
PDP-11 (*ver* Digital Equipment Corporation, PDP-11)
VAX 8700 (*ver* Digital Equipment Corporation, VAX 8700)
VAX-11/780 (*ver* Digital Equipment Corporation, VAX-11/780)
VAX-8600 (*ver* Digital Equipment Corporation, VAX 8600)
MIPS (*ver también* Stanford MIPS)
(*ver* millones de instrucciones por segundo)
MIPS Computer Systems, Inc., 43, 44, 72, 100, 203, 364
arquitectura MIPS II, 758
MIPS R2000, 112, 179, 192, 202-204, 311, 424
MIPS R3000, 89, 179, 192, 202-204, 311, 531, 744
repertorio de instrucciones, 747-749
extensiones comunes a instrucciones DLX, 753-755
única, 755-758
MIPS R3010, 672, 696 (fig.), 697, 747-749 (*ver también* MIPS Computer Systems, Inc., MIPS R3000)
MIPS de Stanford, 203 (*ver también* MIPS Computer Systems Inc.)
MIT (Massachusetts Institute of Technology), 26-27
modelo, para rendimiento vectorial (*ver* procesador vectorial, rendimiento, modelo para)
productor-servidor (*ver* entrada/salida, rendimiento)
modificación, **444** (*ver también* cache, escritura)
modo de direccionamiento, 103-111, 135, 143, 146 (*ver también* Digital Equipment Corporation, VAX; DLX; Intel Corporation, Intel 80x86, 8086;

modo de direccionamiento (*cont.*)
International Business Machines Corp., IBM 360)
autodecremento, **105**
autoincremento, **105**
basado (*ver* modo de direccionamiento, desplazamiento)
codificación de, 109-111
de arquitecturas RISC, 744
desplazamiento (registro base), 105-107, 113-115, 123, 142-143
(*ver* modo de direccionamiento)
campo, 107, 109-111, 113 (fig.), 114 (fig.)
tamaño, 107
valor, 107
diferido de memoria (*ver* modo de direccionamiento, indirecto de memoria)
directo (absoluto), **105**
escalado (índice), **105**, 135
especificadores de operando, 155 (fig), 182 (fig.), 186 (fig.), 190 (fig.), 193 (fig.)
indexado, **105**, 146
(*ver también* modo de direccionamiento)
índice (*ver* modo de direccionamiento, escalado)
indirecto (*ver también* modo de direccionamiento, registro diferido; modo de direccionamiento, indirecto de memoria)
de memoria (diferido de memoria), **105**-106
inmediato (literal), **105**-110
(*ver* modo de direccionamiento)
campo, 110
valor, 108-109
literal (*ver* modo de direccionamiento, inmediato)
registro diferido (indirecto), **105**, 146
modo de instrucción dual
modo de página, para DRAM, **465** (*ver también* memoria, DRAM)
rápido de DRAM, **466**
modo en Intel 860, 766
modo nibble, **465** (*ver también* memoria, DRAM)
modo segmentado, **765**
en Intel 860, 764-767
modo sistema, **474** (*ver también* memoria virtual, esquemas de protección de)
modos de redondeo, **654** (*ver también* aritmética, redondear y)
Morse, S., 201
MOS, 63
Motorola Corporation
68000, 100, 202
arquitectura de, 768
interrupciones en, 230
6809, 97
88000, 179, 203, 533
arquitectura de, 744E-2
repertorio de instrucciones, 747-749
extensiones comunes a instrucciones DLX, 753-755
único, 761-763
88100, 89, 530
88200, 531
compilador C88000 1.8.4m14 C, 88
Moussouris, J., 203
MOVC3, 234, 262-264
fragmento de código, **122** (*ver también* optimización)
MTTF (*ver* tiempo medio entre fallos)
MTTR (*ver* tiempo medio de reparación)
Muchnik, S., 204
Mudge, J. C., 85
Multibus II, 574, 573 (fig.) (*ver también* bus)
multicomputador, 634, 638-639
basado en Transputer, 634
Cosmic Cube, 634
múltiples operaciones por instrucción, 346-349, 365
multiplicación (*ver* aritmética, multiplicación, punto flotante; aritmética, entero)
de base superior, **686**, 693 (*ver también* aritmética, entera, acelerar multiplicación)
y división enteras
con signo y sin signo en SPARC, **761**
en arquitecturas RISC, 751-753
multiplicador, en árbol binario, **690** (*ver también* aritmética)
en array, **686**, 687 (figs.), 689 (fig.), 691 (fig.), 697 (*ver también* aritmética, entero, aceleración de la multiplicación)
par/impar, **689** (*ver también* aritmética)
multiprocesador, 77-78, **618**-620, 626, 635, 638-639
C.mmp, 634 (*ver también* multiprocesador)
caches en (*ver* cache, coherencia)
Encore Multimax, 634 (*ver también* multiprocesador)
escrituras y, 505 (*ver también* cache, coherencia)
frecuencia de fallos, 505 (*ver también* cache, coherencia)
medir rendimiento de, 629-631
multiprocesador C.mmp, 634
multiprocesador Cm*, 634
multiprocesador Encore Multimax, 634
multiprocesador IBM RP3, 634
multiprocesador Symmetry, 626-630, 634
MVL (*ver* longitud máxima de vector; procesador vectorial, longitud de vector)

N

Namjoo, M., 204
NaN (*ver* no un número)
nano-, **262**

nanocódigo, 262-263
nanoinstrucción, 262-263
nivel, de conmutación de memoria de procesador (*ver* organización)
de protección requerido, **483** (*ver también* memoria virtual, esquemas de protección de; memoria virtual, Intel 80286/80386 y)
superior, **434** (*ver también* jerarquía de memoria; cache; memoria; memoria virtual)
no efectivo, **290** (*ver también* salto, no efectivo)
no operación (NOP), 529
frecuencias de fallos de Spice con y sin, 529 (fig.)
no ubicar en escritura, **445** (*ver también* cache, fallo de escritura)
no un número (NaN), **653**-655, 672 (*ver también* aritmética)
Noonan, R., 141
NOP (*ver* no operación)
Nova (*ver* Data General)
NuBus, 15, **604** (*ver también* bus)
números con signo (*ver* aritmética, con signo)

O

oblea, 59-61, 63
chip por, 63, 89
coste de, 63-64, 65
dados por, 63, 65-66
fotografías de, 60-61
rendimiento, 63-64, 65-66, 89, 90
ocupado-espera (*ver* espera circular)
O'Laughlin, J., 141
opción Endian (*ver* transferencia de datos)
operación, de desempaquetado, 117
de empaquetado, 117
de intercambio atómico, 509 (*ver también* cache, coherencia, sincronización)
operaciones, **110**
aritméticas, 111 (*ver también* aritmética; repertorio de instrucciones)
de bloque/desbloqueo (*ver* sincronización)
de cadena, 16
operaciones de punto flotante, 16, 111, 305-312, 341-343 (*ver también* segmentación, DLX y, punto flotante)
conversiones implícitas 754, **755**
en arquitecturas RISC, 749
en solapado, en SPARC, 760
por segundo (*ver* punto flotante, operaciones de punto flotante por segundo)
decimales, 16, **111** (*ver también* aritmética, decimal)
del coprocesador (*ver* instrucciones aritméticas y lógicas)
enteras, 16
multiciclo, 305
DLX y, 305-311
operador, de punto flotante, **111**
de transferencia de datos, **111**
operadores (*ver* operaciones)
aritméticos y lógicos, 111
de cadena, 111
de control, **111**
del sistema, **111**
operando
denominación de, 96-99
tipo y tamaño, 117-119
optimización
alto nivel, 120, **122**
dependiente de la máquina, **122-124**
global, 122, **122-124**, 141
local, **122-124**
orden principal de fila, **394**, 395 (fig.)
ordenación, de bytes, 101-102 (*ver también* Pequeño «Endian» y Gran «Endian»)
por columnas, **394**, 395 (fig.)
rápida (Quicksort) (*ver* programas de referencia, reducido)
organización, **14** (*ver también* memoria, organizaciones de; jerarquía de memoria)
para mejorar el rendimiento de memoria principal (*ver* memoria, organizaciones de)
efecto sobre el tiempo de diseño, 17

P

P1, 475 (*ver también* memoria virtual, VAX-11/780 y)
Padegs, A., 200
página, 20, **467, 468** (*ver también* memoria virtual, página; dirección, memoria)
palabra, **102**
de instrucción muy larga (VLIW), 341, **346**, 345-349, 362-363, 617, 624
instrucciones, 346
planificación de trazas y, 350
larga de instrucción (LIW), **346**, 365
en Intel 860, 766-768
pantalla (*ver* tubo de rayos catódicos; pantallas gráficas)
pantallas, de escala de gris, **561** (*ver también* pantallas gráficas)
de trama de tubos de rayos catódicos (CRT), **561** (*ver también* pantallas gráficas)
gráficas, 561-566, 565 (fig.), 604, 605
buffer de estructura, 561-562, 562 (fig.)
coste de, 563-565
demandas de rendimiento de, 564-566
direcciones futuras en, 565-567

pantallas (*cont.*)
DRAM de vídeo, 564-566
eliminación de superficies ocultas, **566**
eliminación de superficies ocultas aproximación del buffer-z a, **566**
mapa de colores, **564**, 564 (fig.)
tareas y sus requerimientos de rendimiento
paquete (*ver también* coste)
coste de, 59, 63-65, 90
diseño y, 58
paquetes, **567** (*ver también* redes)
paralelismo (*ver también* instrucción, paralelismo)
a nivel de instrucción (*ver* instrucción, paralelismo)
de datos **617**
en segmentación, 270, 337
paralelismo a nivel de instrucción y segmentación, 337-352, 365-366 (*ver también* instrucción, paralelismo)
parámetros, rangos típicos de
buffers de traducción anticipada, 472 (fig.)
cache, 439 (fig.)
compañero, **552** (*ver también* entrada/salida, dispositivos)
memoria virtual, 467 (fig.)
TLB del VAX-11/780, 478 (fig.)
parámetros típicos (*ver* parámetros, rangos típicos de)
pasarela de, **568** (*ver también* redes)
paso, 119, **120**, 122
Patterson, David, 140, 202, 204
PC (*ver* contador de programa, salto)
(personal computer) (*ver* Intel Corporation, 80x86; Intel Corporation 8088; International Business Machines Corp., IBM PC)
PDP (*ver* Digital Equipment Corporation)
penalización, de fallo de predicción, 297-298, 333-334, 334 (fig.), 334-336, 351-352 (*ver también* esquemas de predicción de saltos)
de fallos, **435** (*ver también* jerarquía de memoria, fallo; jerarquía de memoria, bloque; cache, fallos; memoria virtual, penalización de fallos)
por predicción errónea (*ver* penalización de predicción errónea)
Pendleton, J., 204
Pequeño Endian, contraportada anterior, 101
periférico, 537 (*ver también* entrada/salida, dispositivos; disco, magnético; pantallas gráficas; redes; bus)
Pfister, G. F., 634
Phister, M., 86
PIBA (*ver* buffer de dirección física de instrucciones)
PID, **495** (*ver también* etiqueta identificadora de proceso)
pila, 105, 123, 124-127, 133-135, 136, 141, 144 (*ver también* arquitectura de pila)
arquitectura de, 133-134
reducción de altura, **123** (*ver también* optimización)
pistas, **554** (*ver también* disco, magnético)
Pitkowsky, S. H., 85
pixels, **562** (*ver también* pantallas gráficas)
PLA (*ver* array lógico programable)
planificación, 288 (*ver también* salto, planificación, planificación de retardo de salto; planificación dinámica; planificación de instrucciones; planificación en la segmentación)
de trazas, **346, 350**, 348-352, 365
VLIW y, 350
del hueco de retardo de los saltos (*ver* hueco de retardo de los saltos)
dinámica, **313**, 312-336, 344-346, 364-365
algoritmo de Tomasulo (*ver* algoritmo de Tomasulo)
distribución de múltiples instrucciones y, 344-346
método de marcador (*ver* marcador)
reducción de penalizaciones de salto con predicción dinámica por hardware, 328-337 (*ver también* esquemas de predicción de salto; predicción dinámica de saltos por hardware)
en la segmentación , **123**, 128, **287**-288, 338-341, 364 (*ver también* optimización; planificación dinámica)
estática, 287, 293-296, **312-313**, 338-341 (*ver también* planificación dinámica)
frente a algoritmo de Tomasulo, 328-331
frente a planificación dinámica, 344, 365, 374
PO, 475 (*ver también* memoria virtual, VAX-11/780 y)
pocesador con memoria compartida, **618**-620, 622-624, 634, 636, 637
Pohlma, W., 204
postescritura, **444** (*ver también* cache, postescritura)
memoria virtual y, 471
postcopia (copy back), **444** (*ver también* cache, postescritura)
precio (*ver* coste)
del sistema empaquetado (*ver* coste)
medio (*ver* coste)
precisión (*ver* aritmética, precisión)
(*ver* Hewlett-Packard, Precisión)
de la predicción (*ver* esquemas de predicción de saltos, precisión de la predicción)
doble extendida aritmética, precisión)
predecir rendimiento del sistema (*ver* entrada/salida)
predicción, de dos bits, **331**-333 (*ver también* esquemas de predicción de saltos)
de efectividad (*ver* esquemas de predicción de saltos, predicción de efectividad)
de no efectividad (*ver* esquemas de predicción de saltos, predicción no realizada)

predicción (*cont.*)
de saltos (*ver* esquemas de predicción de saltos)
por hardware, **313** (*ver también* predicción dinámica de saltos por hardware)
dinámica de saltos (*ver* predicción dinámica de saltos por hardware)
de saltos por hardware, 328-337, 364-365 (*ver también* esquemas de predicción de saltos)
preguntas para clasificar jerarquías de memoria (*ver* identificación de bloques; ubicación de bloques; reemplazo de bloques; estrategia de escritura)
primera parte hecha (FPD), **234**-235
primero en entrar primero en salir (FIFO), **443** (*ver también* reemplazo de bloques, primero entrar primero salir)
primitiva, 130
principio de localidad, **433** (*ver también* localidad; jerarquía de memoria, principio de localidad y)
problema, de asignación de estados, **221**
de coherencia cache, **502** (*ver también* cache, coherencia)
de ordenación de fases, **119**-120
procedimiento en línea (*ver* integración de procedimiento)
procesador, **213**, 226 (*ver también* unidad central de tratamiento)
camino de datos y, 215
computación y, 215
control y, 215, 218, 228
de E/S **576** (*ver también* entrada/salida, interfaz con la CPU)
de interfaz de mensajes (IMP), **568** (*ver también* redes)
de propósito especial (*ver* procesador, propósito especial)
de repertorio de instrucciones (ISP) (*ver* repertorio de instrucciones, arquitectura)
digital de señales (DSP), **624**
en array (*ver* computador de flujo simple de instrucciones y flujo múltiple de datos)
propósito especial, **624**
superescalar, **363** (*ver también* superescalar)
vectorial, 377-430
antidependencias, **402**-404
arquitectura, 379-385
bancos de memoria y, **388**-391
conflictos de bancos de memoria, 396
compiladores y, 399-405 (fig.) (*ver también* riesgo)
componente, 379-381
control de máscara vectorial, **407**
cuello de botella de Flynn y, **377**
DAXPY (*ver* procesador vectorial, programa de referencia de Linpack)
dependencias (*ver* procesador vectorial, antidependencias; procesador vectorial, dependencias de datos; procesador vectorial, dependencias de salida)
de datos, 387, 399-405, 424 (*ver también* antidependencias; dependencias de salida)
dependencias entre iteraciones por bucle, **400**-401
dependencias verdaderas de datos, **402**
matrices dispersas y, 408-409, 411
recurrencia, **401**
riesgo RAW, 402 (*ver también* procesador vectorial, dependencias verdaderas de datos)
(*ver también* dependencia de salida)
riesgo WAR, 402 (*ver también* antidependencia)
test GCD, 401
de Banerjee, 402
de salida, **402**
DLXV, 379-391, 412-419
frecuencia de iniciación, 385
instrucciones vectoriales, 383 (fig.)
longitud vectorial y, 391-393
tiempo de arranque de, 385, 388 (fig.)
separación y, 396
efectividad (*ver* procesador vectorial, rendimiento)
encadenamiento y, 405-**406**
falacias y pifias de, 419-421
velocidad de terminación, **385**
iniciación, **385**-391
benchmark de Linpack, 381-382
bucle DAXPY, 381-384
bucle SAXPY, 381, 387, 384, 417
en FORTRAN, 392
encadenamiento y, 407
velocidad sostenida, 387, **407**, 414, 415-417
supercomputadores japoneses y, 419
máquinas escalares y, 422
historia de, 422-424
longitud vectorial, 391-394, 412
registros de longitud vectorial (VLR), **392**
longitud vectorial máxima (MVL), **392**, 407
máquinas vectoriales, 24, **378**-380, 382 (fig.), 419-424, 626
máquina vectorial memoria-memoria, **379**, 419-420, 422
máquina vectorial con registros, **379**, 392
tiempos de arranque y, 419
matrices dispersas, 408-411
y dispersar-agrupar, **409**, 410
y dispersar, **409**, 423
y vector de índices, **409**, 410-411
y agrupar, **409**, 423
máximo (*ver* procesador vectorial, rendimiento, rendimiento máximo)

procesador (*cont.*)
mejorar rendimiento, 405-411, 417-419
por reducción vectorial, 410
con segmentación múltiple de memoria, 417-419
con sentencias ejecutadas condicionalmente y matrices dispersas, 407-411
por encadenamiento, 405-408
mod numero banco, 389
reducción (*ver* procesador vectorial, reducción vectorial)
vectorial y expansión escalar, **411**
vectorial (*ver* procesador vectorial)
reducción vectorial, **410**-412
y doblamiento recursivo, **411**-412
registro mediante máscara vectorial, **407**-409
registros, 379, 380
registro de longitud vectorial (*ver* procesador vectorial, longitud vectorial)
registro de máscara vectorial (*ver* procesador vectorial, registro de máscara vectorial)
renombrar, 402-404
vector, 380 (fig.)
rendimiento, 403-405
análisis, 396-399, 412-419
ancho de banda de memoria y, 421
comparación de rendimiento escalar, 420-421
evaluación (*ver* procesador virtual, rendimiento, analizar)
medidas relativas a la longitud, 412
mejora (*ver* procesador vectorial, mejorar rendimiento)
modelo de, 396-399
rendimiento sostenido, 415-417
rendimiento máximo, 413-415, 419-420
rendimiento de SAXPY, 417-419
SAXPY (*ver* procesador vectorial, benchmark Linpack)
sentencias ejecutadas condicionalmente y, 407-411
solapamiento, 387, 417-419
tiempo de arranque, **384**-388, 419
máquinas vectoriales primitivas y, 419
penalizaciones de arranque en el DLXV, 388 (fig.)
separación, **394**, 393-397
no unitaria, **394**, 421
unidades funcionales, 380
ventaja, 371
procesamiento
de transacciones (TP), 16-17, **550** (*ver también* entrada/salida, transacción; disco, magnético, benchmarks de E/S para)
paralelo, 24, 28 (*ver también* paralelismo)
secuencial, 28 (*ver* procesamiento, secuencial)
proceso, **473** (*ver también* memoria virtual, procesos y)
ejecutivo, **474** (*ver también* memoria virtual, procesos y)
núcleo, **474** (*ver también* memoria virtual, procesos y)
supervisor, **474** (*ver también* memoria virtual, procesos y)
productividad, 5-7, 23
de E/S (*ver* entrada/salida, rendimiento, productividad)
de la segmentación (*ver* segmentación, aceleración)
E/S, 538-540 (*ver también* entrada/salida, rendimiento, productividad)
y cauce (*ver también* segmentación)
latencia y, 8
profundidad (*ver* segmentación, profundidad de la segmentación)
programa
benchmark (*ver* benchmark)
comportamiento (*ver* buffer prebúsqueda de instrucciones; ventanas de registro)
de canal, **593** (*ver también* disco, magnético, subsistema de memoria IBM 3990 y)
de computador, **261**
Spice, 13, 46, 48, 71, 73, 75, 77, 84, 88, 91
típico, 196
programas, de benchmark (*ver* benchmark)
de núcleo (kernel), 46, 48-51, 82
Livermore FORTRAN, 46, 82, 86
PROM (*ver* memoria de sólo lectura programable)
propagación, **674**-675 (*ver también* sumador con propagación de arrastre, aritmética)
constante, **123** (*ver también* optimización)
de copias, **123** (*ver también* optimización)
propiedad de inclusión multinivel, **501** (*ver también* cache, cache de dos niveles)
protección, **466** (*ver también* memoria virtual, esquemas de protección de; memoria virtual, Intel 80286/80386 y)
de sólo lectura, 474-475 (*ver también* memoria virtual, esquemas de protección)
protocolos
coherencia (*ver* cache, coherencia)
en cache, **504** (*ver también* cache, coherencia)
multiprocesadores y (*ver* cache, coherencia)
redes y, **568** (*ver también* redes)
proyecto, Cedar de la Universidad de Illinois, 634
SYMBOL, 138, 141
Przybylski, S., 204
PTE (*ver* entrada de tabla de páginas)
puente, **568** (*ver también* redes)
puerta de llamada, **483** (*ver también* memoria virtual, esquemas de protección de; memoria virtual, Intel 80286/80386 y)

punto, de polución, **436** (*ver también* jerarquía de memoria, bloque; cache, bloques y)
flotante (FP), 15, 20 (*ver también* aritmética, punto flotante)
aritmética (*ver* aritmética, punto flotante)
CDC 6600 y, 313-316
IBM 360/91 y, 321-323
millones de operaciones de punto flotante por segundo (MFLOPS) normalizado, 45-47, 88
(MFLOPS), 45-47, 78-80, 83, 88, 91, 412, 415 (*ver también* procesador vectorial, rendimiento)
(MFLOPS) nominal, 45-47, 86, 88
(FLOPS), 387-388 (*ver también* procesador vectorial, rendimiento)
desbordamiento (*ver* aritmética, excepción)
Puzzle (*ver* benchmark, reducido)

Q

Q1 (*ver* ubicación de bloques)
Q2 (*ver* identificación de bloques)
Q3 (*ver* reemplazo de bloques)
Q4 (*ver* estrategia de escritura)
¿Qué bloque se debería reemplazar en un fallo? (*ver* reemplazo de bloque)
¿Qué ocurre en una escritura? (*ver* estrategia de escritura)

R

Radin, G., 202
RAID (arrays redundantes de discos baratos) (*ver* array de discos)
raíz cuadrada (*ver* aritmética, raíz cuadrada)
rangos, de parámetros (*ver* parámetros, rangos típicos de)
vivos, **121**
RAS (*ver* strobe de acceso-fila)
Ravenal, B., 204
recomenzable, **233**-235, 257, 300-304
rearranque anticipado, **498** (*ver también* cache, fallo)
reconocimiento dinámico de caminos (DPR), **596** (*ver también* disco, magnético, subsistemas de almacenamiento IBM 3990 y)
recurrencia, **401** (*ver también* procesador vectorial, dependencias de datos)
recursos
arquitecturas VLIW y, 346
segmentación y, 273-276, 309
emplear, 8, 11
red, de alineación, 102-104, 145
terminal, **567** (*ver también* redes)
redes, 15 (fig.), 566-569
ARPANET, 568, 568 (fig.), 605
de área local (LAN), **567** (*ver también* redes)
de larga distaneia, **568** (*ver también* redes)
Ethernet, **567**, 568, 603
jerarquía de, 569 (fig.)
rango de características, 567 (fig.)
red de área local (LAN), **567**-569, 568 (fig.)
Rs232, 567, 568 (fig.)
Redmond, K. C., 26
redondear (*ver* aritmética, redondear y)
reducción, **410** (*ver también* procesador vectorial, reducción de vectores y)
de potencia, **123** (*ver también* optimización)
de las penalizaciones debidas a saltos (*ver* esquemas de predicción de salto)
vectorial (*ver* procesador vectorial, reducción vectorial y)
redundancia, **684** (*ver también* aritmética, entera, acelerar divisón desplazamiento sobre ceros)
referencias, de datos, 131-134, 142-143
de punto flotante, 128
reflejo, **561**
refresco, 459 (*ver también* memoria, DRAM)
refresco de exploración, **562** (*ver también* pantallas gráficas)
registro, 20, 21, 23, 96-101
almacenamiento de registros (RS) (*ver* International Business Machines Corp., IBM 360, repertorio de instrucciones)
arquitectura, de registros de propósito general (GRP), **97**-101, 136-138
memoria-registro (*ver* arquitectura memoria-registro)
registro-registro (*ver* arquitectura registro-registro)
caches frente, velocidad de, 521
campo, 109-111
comparación de, 99-101
de enteros, 122, 125-128, 132, 146
de longitud vectorial (VLR), **392** (*ver también* procesador vectorial, longitud vectorial)
de máscara vectorial, **407** (*ver también* procesador vectorial, registro de máscara vectorial)
de punto flotante, 122, 126-128, 132
DEC VAX, 153-155
direccionamiento registro diferido (indirecto), **105** (*ver también* modo de direccionamiento)
DLX, 172-174
etiquetas, 325-328
fichero, 347
IBM 360, 159-162
inicializar, 97, 126-128

registro (*cont.*)
Intel 8086, 164-167, 166 (fig.)
longitud de vector (*ver* procesador vectorial, longitud de vector)
máquina (*ver* registro, arquitectura de registros de propósito general)
máscara de vector (*ver* procesador vectorial, registros de máscara de vector)
registro indexado (RX) (*ver* International Business Machines Corp., IBM 360, repertorio de instrucciones)
registro-registro (RR) (*ver* International Business Machines Corp., IBM 360, repertorio de instrucciones)
renombrar, 364, 365
antidependencias y dependencias de salida y, 402-404
riesgo (*ver* riesgo, registro)
sombra (*ver* registros de sombra)
estado del resultado, 317 (fig.), 319, 320 (fig.), 325-326
ubicación, 115-117, 120-155, 122-128, 140
vector (*ver* procesador vectorial, registros)
ventanas (*ver* ventanas de registros)
registros, de sombra, **263**
vectoriales (*ver* procesador vectorial, registros)
regla, de cache dos a uno, contraportada anterior
de crecimiento, de disco, contraportada anterior, 19
de DRAM,
empírica, de Amdahl, 459 (*ver también* regla empírica Case/Amdahl)
de Case/Amdahl, contraportada anterior, **19**, 459 (*ver también* equilibrio, software y hardware; reglas empíricas; rendimiento)
diferencia de rendimiento CPU-DRAM y, 459, 460 (fig.), 465
noventa/diez (*ver* localidad, principio de)
reglas, de crecimiento (*ver* disco, regla de crecimiento; memoria dinámica de acceso aleatorio, regla de crecimiento)
empíricas, contraportada anterior (*ver también* regla empírica de Case/Amdahl)
frecuencia de crecimiento del disco,
incremento del consumo de direcciones,
regla, de Amdahl/Case de,
de cache 2:1,
de crecimiento de DRAM,
de localidad 90/10,
de salto realizado 90/50
relación de tráfico, 529
reloj, 38
período (*ver* ciclo de reloj)
pulsación (*ver* ciclo de reloj)
REM, 666-670, 697 (*ver también* aritmética, resto)
Remington-Rand Corporation, 27
rendimiento, **5-8, 37**-38, 38-43, 57-59, 76, 85, 86, 540 (*ver también* dado; circuito integrado; oblea; entrada/salida, rendimiento del sistema y; ancho de banda; coste/rendimiento; latencia; tiempo de respuesta; productividad)
«más lento que», 7
«más rápido que», 6-7, 30
cache (*ver* cache, rendimiento)
coste y, 23, 28, 36
CPU y, 11, 17 (*ver también* unidad central de proceso, rendimiento)
crecimiento de, 3, 4 (fig.), 5, 6, 23, 30
del sistema, **37** (*ver también* rendimiento; entrada/salida, rendimiento del sistema y)
entrada/salida (*ver* entrada/salida, rendimiento)
ideal de la segmentación, 276-278
jerarquía de memoria (*ver* jerarquía de memoria, rendimiento; memoria, rendimiento; cache, rendimiento; memoria virtual, rendimiento)
Ley de Amdahl y, 8-12
localidad de referencia y, 19, 20 (*ver también* localidad)
máximo, 76, 78-80 (*ver* procesador vectorial, rendimiento, rendimiento máximo)
mejora del rendimiento de la segmentación (*ver* segmentación, DLX y, rendimiento de)
mejora, 540-544 (*ver también* entrada/salida, rendimiento del sistema y)
memoria virtual (*ver* memoria virtual, rendimiento)
pantalla de gráfica (*ver* pantallas de gráficas, demandas de rendimiento de)
procesador vectorial (*ver* procesador vectorial, rendimiento)
requerimientos de diseño y, 13-19
sistema, **37** (*ver también* entrada/salida, rendimiento del sistema y)
restos y, 68
sostenido (*ver* procesador vectorial, rendimiento, rendimiento sostenido)
test final, 59, 64-66
ventaja del rendimiento RISC (*ver* computador de repertorio reducido de instrucciones, ventaja de rendimiento de)
repertorio de instrucciones (*ver también* instrucción; DLX; Intel Corporation, 860; MIPS Computer Corporation, R3000; Motorola Corporation, 88000)
arquitectura, 13, 17, 18, 39, 96-101
comparación, 75
complicaciones (*ver* segmentación, dificultades en implementación, complicaciones del repertorio de instrucciones)

repertorio de instrucciones (*cont.*)
control (*ver* control; instrucciones de control de flujo)
frecuencias de instrucciones DLX, 194 (fig.)
Intel 8086, 191 (fig.)
frecuencias (*ver* repertorio de instrucciones, frecuencia de instrucciones; repertorio de instrucciones, medidas)
de instrucciones (*ver también* repertorio de instrucciones, medidas)
DEC VAX, 185 (fig.)
IBM 360, 188 (fig.)
medidas, 149-151, 152-153, 179-181, 198, 198 (fig.), 199 (fig.), 734
DEC VAX, 181-185, 185 (fig.)
DEC VAX medidas detalladas, 728
dinámicas, 96, **149**, 150 (fig.)
comparaciones de, por arquitectura, 199 (fig.)
distribuciones de tiempo, 149, 184, 197-199, 734-741
IBM 370/168, 734-739
8086 en un IBM PC, 738-740
poniente de DLX, 740-741
VAX11/780, 734-736
distribuciones de frecuencia, 734, 735 (fig)
DLX, 192-197, 194 (fig.)
medidas detalladas, 731-732
estáticas, **149**
IBM 360, 186-189, 188 (fig.), 198 (fig.), 199 (fig.)
IBM 360 medidas detalladas, 730-731
Intel 8086, 176 (fig.), 177 (figs.), 178 (fig.), 186 (fig.)
medidas detalladas, 731
procesador (ISP) (*ver* repertorio de instrucciones, arquitectura)
rendimiento y, 38-40, 41-42, 71
usuario (*ver* Digital Equipment Corporation, VAX, repertorio de instrucciones de usuario)
utilización (*ver* repertorio de instrucciones, medidas)
representación, de dígitos con signo **690-691** (*ver también* aritmética, con signo)
de logaritmos con signo, 708 (*ver también* aritmética, con signo)
requerimientos, funcional, 13-16, 15 (fig.)
requisitos funcionales (*ver* requisitos, funcional)
resto (*ver* aritmética, resto)
resumen de ejemplos de jerarquía de memoria, 523 (fig.)
retardo, de carga, **287**, 300, 312
de cerrojo, 271
de rotación, **556** (*ver también* disco, magnético)
retención, **672**
retorno (*ver* procedimiento de llamada/retorno)
reubicación, **467** (*ver también* memoria virtual, reubicación y)
riesgo, **276**-278, 300 (*ver también* dependencias; procesador vectorial, dependencias de datos)
control, **276** (*ver también* riesgo, salto)
datos, **276**, 279-289, 303, 304-306, 307-312, 312-321, 322-329, 371 (*ver también* procesador vectorial, dependencias de datos; segmentación encauzada)
manipulación en VAX 8800, 355-356
dependencias de datos verdaderas (*ver* procesador vectorial, dependencias de datos)
detección, 287-289, 359 (*ver también* penalización de saltos)
algoritmo de Tomasulo y, 323, 324-329
dinámica de riesgos de memoria, 312-321, 322-329, 364
DLX y detección de riesgos de datos, 287-289
DLX y detección de riesgos estructurales, 314
instrucciones de punto flotante y enteras solapadas y, 306
marcador y, 315-321
punto flotante y, 307-308
VAX 8600 y, 352-354
estructural, **276**, 276-278, 306, 307, 316, 324
CPI y, 279
DLX y, 311, 312-315, 322
máquina superescalar, 342
memoria, detección dinamica de, 364
RAW, **284**, 307, 316, 319, 324, 355 (*ver* riesgo, RAW)
vectores y (*ver* procesador vectorial, dependencias de datos)
salto, **290**-293, 328
manipulación en VAX 8800, 355-356 (*ver también* salto, penalización)
tratamiento vectorial y, 403 (*ver también* procesador vectorial, dependencias de datos)
WAR, **284**, 308, 315-318, 327
WAW, **284**, 308-309, 315-318, 327
riesgos, reducción (*ver* riesgo, detección; salto,-penalización, reducción)
de control (*ver* riesgos, control)
de datos (*ver* riesgos, datos)
en la segmentación encauzadao (*ver* riesgo, datos)
Riordan, T., 204
RISC (*ver* computadores de repertorio reducido de instrucciones)
de Berkeley (*ver* computador de repertorio de instrucciones reducido, Berkeley)
-I y RISC-II, 203, 204
Riseman, E. M., 141

ROM (*ver* memoria de sólo lectura)
Rowen, C., 204
RPS (*ver* sensor de posición rotacional)
RR (*ver* registro-registro)
RS (*ver* registro-almacenamiento)
RS232, 567 (*ver también* redes)
RX (*ver* registro-indexado)

S

S810/20 (*ver* Hitachi S810/20)
Saji, K., 86
salto, **111**-113, 128 (*ver también* bifurcación)
  salto, 110, **111**-117, 143 (*ver también* bifurcación, esquemas de predicción de saltos)
  bucle, 115
  ciclos de reloj y, 239-241, 254
  códigos de condición (CC), 40, **114**, 215, 303-305, 359
  comportamiento, 292-294
  condicional, 25
    condicional 111-116, 217, 233 (*ver también* instrucción de salto)
    de arquitecturas RISC, 751-752
  desplazamiento, 113-114
    optimización, **122** (*ver también* optimización)
  destino, 112-114
  diferido (*ver* salto, retardado)
  DLX y (*ver* instrucción de salto, de DLX)
  efectivo, 114-**116, 290**, 293 (*ver también* esquemas de predicción de salto, predicción de salto efectivo
  en DLX, 237-241
  esquemas, 297 (*ver también* esquemas de predicción de salto; predicción dinámica de saltos por hardware)
  frecuencia, 293
  incondicional, (*ver* bifurcación)
  instrucción, 39-41, 111
    condiciones condicional, 39-41, 111
    condiciones de salto en DLX, 217, 253
    de DLX, 217, 239 (fig.), 247 (fig.), 247-254
  mal predicho (*ver* penalización de fallo de predicción)
  no efectivo, **290**, 292 (*ver también* esquemas de predicción de salto, predicción de efectividad)
  optimización y, 122-124, 127-130
  penalización, 291-293, 297-298
    determinación, 334
    reducción, 293-299, 329-337
      en DLX, 296-298, 333
  planificación, 293-296, 304
  predicción (*ver* esquemas de predicción de salto)
  registro de condición, **114**
  regla 90/50 de salto efectivo,
  retardado, 295 (fig.), **294**, 296-300, 364
  retardo, **291**, 292-298, 360 (*ver también* riesgos, huecos de retardo de salto; esquemas de predicción de saltos)
  riesgo (*ver* riesgo, salto)
  saltos relativos al PC (contador de programa), **113**
  segmentación y (*ver* esquemas de predicción de salto)
saltos incondicionales (*ver* salto)
Samples, D., 205
SAXPY (*ver* procesador vectorial, benchmark Linpack)
Schwartz, J. T., 141, 634
SCRAM (*ver* columna estática de DRAM)
SCSI (*ver* interfaz de sistemas de pequeños computadores)
sección crítica (*ver* sincronización)
seccionamiento, 391-393
sectores, **554** (*ver también* disco, magnético)
segmentación, 348, 351 (fig.)
  segmentación, **269**-374
  acelerar, 269-272
  bucle software-segmentado
  ciclos de reloj y, 377
  dificultades de implementación, 298-306
    complicaciones del repertorio de instrucciones, 303-306, 358-360
    tratar con interrupciones, 299-303
  DLX, 270-276, 290, 298-304, 322, 323 (fig.)
    y entero, 270-299
    y punto flotante, 279, 305-312, 321-322
    y rendimiento de, 299, 311
    y superescalar DLX (*ver* superescalar)
  segmentada, **361**, 368
    de operaciones enteras (*ver* segmentación, DLX y, entero)
    software (*ver* segmentación, software)
  equilibrio, en distribución, 343
    entre etapas, 270
  escrituras y (*ver* resultado de escritura en un cauce)
  hacer que funcione la segmentación, 273-276
  INTEL 860 y, 765
  para sedimentación de punto flotante *ver* segmentación, DLX y, punto flotante)
  paralelismo a nivel de instrucción, 337-352, 365-366
    desenrollamiento de bucles y, 338-342
    planificación dinámica y, 344-346
    máquinas superescalares y, 341-344
      método VLIW y, 345-349
    segmentación software y, 348-352

segmentación (*cont.*)
planificación, de trazas y, 348-352
dinámica, **313**, 312-331, 365
algoritmo de Tomasulo (*ver* algoritmo de Tomasulo)
distribución de múltiples instrucciones y, 344-346
método del marcador (*ver* marcador)
predicción dinámica por hardware, 329-338 (*ver también* esquemas de predicción de saltos)
distribución de múltiples instrucciones y, 344-346
profundidad de la segmentación, 271, 277, 360, 364
rendimiento de, 298, 311
riesgos de (*ver* riesgo)
software para, **348**-352, 365
superescalar DLX (*ver* superescalar)
temporización de instrucciones, 227, 280 (*ver también* aceleración de la segmentación)
VAX 8600, 351-359
y decodificación y búsqueda de operandos, 354-355
y gestión de interrupciones, 356-359
y manipulación de dependencias de datos, 355
y manipulación de dependencias de control, 355, 556
segmentación, 8, 24, 27, **269**
segmento, **467, 468** (*ver también* memoria virtual, segmento)
segmentos, de proceso, **475** (*ver también* memoria virtual, VAX-11/780 y)
del sistema, 476 (*ver también* memoria virtual, VAX-11/780 y)
paginados, **468** (*ver también* memoria virtual, página; memoria virtual segmento)
selección, a nivel de dispositivo (DLS), **596** (*ver también* disco, magnético, subsistema de memoria IBM 3990 y)
de trazas, **350**
semáforo (*ver* sincronización)
semisumadores, **642** (*ver también* aritmética)
sensor de posición de rotacion (RPS), 594-595 (*ver también* disco, magnético, subsistema de almacenamiento IBM 3990 y)
sentencias ejecutadas condicionalmente (*ver* procesador vectorial, sentencias ejecutadas condicionalmente y)
señal
propagación, 19
retardo, 19
separación, **394** (*ver también* procesador vertical)
vectorial (*ver* procesador vectorial)
separaciones no unitarias, **394** (*ver también* procesador vectorial, zancada)
Sequent Corporation, 627
Balance 2100, 627
Balance 8000, 627
multiprocesador Symmetry, 627-630, 634 (*ver también* multiprocesador)
servidor de ficheros
frente a estación de trabajo, 538
Shurkin, J., 27
Shustek, L. J., 148, 184-186, 198, 200
SI (*ver* almacenamiento inmediato)
signo y magnitud, **647** (*ver también* aritmética, con signo)
Simposio Anual Internacional sobre arquitectura de computadores (*ver* arquitectura, Simposio Anual Internacional sobre)
simular la ejecución (*ver* ejecución, simulación)
sincronización, 508 (*ver también* cache, coherencia)
sinónimos, **495** (*ver también* alias)
«síndrome de von Neumann», 632
sistema operativo, 136-139
operativo, 13-15 (fig.), 21 (fig.)
sistemas de ficheros, 551
Slater, R., 27
Slotnick, D. L., 634
Smalltalk, soporte para en SPARC, 758-760
Smith, A., 525, 528
Smith, J. E., 83
Smith, T. M., 26
software, 17-19 (*ver también* equilibrio, software y hardware)
solapamiento
de reloj, 271-273, 361
E/S (*ver* entrada/salida, rendimiento del sistema y)
procesamiento vectorial y, 387, 415-419
recubrimientos, **467** (*ver también* memoria virtual)
tripletas, **686**, 702 (*ver también* aritmética, entero, acelerar multiplicación)
solapar (*ver* segmentación)
sombreado, **561**
SPARC, 179, 203
arquitectura, 203
instrucciones, 747-749
extensiones comunes a instrucciones DLX, 753-755
única, 759-761
resumen de, 744
SPARCstation 1 (*ver* SPARC)
SPEC (Cooperativa de Evaluación del Rendimiento de Sistemas) (*ver* programas de benchmark)
SRAM (*ver* memoria estática de acceso aleatorio)
SS (*ver* almacenamiento-almacenamiento)
SSD (*ver* discos de estado sólido)
estado de unidad funcional, 318, 317 (figs.), 320 (fig.) 326 (fig.), 328 (fig.)
Stern, N., 26
Strapper, C. H., 86

Strecker (*ver* Bell, C. G. y W. D. Strecker)
Strecker, W. W., 140
Stretch (*ver* International Business Machines Corp., IBM 7030)
strobe de acceso, a columna (CAS), **459**
a fila (RAS), **459**
subbloques, **492** (*ver también* cache, subbloques)
subexpresión (*ver* eliminación de subexpresiones comunes)
subsistema de almacenamiento, IBM (*ver* disco, magnético; subsistema de almacenamiento IBM 3990 y)
suma (*ver* aritmética, suma, punto flotante; aritmética, entero, suma; aritmética, entero, aceleración de suma)
con precisión múltiple, 652
sumador, con ahorro de acarreo (CSA), **685**-687, 687 (fig.), 694
con anticipación de acarreo (CLA), **674** (*ver también* aritmética, entero, acelerar suma)
con transmisión (propagación) de acarreo, **642** (*ver también* aritmética, entero, sumador con propagación de acarreo)
(CPA), 686, 490-492, 693, 697
con salto de acarreo, **678** (*ver también* aritmética, entera, acelerar suma)
con selección de acarreo, **680** (*ver también* aritmética, entera, acelerar suma entera)
de suma condicional, **709**
sumadores, 681 (fig.), (*ver también* aritmética, entero, propagación de acarreos; aritmética, entera, acelerar suma)
completos **642** (*ver también* aritmética)
Sumner, F. H., 29
Sun Microsystems (*ver también* SPARC)
compilador 1.2 FORTRAN, 88
compilador C, 88
compilador FORTRAN 77, 87
supercomputador, 3-4, 538
CRAY X-MP (*ver* máquinas Cray Research, CRAY X-MP)
CRAY Y-MP (*ver* máquinas Cray Research, CRAY Y-MP)
CRAY-1 (*ver* máquinas Cray Research, CRAY-1)
CRAY-2 (*ver* máquinas Cray Research, CRAY-2)
E/S y (*ver* entrada/salida, supercomputadores y; disco, magnético, benchmarks de E/S para)
Fujitsu (*ver* supercomputador, japonés)
japonés, 379, 419, 423
NEC SX-2 (*ver* supercomputador, japonés)
supercomputadores japoneses (*ver* supercomputadores, japoneses)
supersegmentados, **363**, 365-366
superescalar
desenrollamiento de bucles y, 342-344
distribución de instrucciones, 341-344
DLX, 341-344, 349
(*ver* superescalar, DLX)
en Intel 860, 766-767
máquinas, 341-344, 365-366, 626-627
paralelismo a nivel de instrucción, 341-344
procesador, 361-363
riesgos estructurales y, 342
segmentación, 342 (fig.)
«super-microprocesador», 538
sustitución (reemplazo), bloque (*ver* reubicación de bloque)
de bloques, **438**, 523
aleatoria, **442**
frente a menos recientemente utilizado, 443 (fig.)
búsqueda fuera de orden, 494
cache VAX-11/780 y, 443
caches y, 442-444, 454
memoria virtual, 470
menos recientemente utilizado (LRU), **442**-443, 470
en la VAX11/780, 477
frente a aleatorio, 443 (fig.)
primero en entrar primero en salir (FIFO), **443**
rearranque anticipado, 494
Sutherland, I., 561, 605, 607
Synapse N+1, 508, 526

T

tabla, de colores, **564** (*ver también* pantallas gráficas)
«look-up» de vídeo, **564** (*ver también* pantallas gráficas)
de descriptores, **481** (*ver también* memoria virtual, tabla de páginas; memoria virtual Intel 80286/80385 y)
de páginas, **469** (*ver también* memoria virtual, tabla de páginas)
de páginas invertida, **469** (*ver también* memoria virtual)
tablas de reserva de la segmentación, 274, 364
tamaño, de bloque (*ver* cache, bloques y, tamaño; memoria virtual, paginada, tamaño de página)
de página (*ver* memoria virtual, paginada, tamaño de página)
Taylor, G., 203
tecnología (*ver* diseño, computador; disco; implementación; lógica)
temporización de instrucciones (*ver* segmentación temporización de instrucciones)
terminación fuera de orden, **309**-311, **314**-316

test, de Banerjee (*ver* procesador vectorial, dependencias de datos, test de Banerjee)
examina e inicializa, **510** (*ver* cache, coherencia, sincronización)
TeX, 48, 71, 73, 75, 84, 85, 91
Texas Instruments
8847, 668, 696 (fig.), 700
Thacker, C., 526, 528, 603, 607
Thadhani, A., 604, 607
TI (*ver* Texas Instruments)
tiempo, compartido, 619
de acceso, 21, **435**, 454, **458** (*ver también* jerarquía de memoria, tiempo de acceso, tiempo de acceso a cache)
de acierto, **435** (*ver también* jerarquía de memoria)
de arranque, **385** (*ver también* procesador vectorial, tiempo de arranque)
de asimilación, **548** (*ver también* entrada/salida, transacciones y)
de búsqueda, **555** (*ver también* disco, magnético, búsquedas y)
de controlador, **556** (*ver también* disco, magnético)
de CPU, de usuario (*ver* unidad central de proceso, tiempo de CPU, usuario)
del sistema (*ver* unidad central de proceso, tiempo de CPU, sistema)
de ejecución, 5-8, 29, **37**, 30, 31 (*ver también* tiempo de respuesta; rendimiento; media)
aceleración y, 10
localidad de referencia y, 11-13 (*ver también* localidad)
media por instrucción, 82
normalizado, 54-57, 88
ponderado, 55, 89
rendimiento y, 6, 37, 42-49, 51-53, 75-77, 86
total, 53, 88
de entrada, **548** (*ver también* entrada/salida, transacciones y)
retardo de cola, **556** (*ver también* disco, magnético)
de respuesta, 6, 23, 546 (*ver también* tiempo de ejecución; rendimiento; entrada/salida, rendimiento, tiempo de respuesta)
de E/S (*ver* entrada/salida, rendimiento, tiempo de respuesta)
definición de, 5
del sistema, **548** (*ver también* entrada/salida, rendimiento, tiempo de respuesta)
de transacción, **546** (*ver también* entrada/salida, transacción y)
de transferencia, **435, 556** (*ver también* jerarquía de memoria, fallo; disco, magnético)
medio, de acceso a memoria, **497** (*ver también* jerarquía de memoria, tiempo de acceso; cache, tiempo de acceso; cache, caches de dos niveles)
de ejecución de una instrucción, 243
de reparación (MTTR), **561** (*ver también* entrada/salida, fiabilidad)
entre fallos (MTTF), **561** (*ver también* entrada/salida, fiabilidad)
transcurrido (elapsed), 37-39, 73, 74, 77
escalado de CPU y E/S solapadas, 542-543 *(ver también* entrada/salida, rendimiento del sistema y)
mejor de CPU y E/S solapadas, 542-544 (*ver también* entrada/salida, rendimiento del sistema y)
peor de CPU y E/S solapadas, 542-544 (*ver también* entrada/salida, rendimiento del sistema y)
TLB (*ver* memoria virtual, buffer de traducción anticipada)
totalmente asociativa, **439** (*ver también* cache, completamente asociativa)
TP (*ver* tratamiento de transacciones)
TP-1, 550, 551 (fig.) (*ver también* benchmark; disco, magnético, benchmarks de E/S para)
TPI (*ver* ciclos de reloj por instrucción)
traducción dinámica de dirección (*ver* memoria virtual, traducción de direcciones)
transacción, **546** (*ver también* entrada/salida, transacciones y)
del bus, **570** (*ver también* bus)
transacciones divididas, **571** (*ver también* bus)
transferencia, **111** (*ver también* salto)
de bloques de bits, 561-562 (*ver también* pantallas gráficas)
de datos, 84, 145
en arquitecturas RISC, 747-748
instrucción atómica de intercambio, **753**-755
no alineado **756**-757
opción «Endian», **753** (*ver también* Gran Endian; Pequeño Endian)
traza, **350**
«tres C» (*ver* cache, fallo)
trivialidades, contraportada anterior
troncos de datos, **318**
tubo de rayos catódicos (CRT), **561** (*ver también* pantallas gráficas)
Tuck, R., 204
Tucker, S., 259

U

ubicación, bloque (*ver* ubicación de bloque)
de bloques, **438**, 523 (*ver también* fallos debidos a conflictos)
caches y, 438-440, 452 (*ver también* cache, completamente asociativa; cache asociativa

ubicación (*cont.*)
por conjuntos; cache, correspondencia directa)
memoria virtual, 470
subbloques, 491-493
VAX-11/780 cache y, 453
interprocedural de registros, **488** (*ver también* ventanas de registros)
de subbloques, **492** (*ver también* cache, subbloques)
en escritura, **444** (*ver también* cache, fallo de escritura)
Ultracomputador de la Universidad de New York (NYU), 634
una sola escritura, muchas lecturas (WORM), 535
Ungar, D., 203
Unibus (*ver* bus, Unibus)
unidad (*ver* disco, magnético)
aritmético-lógica (ALU), 41-45, 215
ciclos de reloj por instrucción y, 239-242, 250
codificación y, 250-253
efecto sobre las lluvias de los bosques de los artículos relacionados con, 215
estados DLX y, 237 (fig.), 237-242
instrucciones y operaciones de, 98, 100, 108, 110, 113, 128-130, 131-133, 142, 146, 216-218, 226, 228, 245-252 (figs.), 254 (fig.)
central de proceso (CPU), 8, 14, 96-99, **213** (*ver también* camino de datos; procesador)
buses de memoria-CPU, **570** (*ver también* bus)
ciclos de reloj de ejecución de CPU y caches, 447
diferencia de rendimiento CPU-DRAM, 459, 460 (fig.), 465-466
equilibrio y, 19
interfaz a E/S (*ver* entrada/salida, interfaz con la CPU)
jerarquía de memoria y, 19-21 (*ver también* jerarquía de memoria)
rendimiento, **37**, 38-43, 76 (*ver también* rendimiento)
del sistema y, 11-12, 17
tiempo, **37**-43, 44, 73-74, 131
caches y, 448, 450
CPU, del usuario, **37**
del sistema, **37**
de inactividad, 538-539
E/S y, 537
de disco (*ver* disco, magnético)
dura (*ver* disco, magnético)
unidades funcionales, 269-277, 312-321, 322-328, 341-343, 346-348, 364
múltiples, 305-307, 364, 371
(*ver* unidades funcionales, múltiple)
procesamiento vectorial y (*ver* procesador vectorial, unidades funcionales)
vectoriales (*ver* procesador vectorial, unidades funcionales)
unidades VAX de rendimiento (VUP), 84
unificadas, **456** (*ver también* cache)
uniprocesador, 76-78
UNIVAC I, 27, 28, 258
Universidad de Manchester, 25, 522
Universidad de Harvard, 25-27
UNIX, 4, 15 (*ver también* sistema operativo)
utilización (*ver* repertorio de instrucciones, medidas)

V

variable escalar, **124**
global, **124**, 128
variables, con alias, **125**
de bloqueo, **508** (*ver también* cache, coherencia, sincronización)
enteros, **117**, 126
VAX (*ver* Digital Equipment Corporation, VAX)
VAXstation (*ver* Digital Equipment Corporation, VAX)
vector, **378** (*ver también* procesador vectorial)
de índices, **409** (*ver también* procesador vectorial, matrices dispersas)
frecuencia, 30
modo, 30
operaciones, en Intel 860, 764
procesador, 27
vectorización, porcentaje de, 30
velocidad de datos, **552**
velocidad de iniciación, **385** (*ver* procesador vectorial, frecuencia de iniciación)
velocidad de terminación, **385** (*ver también* procesador vectorial)
ventanas de registros **486**-490, 523 (fig.), 526, 759
beneficios de, en DLX, 489 (fig.)
carga y almacenamiento, 489 (fig.)
número de frente a frecuencia de desbordamiento, 487 (fig.)
pros y contras de, 487-490
resumen de, 523 (fig.)
VIBA (*ver* dirección virtual del buffer de instrucciones)
VLIW (*ver* palabra de instrucción muy larga)
VLR (*ver* registro de longitud vectorial, procesadores vectoriales, longitud vectorial)
VMS
compilador C, 72
fort (compilador FORTRAN), 72
von Neumann, J., 24-26

W

Wakerly, J., 202
Wallace, J. J., 83
WAR (*ver* escritura después de lectura)
Ward, S., 604
Waters, F., 203
WAW (*ver* escritura después de escritura)
WCS (*ver* almacenamiento de control escribible)
Weitek 3364, 696 (fig.), 697, 700
Wheeler, D. J., 25
Whetstone (*ver* programas de benchmark, sintético)
Whirlwind, 26-27
Wichmann, B. A., 82
Wiecek, 182, 184, 201
Wilkes, M., 25, 27, 524, 528
WORM (*ver* escribir una vez, muchas lecturas)
Wortman, D. B., 139, 204
Wulf, W., 140, 524, 528

X

X-MP (*ver* máquinas de Cray Research)

Y

Y-MP (*ver* máquinas de Cray Research)

Z

Zimmermann, R., 202
Zorn, B., 204
Zuse, 26