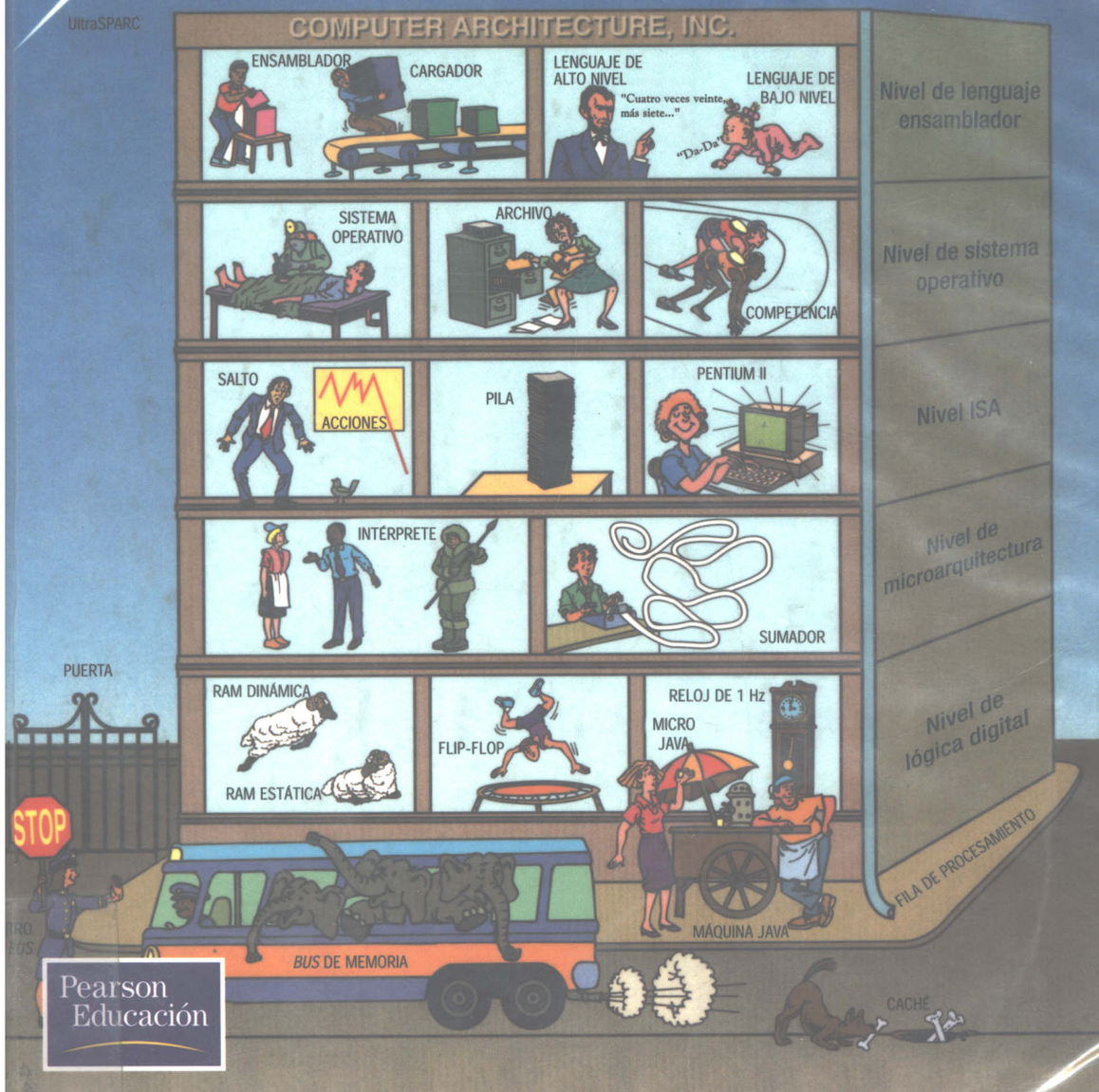
Cuarta edición

ORGANIZACIÓN DE COMPUTADORAS

UN ENFOQUE ESTRUCTURADO

Andrew S. Tanenbaum



Pearson Educación

ORGANIZACIÓN DE COMPUTADORAS
un enfoque estructurado

LOSA LIBROS
Colonia 1551/53
Tel/Fax: 401 - 2905 - 401 - 8587
http:// www.losa.com.uy
libros@losa.com.uy
Montevideo - Uruguay

Otros títulos exitosos de Andrew S. Tanenbaum

Redes de computadoras, tercera edición

Este clásico tan leído proporciona la introducción ideal a las redes de hoy y del mañana. Explica en detalle cómo se estructuran las redes modernas. Partiendo de un nivel físico y avanzando hasta el nivel de aplicación, el libro abarca un vasto número de temas importantes, incluyendo comunicación inalámbrica, fibra óptica, protocolos de cadenas de datos, Ethernet, algoritmos de enrutamiento, comportamiento de redes, seguridad, DNS, correo electrónico, novedades de USENET, la World Wide Web y multimedia. El libro presenta un estudio completo de TCP/IP e Internet.

Sistemas operativos: diseño e implementación, segunda edición

Este famoso texto sobre sistemas operativos es el único que abarca tanto los principios de los sistemas operativos como sus aplicaciones en un sistema real. En él se analizan en detalle todos los temas tradicionales de sistemas operativos. Se ilustran los principios con MINIX, un sistema operativo libre del tipo UNIX basado en POSIX, para computadoras personales. Cada libro incluye un disco compacto gratis con el sistema MINIX completo, incluyendo todo el código fuente. Éste se lista en un apéndice del libro y se explica en detalle dentro del texto.

Sistemas operativos modernos

Este exitoso texto presenta las bases de un procesador simple y de un sistema de computadora distribuido. Trata todos los temas tradicionales, incluyendo procesos, administración de memoria y sistemas de archivos, así como temas clave sobre sistemas distribuidos, entre ellos, el modelo cliente-servidor, las llamadas de procedimiento remoto, las redes y los servidores de archivo distribuidos. Los principios empleados en el libro se ilustran con dos procesadores simples extensivos y dos sistemas distribuidos.

Sistemas operativos distribuidos

Este texto incluye los conceptos fundamentales de los sistemas operativos distribuidos. Entre los temas clave están la comunicación y la sincronización, los procesos y los procesadores, la memoria compartida distribuida, los sistemas de archivos distribuidos y los sistemas distribuidos de tiempo real. Los principios se ilustran con cuatro ejemplos de capítulo extensos.

ORGANIZACIÓN DE COMPUTADORAS
un enfoque estructurado

CUARTA EDICIÓN

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, Países Bajos

Con la colaboración de
JAMES R. GOODMAN
University of Wisconsin
Madison, WI

Prentice Hall
Pearson Educación
Addison Wesley

MÉXICO • ARGENTINA • BRASIL • COLOMBIA • COSTA RICA • CHILE
ESPAÑA • GUATEMALA • PERÚ • PUERTO RICO • VENEZUELA

Datos de catalogación bibliográfica

Tanenbaum, S. Andrew
Organización de computadoras:
un enfoque estructurado
PRENTICE HALL
México, 2000

ISBN: 970-17-0399-5
Área: Universitarios

Formato: 17 × 23 cm Páginas: 688

Versión en español de la obra titulada *Structured computer organization, Fourth Edition*, de Andrew S. Tanenbaum, publicada originalmente en inglés por Prentice Hall Inc., Upper Saddle River, New Jersey, U.S.A.

Esta edición en español es la única autorizada.

Original English language title by
Prentice Hall Inc.

ISBN 0-13-095990-1

Edición en español:
Editor: Pablo Eduardo Roig Vázquez
Supervisor de traducción: Rocío Cabañas Chávez
Supervisor de producción: Alejandro A. Gómez Ruiz

Edición en inglés:
Publisher: Alan Apt
Development editor: Sondra Chavez
Editor-in-chief: Marcia Horton
Production editor: Irwin Zucker
Managing editor: Eileen Clark
Composition and interior design: Andrew S. Tanenbaum
Cover concept: Andrew S. Tanenbaum
Cover illustrator: Don Martinetti, DM Graphics, Inc.

SÉPTIMA EDICIÓN, 2000

MAR

LITOGRAFICA INGRAMEX, S.A. DE C.V.
CENTENO NO. 162-1
MEXICO, D.F.
C.P. 09810

2000

ISBN 970-17-0399-5

Impreso en México. *Printed in Mexico*

1 2 3 4 5 6 7 8 9 0 03 02 01 00

A Suzanne, Barbara, Marvin, Bram, y a la memoria de Sweetie π

CONTENIDO

PREFACIO

Las primeras tres ediciones de este libro se basaron en la idea de que una computadora puede considerarse como una jerarquía de niveles, cada uno de los cuales desempeña alguna función bien definida. Este concepto fundamental es tan válido hoy como cuando apareció la primera edición, así que se ha conservado como base para la cuarta edición. Al igual que en las tres primeras ediciones, se analizan detalladamente el nivel de lógica digital, el nivel de microarquitectura, el nivel de arquitectura del conjunto de instrucciones, el nivel de máquina del sistema operativo y el nivel de lenguaje ensamblador (aunque hemos modificado algunos de los nombres de modo que reflejen la práctica moderna).

Aunque se ha mantenido la estructura básica, esta cuarta edición contiene muchos cambios, pequeños y grandes, que lo ponen al día en la industria de las computadoras con su vertiginoso ritmo de cambio. Por ejemplo, todos los ejemplos de código, que estaban en Pascal, se han reescrito en Java con objeto de reflejar la popularidad de este lenguaje en el mundo de la computación. Además, se han actualizado los ejemplos de máquinas empleados. Los ejemplos actuales son el Intel Pentium II, el Sun UltraSPARC II y el Sun picoJava II, un chip Java incorporado de bajo costo.

El uso de los multiprocesadores y las computadoras paralelas se ha extendido mucho desde la tercera edición, por lo que se reescribió totalmente el material sobre arquitecturas paralelas, expandiéndolo considerablemente; ahora cubre una amplia gama de temas, desde multiprocesadores hasta los COW.

El libro se ha vuelto más extenso con el paso de los años (aunque todavía no es tan largo como otros libros populares sobre el tema). Tal expansión es inevitable a medida que avanzan los conocimientos en el área. Por ello, si el libro se usa para un curso, tal vez no siempre sea posible terminar el libro en un solo curso (por ejemplo, en un sistema de trimestres). Una

posible estrategia sería cubrir los capítulos 1, 2 y 3 en su totalidad, la primera parte del capítulo 4 (hasta la sección 4.4 inclusive) y el capítulo 5 como mínimo indispensable. Si sobra tiempo, podría dedicarse al resto del capítulo 4 y partes de los capítulos 6, 7 y 8, dependiendo del interés del profesor.

A continuación describimos los principales cambios hechos a cada capítulo desde la tercera edición. El capítulo 1 todavía contiene una reseña histórica de la arquitectura de las computadoras; nos dice cómo llegamos a dónde estamos ahora y qué hitos pasamos en el camino. Se describe la gama más amplia de computadoras que existen ahora y se presentan nuestros tres ejemplos principales (Pentium II, UltraSPARC II y picoJava II).

En el capítulo 2 se actualizó el material sobre dispositivos de entrada/salida, con hincapié en la tecnología de los dispositivos modernos, que incluyen los discos RAID, los CD grabables, los DVD y las impresoras a color, entre muchos otros.

El capítulo 3 (nivel de lógica digital) sufrió algunas modificaciones y ahora trata los buses de computadora y los modernos chips de E/S. El principal cambio aquí es material adicional sobre buses, sobre todo el bus PCI y el bus USB. Los tres nuevos ejemplos se describen aquí en el nivel de chip.

El capítulo 4 (que ahora se llama nivel de microarquitectura) se reescribió totalmente. Se conservó la idea de usar un ejemplo detallado de máquina microprogramada para ilustrar las ideas de control del camino de datos, pero la máquina de ejemplo ahora es un subconjunto de la Máquina Virtual Java. La microarquitectura subyacente se modificó de manera acorde. Se dan varias iteraciones del diseño, mostrando los posibles equilibrios en términos de costo y desempeño. El último ejemplo, el Mic-4, usa un conducto de siete etapas y ofrece una introducción accesible al funcionamiento de las computadoras modernas importantes como el Pentium II. Se añadió una sección nueva sobre cómo mejorar el desempeño, que se concentra en las técnicas más recientes como uso de cachés, predicción de ramas, ejecución en desorden (superescalar) y predicación. Las nuevas máquinas de ejemplo se describen en el nivel de microarquitectura.

El capítulo 5 (que ahora se llama nivel de arquitectura del conjunto de instrucciones) se ocupa de lo que mucha gente llama el "lenguaje de máquina". Aquí se usan como ejemplos primarios el Pentium II, el UltraSPARC II y la Máquina Virtual Java.

El capítulo 6 (nivel de máquina del sistema operativo) tiene ejemplos para el Pentium II (Windows NT) y UltraSPARC II (UNIX). El primero es nuevo y tiene muchas características que vale la pena examinar, pero UNIX sigue siendo un caballito de batalla confiable en muchas universidades y compañías, y además resulta provechoso examinarlo con detalle por su sencillo y elegante diseño.

El capítulo 7 (nivel de lenguaje ensamblador) se actualizó usando ejemplos de las máquinas que hemos estado estudiando. También se añadió material nuevo sobre enlazado dinámico.

El capítulo 8 (arquitecturas de computadoras paralelas) se reescribió totalmente. Ahora cubre con detalle tanto multiprocesadores (UMA, NUMA y COM) como multicomputadoras (MPP y COW).

La bibliografía se sometió a una revisión y actualización exhaustivas. Más de las dos terceras partes de las referencias corresponden a obras que se publicaron después de que lo

hiciera la tercera edición de este libro. Los números binarios y los de punto flotante no han sufrido muchos cambios a últimas fechas, por lo que los apéndices son prácticamente los mismos de la edición anterior.

Por último, algunos problemas se modificaron y se añadieron muchos problemas nuevos respecto a la tercera edición.

Existe un sitio Web para este libro. Es posible obtener electrónicamente archivos PostScript para todas las ilustraciones del libro. Estos archivos pueden imprimirse, por ejemplo, para crear transparencias. Además, el sitio incluye un simulador y otras herramientas de software. El URL para este sitio es

http://www.cs.vu.nl/~ast/sco4/

El simulador y las herramientas de software son producto de Ray Ontko. El autor desea expresar su gratitud a Ray por haber producido tan útiles programas.

Varias personas leyeron (parcialmente) el manuscrito y ofrecieron sugerencias útiles o ayudaron de otras maneras. En particular, quiero agradecer a Henri Bal, Alan Charlesworth, Kourosh Gharachorloo, Marcus Goncalves, Karen Panetta Lentz, Timothy Mattson, Harlan McGhan, Miles Murdocca, Kevin Normoyle, Mike O'Connor, Mitsunori Ogihara, Ray Ontko, Aske Plaat, William Potvin II, Nagarajan Prabhakaran, James H. Pugsley, Ronald N. Schroeder, Ryan Shoemaker, Charles Silio, Jr., y Dale Skrien su ayuda, que ha sido inapreciable. Mis estudiantes, sobre todo Adriaan Bon, Laura de Vries, Dolf Loth y Guido van't Noordende, también ayudaron a depurar el texto. Muchas gracias.

Me gustaría expresar un agradecimiento especial a Jim Goodman por sus muchas contribuciones a este libro, sobre todo a los capítulos 4 y 5. La idea de usar la Máquina Virtual Java fue suya, lo mismo que las microarquitecturas para implementarla. Muchas de las ideas avanzadas se deben a él. Este libro es mucho mejor de lo que habría sido sin su esfuerzo.

Por último, quiero agradecer a Suzanne su paciencia ante mis largas horas absorto en mi Pentium. Desde mi punto de vista, el Pentium es mucho mejor que mi antiguo 386, pero a ella no le parece que haya mucha diferencia. También quiero expresar mi gratitud a Barbara y Marvin por ser unos chicos excelentes, y a Bram por permanecer callado cuando yo trataba de escribir.

Andrew S. Tanenbaum

1

INTRODUCCIÓN

Una computadora digital es una máquina que puede resolver problemas ejecutando las instrucciones que recibe de las personas. Una secuencia de instrucciones que describe cómo realizar cierta tarea se llama **programa**. Los circuitos electrónicos de una computadora pueden reconocer y ejecutar directamente un conjunto limitado de instrucciones sencillas, y todos los programas tienen que convertirse en una serie de esas instrucciones para que la computadora pueda ejecutarlos. Dichas instrucciones básicas casi nunca son más complicadas que

Sumar dos números.

Verificar si un número es cero.

Copiar un dato de una parte de la memoria de la computadora a otra.

Juntas, las instrucciones primitivas de una computadora constituyen un lenguaje que permite a las personas comunicarse con la computadora. Dicho lenguaje se llama **lenguaje de máquina**. Las personas que diseñan una computadora nueva deben decidir qué instrucciones incluirán en su lenguaje de máquina. Por lo regular, se trata de hacer las instrucciones primitivas lo más simples posible, en congruencia con el uso que se piensa dar a la computadora y sus requisitos de desempeño, a fin de reducir la complejidad y el costo de los circuitos requeridos. Casi todos los lenguajes de máquina son tan simples que para las personas resulta difícil y tedioso usarlos.

Con el paso de los años, esta sencilla observación ha dado pie a que las computadoras se estructuren como una serie de abstracciones, donde cada una de éstas se apoya en la que está

abajo de ella. De este modo es posible controlar la complejidad y diseñar sistemas de cómputo de manera sistemática y organizada. Llamamos a este enfoque **organización estructurada de computadoras** y éste es también el nombre que hemos dado al presente libro. En la sección que sigue describiremos el significado de este término. Después veremos un poco de historia, hablaremos de lo último en organización de computadoras, y daremos algunos ejemplos importantes.

1.1 ORGANIZACIÓN ESTRUCTURADA DE COMPUTADORAS

Como ya se mencionó, hay una gran diferencia entre lo que es cómodo para las personas y lo que es cómodo para las computadoras. Las personas quieren hacer *X*, pero las computadoras sólo pueden hacer *Y*. Esto crea un problema. El objetivo de este libro es explicar cómo puede resolverse ese problema.

1.1.1 Lenguajes, niveles y máquinas virtuales

Se puede atacar el problema de dos maneras; ambas implican diseñar un nuevo conjunto de instrucciones que para las personas sea más fácil de usar que el conjunto de instrucciones de máquina original. Juntas, estas nuevas instrucciones también forman un lenguaje, que llamaremos L1, así como las instrucciones de máquina originales forman un lenguaje, que nombraremos L0. Las dos estrategias difieren en la forma en que la computadora ejecuta los programas escritos en L1 ya que, como dijimos, la computadora sólo puede ejecutar programas escritos en su lenguaje de máquina, L0.

Un método de ejecutar un programa escrito en L1 es sustituir primero cada instrucción escrita en L1 por una sucesión equivalente de instrucciones en L0. El programa resultante consiste exclusivamente en instrucciones de L0. Luego, la computadora ejecuta el nuevo programa en L0 en lugar del antiguo programa en L1. Esta técnica se llama **traducción**.

La otra técnica consiste en escribir un programa en L0 que tome programas en L1 como datos de entrada y los ejecute examinando sus instrucciones una por una y ejecutando directamente la sucesión de instrucciones en L0 que equivale a cada una. Con esta técnica, no es necesario generar primero un nuevo programa en L0. La técnica se conoce con el nombre de **interpretación** y el programa que la implementa se denomina **intérprete**.

La traducción y la interpretación son similares. Con ambos métodos las instrucciones en L1 se ejecutan finalmente realizando sucesiones equivalentes de instrucciones en L0. La diferencia es que, con la traducción, todo el programa en L1 se convierte primero en un programa en L0, el programa en L1 se desecha, el nuevo programa en L0 se carga en la memoria de la computadora, y se ejecuta. Durante la ejecución, el programa en L0 recién generado es el que se ejecuta y controla la computadora.

Con la interpretación, después de que cada instrucción en L1 se examina y se decodifica, se ejecuta inmediatamente. No se genera ningún programa traducido. Aquí es el intérprete el que controla la computadora. Para él, el programa en L1 no es más que datos. Ambos métodos se usan ampliamente, y cada vez se usa más una combinación de los dos.

En lugar de pensar en términos de traducción o interpretación, a menudo es más fácil imaginar la existencia de una computadora hipotética o **máquina virtual** cuyo lenguaje de máquina es L1. Llamemos a esta máquina virtual M1 (y sea M0 la máquina virtual que corresponde a L0). Si fuera posible construir tal máquina a un costo razonable, no habría necesidad de tener L1 ni una máquina que ejecutara programas en L1. La gente simplemente escribiría sus programas en L1 y la computadora los ejecutaría directamente. Incluso si es demasiado costoso o complicado construir con circuitos electrónicos la máquina virtual cuyo lenguaje es L1, es posible escribir programas para ella. Tales programas podrían interpretarse o traducirse con un programa escrito en L1 que la computadora existente puede ejecutar directamente. En otras palabras, las personas pueden escribir programas para las máquinas virtuales como si realmente existieran.

Para que la traducción o interpretación sea práctica, los lenguajes L0 y L1 no deben ser "demasiado" diferentes. Esta restricción a menudo implica que L1, aunque mejor que L0, todavía dista mucho de ser ideal para la generalidad de las aplicaciones. Este resultado podría ser decepcionante si pensamos en que el propósito original de crear L1 era evitar que el programador tuviera que expresar algoritmos en un lenguaje más apropiado para las máquinas que para las personas. No obstante, la situación no es desesperada.

La estrategia obvia es inventar un tercer conjunto de instrucciones que esté más orientado hacia las personas y menos orientado hacia la máquina que L1. Esas instrucciones también constituyen un lenguaje, al que llamaremos L2 (con una máquina virtual M2). Las personas pueden escribir programas en L2 como si en realidad existiera una máquina virtual M2 cuyo lenguaje de máquina es L2. Esos programas podrían traducirse a L1 o ser ejecutados por un intérprete escrito en L1.

La invención de una serie de lenguajes, cada uno más cómodo que sus predecesores, puede continuar indefinidamente hasta llegar a uno adecuado. Cada lenguaje se basa en su predecesor, por lo que podemos pensar en una computadora que emplea esta técnica como una serie de **capas** o **niveles**, uno encima del otro, como se muestra en la figura 1-1. El lenguaje o nivel más bajo es el más simple, y el lenguaje o nivel más alto es el más sofisticado.

Existe una relación importante entre un lenguaje y una máquina virtual. Cada máquina tiene cierto lenguaje de máquina, que consiste en todas las instrucciones que la máquina puede ejecutar. Efectivamente, una máquina define un lenguaje. De igual modo, un lenguaje define una máquina, la máquina que puede ejecutar todos los programas escritos en ese lenguaje. Desde luego, la máquina definida por un lenguaje dado podría ser enormemente complicada y demasiado costosa para construirse directamente con circuitos electrónicos, pero eso no quiere decir que no podamos imaginarla. Una máquina cuyo lenguaje de máquina es C++ o COBOL sería en verdad compleja, pero con la tecnología actual sería fácil construirla. No obstante, hay razones de peso para no construir semejante computadora: no sería económica en comparación con otras técnicas.

Una computadora con *n* niveles puede verse como *n* máquinas virtuales distintas, cada una con diferente lenguaje de máquina. Usaremos los términos "nivel" y "máquina virtual" indistintamente. Los circuitos electrónicos sólo pueden ejecutar directamente programas escritos en el lenguaje L0 sin necesidad de traducción ni interpretación. Los programas escritos

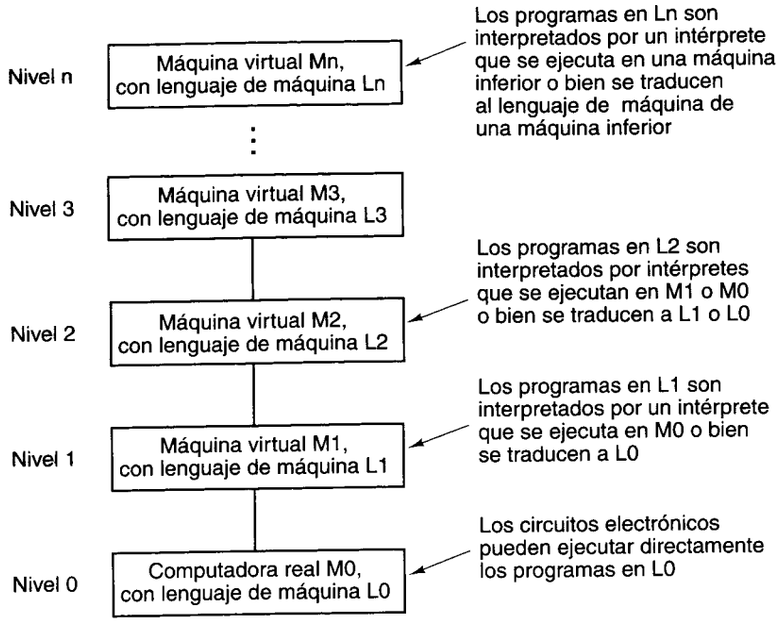


Figura 1-1. Una máquina multinivel.

en L1, L2, . . . , L*n* deberán ser interpretados por un intérprete que se ejecute en un nivel más bajo, o traducirse a otro lenguaje correspondiente a un nivel más bajo.

La persona encargada de escribir programas para la máquina virtual del nivel *n* no tiene que estar consciente de los intérpretes y traductores subyacentes. La estructura de la máquina asegura que de un modo u otro esos programas se ejecutarán. No interesa si un intérprete los ejecuta paso por paso, o si ese intérprete es a su vez ejecutado por otro intérprete o por los circuitos electrónicos. El resultado es el mismo en ambos casos: los programas se ejecutan.

En general, a los programadores que usan una máquina de nivel *n* sólo les interesa el nivel más alto, el que menos se parece al lenguaje de máquina que está hasta abajo. En cambio, las personas interesadas en entender cómo funciona realmente una computadora deben estudiarla en todos los niveles. Quienes se interesen en diseñar nuevas computadoras o nuevos niveles (o sea, nuevas máquinas virtuales) también deberán familiarizarse con niveles distintos del más alto. Los conceptos y técnicas para construir máquinas como una serie de niveles, y los detalles de los niveles mismos, constituyen el tema principal de este libro.

1.1.2 Máquinas multinivel contemporáneas

Casi todas las computadoras modernas constan de dos o más niveles, y pueden llegar a existir máquinas con hasta seis niveles, como se muestra en la figura 1-2. El nivel 0, en la base, es el

verdadero hardware de la máquina. Sus circuitos ejecutan los programas en lenguaje de máquina de nivel 1. Con ánimo totalizador, deberíamos mencionar la existencia de un nivel más bajo de nuestro nivel 0. Este nivel, que no se muestra en la figura 1-2 porque queda dentro del ámbito de la ingeniería eléctrica (y rebasa el alcance de este libro), se llama **nivel de dispositivos**. En este nivel, el diseñador ve transistores individuales, que son las primitivas de más bajo nivel para los diseñadores de computadoras. Si nos preguntamos cómo funcionan internamente los transistores, entramos en la física del estado sólido.

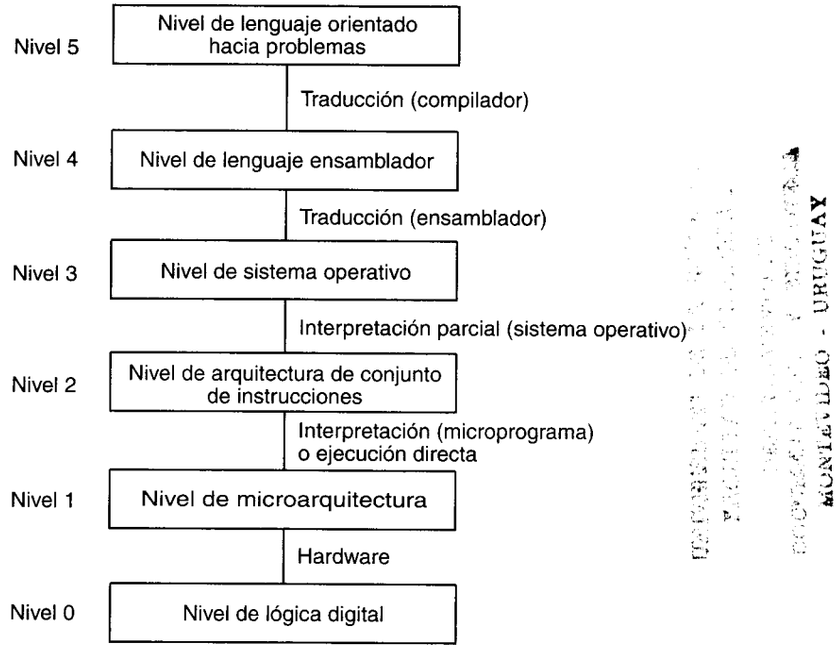


Figura 1-2. Computadora de seis niveles. El método de apoyo para cada nivel se indica inmediatamente abajo de él (junto con el nombre del programa de apoyo).

En el nivel más bajo que estudiaremos, el **nivel de lógica digital**, los objetos integrantes se llaman **compuertas**. Aunque se construyen con componentes analógicos, como los transistores, las compuertas pueden modelarse con exactitud como dispositivos digitales. Cada compuerta tiene una o más entradas digitales (señales que representan 0 o 1) y para generar su salida calcula alguna función sencilla de dichas entradas, como AND u OR. Cada compuerta contiene unos cuantos transistores como máximo. Podemos combinar pocas compuertas para formar una memoria de un bit, capaz de almacenar un 0 o un 1. Las memorias de un bit pueden combinarse en grupos de (por ejemplo) 16, 32 o 64 para formar registros. Cada **registro** puede contener un solo número binario menor que cierto valor límite. Las compuertas también pueden combinarse para formar la máquina calculadora principal misma. Examinaremos las compuertas y el nivel de lógica digital con detalle en el capítulo 3.

El siguiente nivel hacia arriba es el **nivel de microarquitectura**. En este nivel vemos una colección de (típicamente) 8 a 32 registros que forman una memoria local y un circuito llamado **ALU** (**unidad de aritmética lógica**, *Arithmetic Logic Unit*) que puede efectuar operaciones aritméticas sencillas. Los registros se conectan a la ALU para formar una **trayectoria de datos** por donde fluyen los datos. La operación básica de la trayectoria de datos consiste en seleccionar uno o dos registros, hacer que la ALU opere con ellos (sumándolos, por ejemplo), y almacenar después el resultado en algún registro.

En algunas máquinas un programa llamado **microprograma** controla la operación de la trayectoria de datos. En otras máquinas la trayectoria de datos está bajo el control directo del hardware. En ediciones anteriores de este libro llamamos a este nivel el "nivel de microprograma ción" porque en el pasado casi siempre se trataba de un intérprete en software. Puesto que ahora es común que el hardware controle directamente la trayectoria de datos, hemos cambiado el nombre para que refleje este hecho.

En las máquinas con control por software de la trayectoria de datos, el microprograma es un intérprete de las instrucciones en el nivel 2: obtiene, examina y ejecuta las instrucciones una por una, utilizando la trayectoria de datos para hacerlo. Por ejemplo, para una instrucción ADD (sumar), se obtendría la instrucción, se localizarían sus operandos y se colocarían en registros, la ALU calcularía la suma, y por último el resultado se enviaría al lugar correcto. En una máquina con control por hardware de la trayectoria de datos, se llevarían a cabo pasos similares, pero sin un programa almacenado explícito que controle la interpretación de las instrucciones del nivel 2.

En el nivel 2 tenemos un nivel que llamaremos **nivel de arquitectura del conjunto de instrucciones** (*Instruction Set Architecture*) o **nivel ISA**. Cada fabricante de computadoras publica un manual para cada una de las computadoras que vende, intitulado "Manual de referencia del lenguaje de máquina" o "Principios de operación de la computadora Western Wombat Model 100X" o algo parecido. Estos manuales se ocupan realmente del nivel ISA, no de los niveles subyacentes. Cuando describen el conjunto de instrucciones de la máquina, estos manuales están describiendo realmente las instrucciones que el microprograma o los circuitos de ejecución en hardware ejecutan de forma interpretativa. Si un fabricante de computadoras incluye dos intérpretes en una de sus máquinas, para interpretar dos niveles ISA distintos, tendrá que proporcionar dos manuales de referencia del "lenguaje de máquina", uno para cada intérprete.

El siguiente nivel suele ser un nivel híbrido. Casi todas las instrucciones de su lenguaje están también en el nivel ISA. (No hay razón para que una instrucción que aparece en un nivel no pueda estar presente también en otros niveles.) Además, hay un nuevo conjunto de instrucciones, una diferente organización de memoria, la capacidad para ejecutar dos o más programas al mismo tiempo, y varias características más. Entre los distintos diseños de nivel 3 hay más variación que entre los de los niveles 1 o 2.

Las nuevas funciones que se añaden en el nivel 3 son desempeñadas por un intérprete que se ejecuta en el nivel 2, que históricamente se conoce como sistema operativo. El microprograma (o el control por hardware), no el sistema operativo, ejecuta directamente las instrucciones del nivel 3 que son idénticas a las del nivel 2. En otras palabras, algunas de las instrucciones del nivel 3 son interpretadas por el sistema operativo y otras son interpretadas directamente por el microprograma. A esto es a lo que nos referimos con el término "híbrido". Llamamos a este nivel el **nivel de máquina del sistema operativo**.

Existe una discontinuidad fundamental entre los niveles 3 y 4. Los tres niveles más bajos no están diseñados para que sean usados por un programador ordinario. Su propósito primordial es la ejecución de los intérpretes y traductores que se necesitan para apoyar a los niveles superiores. Estos intérpretes y traductores son escritos por personas llamadas **programadores de sistemas** que se especializan en el diseño e implementación de máquinas virtuales nuevas. Los niveles 4 y superiores pertenecen a los programadores de aplicaciones que tienen un problema que resolver.

Otro cambio que ocurre en el nivel 4 tiene que ver con el método de apoyo de los niveles superiores. Los niveles 2 y 3 siempre se interpretan. Los niveles del 4 en adelante por lo regular se traducen, aunque no siempre es así.

Una diferencia más entre los niveles 1, 2 y 3, por un lado, y los niveles del 4 en adelante, por el otro, es la naturaleza del lenguaje empleado. Los lenguajes de máquina de los niveles 1, 2 y 3 son numéricos. Los programas escritos en ellos constan de largas series de números, lo cual es magnífico para las máquinas pero malo para las personas. A partir del nivel 4, los lenguajes contienen palabras y abreviaturas que tienen un significado para las personas.

El nivel 4, el nivel de lenguaje ensamblador, es en realidad una forma simbólica de uno de los lenguajes subyacentes. Este nivel ofrece a las personas un método de escribir programas para los niveles 1, 2 y 3 en una forma no tan incomprensible como los lenguajes de máquina virtuales. Los programas en lenguaje ensamblador primero se traducen a un lenguaje de nivel 1, 2 o 3 y luego se interpretan por la máquina virtual o real apropiada. El programa que realiza la traducción se llama **ensamblador**.

El nivel 5 por lo regular consta de lenguajes diseñados para ser usados por programadores de aplicaciones que quieren resolver problemas. Tales lenguajes suelen llamarse **lenguajes de alto nivel**, y hay literalmente cientos de ellos. Entre los más conocidos están BASIC, C, C++, Java, LISP y Prolog. Los programas escritos en estos lenguajes generalmente se traducen a lenguajes de nivel 3 o 4 con traductores llamados **compiladores**, aunque ocasionalmente se interpretan en vez de traducirse. Los programas en Java, por ejemplo, a menudo se interpretan.

En algunos casos, el nivel 5 consiste en un intérprete para un dominio de aplicación específico, como las matemáticas simbólicas. El nivel proporciona datos y operaciones para resolver problemas de este dominio en términos que la gente que conoce el dominio puede entender fácilmente.

En síntesis, lo más importante que debemos recordar es que las computadoras se diseñan como una serie de niveles, cada uno construido sobre sus predecesores. Cada nivel representa una abstracción distinta, y contiene diferentes objetos y operaciones. Al diseñar y analizar las computadoras de esta manera, podemos suprimir temporalmente los detalles que no son pertinentes y así reducir un tema complejo a algo más fácil de entender.

El conjunto de tipos de datos, operaciones y características de cada nivel es su **arquitectura**. La arquitectura se ocupa de los aspectos que el usuario de ese nivel puede ver. Las características que el programador ve, como la cantidad de memoria disponible, forman parte de la arquitectura. Los aspectos de implementación, como el tipo de tecnología de chips empleado para implementar la memoria, no forman parte de la arquitecutra. El estudio del diseño de las partes de un sistema de cómputo que los programadores pueden ver se llama **arquitectura de computadoras**. En la práctica común, la arquitectura de las computadoras y la organización de las computadoras significan prácticamente lo mismo.

1.1.3 Evolución de las máquinas multinivel

Para entender mejor las máquinas multinivel, examinaremos brevemente su desarrollo histórico, mostrando cómo han evolucionado al paso de los años el número y la naturaleza de los niveles. Los programas escritos en el verdadero lenguaje de máquina de una computadora (nivel 1) pueden ser ejecutados directamente por los circuitos electrónicos de la máquina (nivel 0), sin intervención de intérpretes ni traductores. Estos circuitos electrónicos, junto con la memoria y los dispositivos de entrada/salida, constituyen el **hardware** de la computadora. El hardware consta de objetos tangibles –circuitos integrados, tarjetas de circuitos impresos, cables, fuentes de alimentación, memorias e impresoras– no de ideas abstractas, algoritmos o instrucciones.

El **software**, en cambio, consta de **algoritmos** (instrucciones detalladas que indican cómo hacer algo) y sus representaciones en la computadora: los programas. Los programas se pueden almacenar en un disco duro, disco flexible, CD-ROM u otro medio, pero la esencia del software es la serie de instrucciones que constituye los programas, no los medios físicos en los que se registran.

En las primeras computadoras, la frontera entre hardware y software era muy nítida, pero con el tiempo se ha vuelto mucho más borrosa, principalmente por la adición, eliminación y fusión de niveles a medida que las computadoras evolucionaban. Hoy día, a veces es difícil distinguirlos. De hecho, un tema central de este libro es

El hardware y el software son lógicamente equivalentes.

Cualquier operación realizada por software también puede incorporarse directamente en hardware, de preferencia una vez que se ha entendido muy bien. Como dijo Karen Panetta Lenz: "El hardware no es más que software petrificado." Desde luego, lo contrario también es cierto: cualquier ejecución realizada por el hardware también puede simularse en software. La decisión de colocar ciertas funciones en hardware y otras en software se basa en factores como costo, rapidez, confiabilidad y frecuencia de cambios previstos. Existen pocas reglas rígidas para determinar si X debe incluirse en el hardware y Y debe programarse explícitamente. Tales decisiones cambian con las tendencias en la tecnología y el uso de las computadoras.

La invención de la microprogramación

Las primeras computadoras digitales, en los años cuarenta, sólo tenían dos niveles: el ISA, en el que se efectuaba toda la programación, y el de lógica digital, que ejecutaba esos programas. Los circuitos del nivel de lógica digital eran complicados, difíciles de entender y construir, y poco confiables.

En 1951 Maurice Wilkes, investigador de la Universidad de Cambridge, sugirió la idea de diseñar una computadora de tres niveles a fin de simplificar drásticamente el hardware (Wilkes, 1951). Esta máquina tendría un intérprete inmutable integrado (el microprograma) cuya función sería ejecutar programas del nivel ISA por interpretación. Puesto que el hardware ahora sólo tendría que ejecutar microprogramas, que tienen un conjunto limitado de instrucciones, en lugar de programas del nivel ISA, que tienen un conjunto de instrucciones mucho más grande, se requerirían menos circuitos electrónicos. Dado que en esa época los circuitos se construían con bulbos, tal simplificación prometía reducir el número de bulbos y por tanto mejorar la confiabilidad.

Durante los cincuenta se construyeron unas cuantas máquinas de tres niveles. En los años sesenta se construyeron más, y para 1970 la idea de interpretar el nivel ISA con un microprograma, y no directamente con los circuitos, era preponderante.

La invención del sistema operativo

En estos primeros años, la mayor parte de las computadoras eran de "taller abierto", lo que significa que el programador tenía que operar la máquina personalmente. Junto a cada máquina había una hoja para reservar. Un programador que quería ejecutar un programa reservaba cierto tiempo, digamos el miércoles de las 3 A.M. a las 5 A.M. (a muchos programadores les gustaba trabajar cuando el cuarto de máquinas estaba tranquilo). Al llegar la hora, el programador se dirigía al cuarto de máquinas con un paquete de tarjetas perforadas de 80 columnas (un medio de entrada primitivo) en una mano y un lápiz afilado en la otra. Al llegar al cuarto de la computadora, conducía amablemente al programador anterior hacia la puerta y se hacía cargo de la computadora.

Si el programador quería ejecutar un programa en FORTRAN, era necesario realizar los pasos siguientes:

1. Él† se dirigía al mueble en el que se guardaba la biblioteca de programas, sacaba el gran paquete verde rotulado "compilador FORTRAN", lo colocaba en el lector de tarjetas, y oprimía el botón de inicio.
2. El programador colocaba el programa en FORTRAN en el lector de tarjetas y oprimía el botón de continuar. Se leía el programa.
3. Una vez que la computadora se detenía, él hacía que se leyera otra vez el programa en FORTRAN. Aunque algunos compiladores sólo requerían leer una vez sus entradas, muchos requerían varias pasadas. En cada pasada era necesario leer un paquete grande de tarjetas.
4. Por último, se llegaba a la etapa final de la traducción. El programador a menudo se ponía nervioso a estas alturas porque si el compilador encontraba un error en el programa, el programador tenía que corregirlo e iniciar otra vez el proceso. Si no había errores, el compilador perforaba el programa traducido a lenguaje de máquina en otras tarjetas.
5. Luego, el programador colocaba el programa en lenguaje de máquina en el lector de tarjetas junto con el paquete de la biblioteca de subrutinas para que se leyeran.

†"Él" debe leerse como "él o ella" en todo el libro.

6. El programa comenzaba a ejecutarse. Las más de las veces, no funcionaba y se detenía inesperadamente antes de terminar. En general, el programador movía los interruptores de la consola y miraba las luces de la consola durante un rato. Si tenía suerte, encontraba el problema, corregía el error, y regresaba al mueble que contenía el gran paquete verde del compilador FORTRAN para comenzar otra vez desde el principio. Si no tenía suerte, preparaba un listado del contenido de la memoria, llamado **vaciado de núcleo** (*core dump*) y se lo llevaba a casa para estudiarlo.

Este procedimiento, con variaciones menores, fue normal en muchos centros de cómputo durante años; obligaba a los programadores a aprender a operar la máquina y a saber qué hacer cuando tenía un desperfecto, que era muy seguido. La máquina pasaba mucho tiempo ociosa mientras la gente llevaba tarjetas de un lado a otro del recinto o se rascaba la cabeza tratando de averiguar por qué sus programas no estaban funcionando correctamente.

Alrededor de 1960 la gente trató de reducir el desperdicio de tiempo automatizando la tarea del operador. Un programa llamado **sistema operativo** estaba cargado en la computadora todo el tiempo. El programador incluía ciertas tarjetas de control junto con el programa, y el sistema operativo las leía y ejecutaba sus instrucciones. En la figura 1-3 se muestra un ejemplo de paquete de tarjetas para uno de los primeros sistemas operativos de mayor uso, FMS (FORTRAN Monitor System), en la IBM 709.

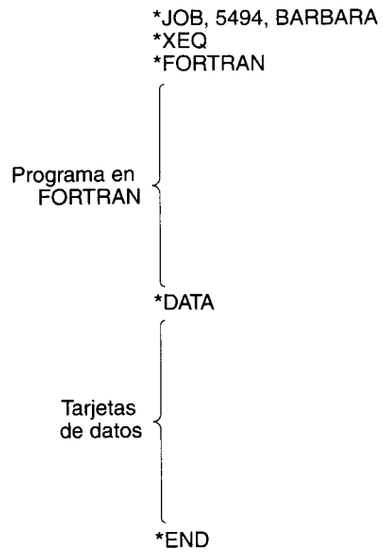


Figura 1-3. Ejemplo de trabajo para el sistema operativo FMS.

El sistema operativo leía la tarjeta *JOB y utilizaba la información que contenía para fines de contabilidad. (El asterisco identificaba las tarjetas de control, para que la computadora no las confundiera con las tarjetas del programa y de datos.) Después, el sistema operativo leía la tarjeta *FORTRAN, que era una instrucción para cargar el compilador FORTRAN de una cinta magnética. Luego, el compilador leía y compilaba el programa en FORTRAN. Cuando el compilador terminaba, devolvía el control al sistema operativo, que entonces leía la tarjeta *DATA. Ésta era una instrucción para ejecutar el programa traducido, utilizando las tarjetas que seguían a la tarjeta *DATA como datos.

Aunque el sistema operativo se diseñó para automatizar la tarea del operador (de ahí el nombre), también fue el primer paso en la creación de una nueva máquina virtual. La tarjeta *FORTRAN podía verse como una instrucción "compilar programa" virtual. Así mismo, la tarjeta *DATA podía considerarse como una instrucción "ejecutar programa" virtual. Un nivel con sólo dos instrucciones no era mucho, pero era un paso en esa dirección.

En años subsecuentes, los sistemas operativos se volvieron más y más sofisticados. Se añadieron nuevas instrucciones, recursos y características al nivel ISA hasta que comenzó a parecer un nuevo nivel. Algunas de las instrucciones de este nuevo nivel eran idénticas a las del nivel ISA, pero otras, sobre todo las de entrada/salida, eran totalmente distintas. Las nuevas instrucciones se conocían como **macros del sistema operativo** o **llamadas al supervisor**. El término usual ahora es **llamada al sistema**.

Los sistemas operativos evolucionaron también en otros sentidos. Los primeros leían paquetes de tarjetas e imprimían sus salidas en la impresora en línea. Esta organización se llamaba **sistema por lotes**. Por lo regular, había una espera de varias horas entre el momento en que se presentaba un programa y el momento en que los resultados estaban listos. Desarrollar software era difícil en esas circunstancias.

A principios de los años sesenta investigadores del Dartmouth College, MIT, y otros sitios crearon sistemas operativos que permitían a (varios) programadores comunicarse directamente con la computadora. En esos sistemas, terminales remotas se conectaban a la computadora central a través de líneas telefónicas. Muchos usuarios compartían el mismo CPU. Un programador podía introducir un programa con el teclado y obtener los resultados impresos casi de inmediato en su oficina, en la cochera de su casa o en donde fuera que se instalara la terminal. Esos sistemas se llamaban, y se siguen llamando, **sistemas de tiempo compartido**.

Nuestro interés en los sistemas operativos se centra en las partes que interpretan las instrucciones y en las características que están presentes en el nivel 3 pero no en el nivel ISA, más que en los aspectos de tiempo compartido. Aunque no haremos hincapié en ello, el lector debe tener presente que los sistemas operativos hacen más que sólo interpretar funciones añadidas al nivel ISA.

Migración de la funcionalidad al microcódigo

Una vez que se hizo común la microprogramación (hacia 1970), los diseñadores se dieron cuenta de que podían añadir nuevas instrucciones con sólo extender el microprograma. En otras palabras, podían agregar "hardware" (nuevas instrucciones de máquina) programando. Esta revelación dio pie a una verdadera explosión en los conjuntos de instrucciones de máquina, a medida que los diseñadores trataban de producir conjuntos de instrucciones más grandes y

mejores que los de otros diseñadores. Muchas de esas instrucciones no eran indispensables en el sentido de que su efecto podía lograrse fácilmente con las instrucciones existentes, pero a menudo eran un poco más rápidas que una sucesión de instrucciones existentes. Por ejemplo, muchas instrucciones tenían una instrucción INC (INCrementar) que sumaba 1 a un número. Puesto que esas máquinas también tenían una instrucción general ADD para sumar, tener una instrucción especial para sumar 1 (o para sumar 270, para el caso) no era necesario. Sin embargo, INC era por lo regular un poco más rápida que ADD, así que se incluyó.

Se añadieron muchas otras instrucciones al microprograma siguiendo el mismo razonamiento. Entre ellas estaban

1. Instrucciones para multiplicación y división de enteros.
2. Instrucciones para aritmética de punto flotante.
3. Instrucciones para invocar procedimientos y regresar de ellos.
4. Instrucciones para acelerar los ciclos.
5. Instrucciones para manejar cadenas de caracteres.

Además, una vez que los diseñadores de máquinas vieron lo fácil que era añadir nuevas instrucciones, comenzaron a buscar otras funciones que añadir a sus microprogramas. Ejemplos de tales adiciones son

1. Funciones para acelerar cálculos con matrices (direccionamiento indexado e indirecto).
2. Funciones para trasladar programas de un lugar a otro de la memoria después de que han iniciado su ejecución (recursos de reubicación).
3. Sistemas de interrupciones que avisan a la computadora cuando se termina una operación de entrada o salida.
4. La capacidad para suspender un programa e iniciar otro en un número reducido de instrucciones (conmutación de procesos).

Con los años, se han añadido muchas otras funciones y recursos, casi siempre para acelerar alguna actividad específica.

La eliminación de la microprogramación

Los microprogramas crecieron durante los años dorados de la microprogramación (años sesenta y setenta). El único problema fue que también tendían a volverse más y más lentos a medida que se hacían más voluminosos. Por fin algunos investigadores se dieron cuenta de que si eliminaban el microprograma, reducían considerablemente el conjunto de instrucciones y hacían que las instrucciones restantes se ejecutaran en forma directa (es decir, control por hardware de la trayectoria de datos), las máquinas podían acelerarse. En cierto sentido, el

diseño de computadoras había descrito un círculo completo, volviendo a la forma que tenía antes de que Wilkes inventara la microprogramación.

El punto importante de esta explicación es mostrar que la frontera entre hardware y software es arbitraria y cambia constantemente. El software de hoy podría ser el hardware de mañana, y viceversa. Además, las fronteras entre los diversos niveles también son variables. Desde el punto de vista del programador, la forma en que se implementa una instrucción dada no es importante (excepto tal vez por su velocidad). Una persona que programa en el nivel ISA puede utilizar su instrucción de multiplicar como si fuera una instrucción de hardware sin preocuparse por ello, o sin siquiera saber si en efecto es una instrucción de hardware. El hardware de una persona es el software de otra. Regresaremos a estos temas más adelante.

1.2 ACONTECIMIENTOS IMPORTANTES EN ARQUITECTURA DE COMPUTADORAS

Se han diseñado y construido cientos de tipos distintos de computadoras durante la evolución de la computadora digital moderna. Casi todos se han perdido en el olvido, pero unos cuantos han tenido un impacto importante sobre las ideas modernas. En esta sección bosquejaremos brevemente algunos de los sucesos históricos clave para entender mejor cómo llegamos a donde estamos ahora. No es necesario decir que esta sección sólo trata los puntos principales y deja muchas cosas sin mencionar. En la figura 1-4 se lee una lista de algunas de las máquinas que trataremos en esta sección. Slater (1987) presenta material histórico adicional acerca de las personas que inauguraron la era de las computadoras. En la obra de Morgan (1997), se encuentran biografías breves, ilustradas con bellas fotografías a color de esos fundadores, tomadas por Louis Fabian Bachrach. Otro libro de biografías es (Slater, 1987).

1.2.1 La generación cero – computadoras mecánicas (1642-1945)

La primera persona que construyó una máquina calculadora funcional fue el científico francés Blaise Pascal (1623-1662), en cuyo honor se nombró el lenguaje de programación Pascal. Este dispositivo, construido en 1642, cuando Pascal tenía apenas 19 años, fue diseñado para ayudar a su padre, un cobrador de impuestos del gobierno francés. La calculadora era totalmente mecánica, con engranes, y se impulsaba con una manivela operada a mano.

La máquina de Pascal sólo podía sumar y restar, pero 30 años después el gran matemático alemán Barón Gottfried Wilhelm von Leibniz (1646-1716) construyó otra máquina mecánica que también podía multiplicar y dividir. Efectivamente, Leibniz había construido el equivalente de una calculadora de bolsillo de cuatro funciones, hace tres siglos.

No sucedió algo relevante durante 150 años hasta que un profesor de matemáticas de la University of Cambridge, Charles Babbage (1792-1871), inventor del velocímetro, diseñó y construyó su **máquina de diferencias**. Este aparato mecánico, que al igual que el de Pascal sólo podía sumar y restar, fue diseñado para calcular tablas de números útiles para la navegación marítima. La máquina entera estaba diseñada para ejecutar un solo algoritmo, el método

Año	Nombre	Hecho por	Comentario
1834	Máquina analítica	Babbage	Primer intento de construir una computadora digital
1936	Z1	Zuse	Primera máquina calculadora de relevadores funcional
1943	COLOSSUS	Gobierno inglés	Primera computadora electrónica
1944	Mark I	Aiken	Primera computadora estadounidense de propósito deneral
1946	ENIAC I	Eckert/Mauchley	Inicia la hitoria moderna de la computación
1949	EDSAC	Wilkes	Primera computadora de programa almacenado
1951	Whielwind I	M.I.T.	Primera computadora de tiempo lineal
1952	IAS	Von Neumann	Casi todas las máqinas actuales emplean este diseño
1960	PDP-1	DEC	Primera minicomputadora (50 vendidas)
1961	1401	IBM	Máquina pequeña para ngocios de enorme popularidad
1962	7094	IBM	Dominó la computación científica a principios de los añossesenta
1963	B5000	Burroughs	Primera máquina diseñada para un lenguaje de alto nivel
1964	360	IBM	Primera línea de productos diseñada como familia
1964	6600	CDC	Primera supercomputadora cietífica
1965	PDP-8	DEC	Primera minicomputadora con mercado masivo (50,000 vedidas)
1970	PDP-11	DEC	Dominó las minicomputadoras en los años setenta
1974	8080	Intel	Primera computadora de propósito general de 8 bits en un chip
1974	CRAY-1	Cray	Primera supercomputadora vectorial
1978	VAX	DEC	Primera superminicomputadora de 32 bits
1981	IBM PC	IBM	Inició la era de la computadora personal moderna
1985	MIPS	MIPS	Primera máquina RISC comercial
1987	SPARC	Sun	Primera estación de trabajo RISC basada en SPARC
1990	RSC6000	IBM	Primera máquina superescalar

Figur a 1-4. Algunos acontecimientos importantes en la evolución de la computadora digital moderna.

de diferencias finitas empleando polinomios. La característica más interesante de la máquina de diferencias era su método de salida: perforaba sus resultados en una placa de cobre para grabados con un troquel de acero, un precursor de posteriores medios de escritura única como las tarjetas perforadas y los CD-ROM.

Aunque la máquina de diferencias funcionaba razonablemente bien, Babbage pronto se aburrió de una máquina que sólo podía ejecutar un algoritmo. Él comenzó a dedicar cantidades cada vez más grandes de su tiempo y de la fortuna de su familia (sin mencionar 17,000 libras esterlinas de dinero del gobierno) en el diseño y construcción de una sucesora llamada **máquina analítica**. La máquina analítica tenía cuatro componentes: el almacén (memoria), el molino (unidad de cómputo), la sección de entrada (lector de tarjetas perforadas) y la

sección de salida (salidas perforadas e impresas). El almacén consistía en 1000 palabras de 50 dígitos decimales, cada una de las cuales servía para almacenar variables y resultados. El molino podía aceptar operandos del almacén, y luego sumarlos, restarlos, multiplicarlos o dividirlos, y por último devolver el resultado al almacén. Al igual que la máquina de diferencias, la máquina analítica era totalmente mecánica.

El gran avance de la máquina analítica fue su utilidad general: leía instrucciones de tarjetas perforadas y las ejecutaba. Algunas instrucciones ordenaban a la máquina obtener dos números del almacén, llevarlos al molino, efectuar operaciones sobre ellos (por ejemplo, sumarlos) y enviar el resultado de vuelta al almacén. Otras instrucciones podían probar un número y efectuar una ramificación condicional dependiendo de si era positivo o negativo. Al perforar un programa distinto en las tarjetas de entrada, era posible lograr que la máquina analítica realizara diferentes cálculos, algo que no podía hacer la máquina de diferencias.

Puesto que la máquina analítica era programable en un sencillo lenguaje ensamblador, necesitaba software. Para producir este software, Babbage contrató a una joven mujer llamada Ada Augusta Lovelace, hija del afamado poeta británico Lord Byron. Así, Ada Lovelace fue la primera programadora de computadoras del mundo. El moderno lenguaje de programación Ada® honra su memoria.

Desafortunadamente, al igual que muchos diseñadores modernos, Babbage nunca logró eliminar todos los problemas del hardware. La dificultad es que necesitaba miles y miles de levas y engranes y ruedas fabricadas con un grado de precisión que la tecnología del siglo XIX no podía alcanzar. No obstante, sus ideas se habían adelantado mucho a su época, e incluso hoy la mayor parte de las computadoras modernas tienen una estructura muy similar a la de la máquina analítica, por lo que es justo decir que Babbage fue el padre (o el abuelo) de la computadora digital moderna.

El siguiente avance importante ocurrió a fines de la década de los treinta, cuando un estudiante de ingeniería alemán llamado Konrad Zuse construyó una serie de máquinas calculadoras automáticas empleando relevadores electromagnéticos. Zuse no logró obtener financiamiento del gobierno después de iniciada la guerra porque los burócratas esperaban ganar la guerra con tal rapidez que la nueva máquina no estaría lista antes de terminar las acciones. Zuse no conocía el trabajo de Babbage, y sus máquinas fueron destruidas por el bombardeo aliado de Berlín en 1944, por lo que su trabajo no influyó en las máquinas subsecuentes. No obstante, Zuse fue uno de los pioneros en este campo.

Poco tiempo después, en Estados Unidos, dos personas diseñaron también calculadoras. John Atanasoff del Iowa State College y George Stibbitz de Bell Labs. La máquina de Atanasoff era asombrosamente avanzada para su época; utilizaba aritmética binaria y tenía una memoria de condensadores, los cuales se renovaban periódicamente para evitar que la carga desapareciera, en un proceso que él llamaba "refrescar la memoria". Los modernos chips de memoria dinámica (RAM) funcionan exactamente de la misma manera. Por desgracia, la máquina nunca logró entrar en operación. En cierto modo, Atanasoff era como Babbage: un visionario que finalmente fue derrotado por la insuficiente tecnología de hardware de su época.

La computadora de Stibbitz, aunque más primitiva que la de Atanasoff, sí funcionaba. Stibbiz hizo una demostración pública en una conferencia en Dartmouth College en 1940. Entre el públi-

co estaba John Mauchley, un profesor de física desconocido de la Universidad de Pennsylvania. El mundo de las computadoras oiría más acerca del profesor Mauchley posteriormente.

Mientras Zuse, Stibbitz y Atanasoff diseñaban calculadoras automáticas, un joven llamado Howard Aiken realizaba tediosos cálculos numéricos a mano como parte de su tesis de doctorado en Harvard. Después de graduarse, Aiken reconoció la importancia de poder efectuar cálculos con una máquina. Él acudió a la biblioteca, descubrió los trabajos de Babbage, y decidió construir con relevadores la computadora de propósito general que Babbage no había logrado construir con ruedas dentadas.

La primera máquina de Aiken, el Mark I, se completó en Harvard en 1944. Tenía 72 palabras de 23 dígitos decimales cada una, y un tiempo de instrucción de 6 segundos. Las entradas y salidas se efectuaban mediante una cinta de papel perforada. Para cuando Aiken terminó su sucesor, el Mark II, las computadoras de relevadores eran obsoletas. Se había iniciado la era de la electrónica.

1.2.2 La primera generación – bulbos (1945-1955)

El estímulo para la computadora electrónica fue la Segunda Guerra Mundial. Durante la primera parte de la guerra, submarinos alemanes hacían estragos entre la armada británica. Los almirantes alemanes enviaban órdenes por radio hasta los submarinos, pero eran interceptadas por los ingleses. El problema es que los mensajes se codificaban mediante un aparato llamado **ENIGMA**, cuyo precursor había sido diseñado por un inventor aficionado y expresidente de Estados Unidos, Thomas Jefferson.

A principios de la guerra, los espías británicos lograron adquirir una máquina ENIGMA de los espías polacos, quienes la habían robado a los alemanes. Sin embargo, se requería un número abrumador de cálculos para decodificar un mensaje, y esto tenía que hacerse inmediatamente después de interceptarse el mensaje para que sirviera de algo. Para decodificar estos mensajes, el gobierno británico estableció un laboratorio supersecreto que construyó una computadora electrónica llamada COLOSSUS. El famoso matemático británico Alan Turing ayudó a diseñar esta máquina. COLOSSUS entró en operación en 1943, pero como el gobierno británico mantuvo prácticamente todos los aspectos del proyecto clasificados como secreto militar durante 30 años, la línea de COLOSSUS fue básicamente un callejón sin salida. Sólo vale la pena mencionarlo porque fue la primera computadora electrónica digital del mundo.

Además de destruir las máquinas de Zuse y estimular la construcción de COLOSSUS, la guerra también afectó la computación en Estados Unidos. El ejército necesitaba tablas de alcance para apuntar su artillería pesada, y producía esas tablas contratando a cientos de mujeres que las elaboraban utilizando calculadoras manuales (se pensaba que las mujeres eran más precisas que los hombres). No obstante, el proceso era muy tardado y los errores eran frecuentes.

John Mauchley, quien conocía el trabajo de Atanasoff y de Stibbitz, se dio cuenta de que el ejército estaba interesado en calculadoras mecánicas. Al igual que muchos científicos de la computación que le siguieron, Mauchley elaboró una propuesta de subvención en la que pedía al ejército fondos para construir una computadora electrónica. La propuesta fue aceptada en 1943, y Mauchley y su estudiante de posgrado, J. Presper Eckert, procedieron a construir una computadora electrónica a la que llamaron **ENIAC** (***Electronic Numerical Integrator And Computer***). ENIAC consistía en 18,000 bulbos y 1500 relevadores, pesaba 30 toneladas y consumía 140 kilowatts de potencia. En términos de arquitectura, la máquina tenía 20 registros, cada uno capaz de almacenar un número decimal de 10 dígitos. (Un registro decimal es una memoria muy pequeña que puede almacenar un número menor que cierto número máximo de dígitos decimales, parecido al odómetro que registra la distancia recorrida por un automóvil.) ENIAC se programaba ajustando 6000 interruptores de multiposición y conectando numerosas bases con una verdadera maraña de cables interconectores.

La máquina no quedó terminada sino hasta 1946, demasiado tarde para realizar su tarea original. Sin embargo, como la guerra había terminado, se les permitió a Mauchley y Eckert organizar un curso de verano para describir el trabajo a sus colegas científicos. Ese curso de verano fue el principio del auge del interés en construir computadoras digitales grandes.

Después de ese histórico curso de verano, muchos otros investigadores se propusieron construir computadoras electrónicas. La primera que entró en operación fue EDSAC (1949), construida por Maurice Wilkes en la Universidad de Cambridge. Otras fueron JOHNIAC, de la Rand Corporation, ILLIAC de la Universidad de Illinois, MANIAC de Los Alamos Laboratory y WEIZAC del Instituto Weizmann de Israel.

Eckert y Mauchley pronto comenzaron a trabajar en un sucesor, **EDVAC** (***Electronic Discrete Variable Automatic Computer***). Sin embargo, ese proyecto quedó condenado al fracaso cuando ellos dejaron la Universidad de Pennsylvania para formar una compañía nueva, la Eckert-Mauchley Computer Corporation, en Filadelfia (todavía no se había creado Silicon Valley). Después de una serie de fusiones, esta compañía se convirtió en la moderna Unisys Corporation.

Como dato curioso, Eckert y Mauchley solicitaron una patente asegurando haber inventado la computadora digital. En retrospectiva, habría sido bueno tener una patente así. Después de años de pleitos, las cortes decidieron que la patente de Eckert y Mauchley no era válida y que John Atanasoff era el inventor de la computadora digital, aunque nunca la haya patentado.

Mientras Eckert y Mauchley estaban trabajando en la EDVAC, una de las personas que participaron en el proyecto ENIAC, John von Neumann, acudió al Institute of Advanced Studies de Princeton para construir su propia versión de EDVAC, la **máquina IAS**. Von Neuman fue un genio del calibre de Leonardo da Vinci; hablaba muchos idiomas, era experto en las ciencias físicas y en matemáticas, y recordaba perfectamente todo lo que había escuchado, visto o leído. Podía citar de memoria el texto exacto de libros que había leído hacía años. Cuando se interesó por las computadoras, ya era el matemático más eminente del mundo.

Una cosa que pronto fue obvia para él era que programar computadoras con un gran número de interruptores era lento, tedioso e inflexible. Von Neumann se dio cuenta de que el programa podía representarse en forma digital en la memoria de la computadora, junto con los datos. Él percibió también que la torpe aritmética decimal en serie utilizada por ENIAC, en la que cada dígito estaba representado por 10 bulbos (uno encendido y nueve apagados) podía ser sustituida por una aritmética binaria.

El diseño básico, que él describió por primera vez, ahora se conoce como **máquina de von Neumann**. Se usó en EDSAC, la primera computadora de programa almacenado, y sigue siendo la base de casi todas las computadoras digitales aun ahora, más de medio siglo después. Este diseño, y la máquina IAS, construida en colaboración con Herman Goldstine, ha tenido una influencia de tal magnitud que vale la pena describirla brevemente. En la figura 1-5 se presenta un bosquejo de su arquitectura.

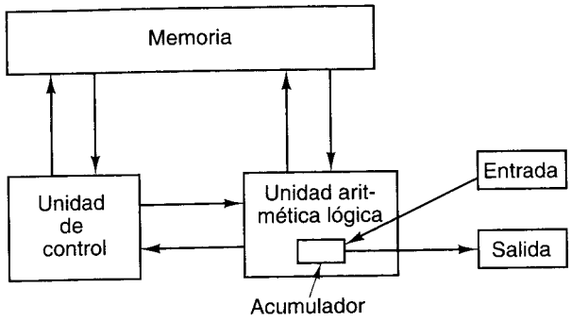


Figura 1-5. La máquina de von Neumann original.

La máquina de von Neumann tenía cinco partes básicas: la memoria, la unidad aritmética lógica, la unidad de control y el equipo de entrada y salida. La memoria constaba de 4096 palabras, cada una de las cuales contenía 40 bits que podían ser 0 o 1. Cada palabra contenía dos instrucciones de 20 bits o bien un entero con signo de 40 bits. Ocho bits de cada instrucción estaban dedicados a indicar el tipo de instrucción, y 12 bits se dedicaban a especificar una de las 4096 palabras de memoria.

Dentro de la unidad de aritmética lógica había un registro interno especial de 40 bits llamado **acumulador**. Una instrucción típica sumaba una palabra de la memoria al acumulador o almacenaba el contenido del acumulador en la memoria. La máquina no tenía aritmética de punto flotante porque von Neumann pensaba que cualquier matemático competente debería poder seguir mentalmente la pista al punto decimal (en realidad el punto binario).

Más o menos en las mismas fechas en que von Neumann estaba construyendo la máquina IAS, investigadores de MIT también estaban construyendo una computadora. A diferencia de IAS, ENIAC y otras máquinas de su clase, que empleaban palabras largas y estaban diseñadas para manipulación pesada de números, la máquina del MIT, Whirlwind I, tenía palabras de 16 bits y estaba diseñada para el control en tiempo real. Este proyecto dio pie a la invención de la memoria de núcleos magnéticos por Jay Forrester, y finalmente a la primera minicomputadora comercial.

Mientras todo esto estaba sucediendo, IBM era una compañía pequeña dedicada a fabricar perforadoras de tarjetas y máquinas clasificadoras de tarjetas. Aunque IBM había aportado parte del financiamiento de Aiken, no estaba muy interesada en las computadoras hasta que produjo la 701 en 1953, mucho después de que la compañía de Eckert y Mauchley se había convertido en la número uno del mercado comercial con su computadora UNIVAC.

La 701 tenía 2048 palabras de 36 bits, con dos instrucciones por palabra. Era la primera de una serie de máquinas científicas que llegaron a dominar la industria en el plazo de una década. Tres años después apareció la 704, que tenía 4K de memoria de núcleos, instrucciones de 36 bits y hardware de punto flotante. En 1958 IBM inició la producción de su última máquina de bulbos, la 709, que básicamente era una 704 más potente.

1.2.3 La segunda generación – transistores (1955-1965)

El transistor fue inventado en los Bell Labs en 1948 por John Bardeen, Walter Brattain y William Schockley, quienes fueron galardonados con el premio Nobel de física de 1956 por su trabajo. En un plazo de 10 años el transistor revolucionó a las computadoras, y para fines de la década de los cincuenta las computadoras de bulbos eran obsoletas. La primera computadora transistorizada se construyó en el Lincoln Laboratory del MIT, una máquina de 16 bits con un diseño parecido al de la Whirlwind I. Se le llamó **TX-0** (computadora Transistorizada eXperimental 0) y su único propósito era probar la TX-2, mucho más elegante.

La TX-2 no logró un progreso decisivo, pero uno de los ingenieros que trabajaban en el laboratorio, Kenneth Olsen, formó una compañía, Digital Equipment Corporation (DEC) en 1957 para fabricar una máquina comercial parecida a la TX-0. Tuvieron que pasar cuatro años para que apareciera esta máquina, la PDP-1, principalmente porque los inversionistas que financiaban a DEC creían firmemente que no había mercado para las computadoras. En vez de ello, DEC se dedicó primordialmente a vender pequeñas tarjetas de circuitos.

Cuando por fin la PDP-1 apareció en 1961, tenía 4K de palabras de 18 bits y un tiempo de ciclo de 5 μs.† Este desempeño era la mitad del de la IBM 7090, la sucesora transistorizada de la 709 y la computadora más rápida del mundo en esas fechas. La PDP-1 costaba $120,000 dólares; la 7090 costaba millones. DEC vendió docenas de PDP-1, y con esto la industria de las minicomputadoras había nacido.

Una de las primeras PDP-1 se entregó a MIT, donde rápidamente atrajo la atención de algunos de los jóvenes genios en formación tan comunes en esa institución. Una de las muchas innovaciones de la PDP-1 era una presentación visual y la capacidad de graficar puntos en cualquier lugar de su pantalla de 512 por 512. En poco tiempo, los estudiantes habían programado la PDP-1 para jugar guerras espaciales, y el mundo tuvo su primer juego de video.

Unos cuantos años después DEC introdujo la PDP-8, que era una máquina de 12 bits, pero mucho más económica que la PDP-1 ($16,000). La PDP-8 tenía una importante innovación: un bus único, u ómnibus, que se muestra en la figura 1-6. Un **bus** es un conjunto de alambres en paralelo que sirve para conectar los componentes de una computadora. Esta arquitectura marcó una divergencia importante respecto a la máquina IAS centrada en la memoria, y ha sido adoptada por casi todas las computadoras pequeñas posteriores. DEC llegó a vender 50,000 PDP-8, lo que la estableció como líder en el negocio de las minicomputadoras

†Prefijos métricos: mili = 10^{-3}, micro (μ) = 10^{-6}, nano = 10^{-9}; kilo = 10^{3}, mega = 10^{6}, giga = 10^{9}

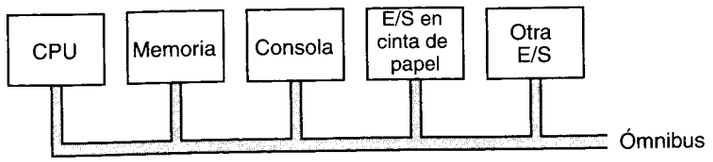


Figura 1-6. El ómnibus de la PDP-8.

Mientras tanto, la reacción de IBM ante el transistor fue construir una versión transistorizada de la 709, la 7090, como ya dijimos, y después la 7094. Ésta tenía un tiempo de ciclo de 2 μs y una memoria de núcleos de 32 K con palabras de 36 bits. La 7090 y la 7094 marcaron el fin de las máquinas tipo ENIAC, y dominaron la computación científica gran parte de la década de los sesenta.

Al mismo tiempo que IBM se convertía en un protagonista importante en la computación científica con la 7094, estaba ganando enormes cantidades de dinero vendiendo una pequeña máquina orientada hacia los negocios: la 1401. Esta máquina podía leer y grabar cintas magnéticas, leer y perforar tarjetas, e imprimir salidas casi tan rápidamente como la 7094, a una fracción del precio. Era muy mala para la computación científica, pero era perfecta para la contabilidad comercial.

La 1401 era poco común en cuanto a que no tenía registros, y ni siquiera una longitud de palabra fija. Su memoria consistía en 4K bytes de 8 bits (4 KB†). Cada byte contenía un carácter de 6 bits, un bit administrativo y un bit que servía para indicar el fin de una palabra. Una instrucción MOVE (mover), por ejemplo, tenía una dirección de origen y una de destino, y comenzaba a transferir bytes del origen al destino hasta encontrar uno cuyo bit de fin de palabra valía 1.

En 1964 una compañía nueva, Control Data Corporation (CDC), introdujo la 6600, una máquina que era casi un orden de magnitud más rápida que la poderosa 7094. Para los "trituradores de números" fue amor a primera vista, y CDC estaba camino al éxito. El secreto de su velocidad, la razón por la que era mucho más rápida que la 7094, era que dentro de la CPU había una máquina con alto grado de paralelismo: contaba con varias unidades funcionales para sumar, otras para multiplicar y otras para dividir, y todas podían operar en paralelo. Aunque para aprovecharla al máximo se requería una programación cuidadosa, con un poco de esfuerzo era posible lograr la ejecución simultánea de 10 instrucciones.

Como si esto no fuera suficiente, la 6600 tenía en su interior varias computadoras pequeñas que la ayudaban, algo así como Blanca Nieves y las siete personas con disfunción vertical. Esto permitía a la CPU dedicar todo su tiempo a la trituración de números, dejando todos los detalles de control de trabajos y entrada/salida a las computadoras pequeñas. En retrospectiva, la 6600 se había adelantado décadas a su época. Muchas de las ideas fundamentales en que se basan las computadoras modernas se remontan directamente a la 6600.

El diseñador de la 6600, Seymour Cray, fue una figura legendaria, comparable con von Neumann. Dedicó su vida a construir máquinas cada vez más rápidas, ahora llamadas

†$1K = 2^{10} = 1024$, así que 1 KB = 1024 bytes; también $1M = 2^{20} = 1{,}048{,}576$ bytes

supercomputadoras, entre las que figuraron la 6600, la 7600 y la Cray-1. También inventó un algoritmo para comprar automóviles que se ha hecho famoso: acudir a la concesionaria más cercana al hogar, apuntar al automóvil más cercano a la puerta, y decir: "Me llevo ése." Este algoritmo desperdicia el mínimo de tiempo en hacer cosas sin importancia (como comprar automóviles) y deja el máximo de tiempo libre para hacer cosas importantes (como diseñar supercomputadoras).

Hubo muchas otras computadoras en esta era, pero una sobresale por una razón muy distinta y vale la pena mencionarla: la Burroughs B5000. Los diseñadores de máquinas como la PDP-1, la 7094 y la 6600 se concentraban totalmente en el hardware, ya sea en bajar su costo (DEC) o en hacerlo más rápido (IBM y CDC). El software casi no venía al caso. Los diseñadores de la B5000 adoptaron una estrategia distinta: construyeron una máquina con la intención específica de programarla en Algol 60, un precursor del Pascal, e incluyeron en el hardware muchas funciones que facilitaban la tarea del compilador. Había nacido la idea de que el software también cuenta. Lo malo es que la idea cayó en el olvido casi de inmediato.

1.2.4 La tercera generación – circuitos integrados (1965-1980)

La invención del circuito integrado de silicio por Robert Noyce en 1958 hizo posible colocar docenas de transistores en un solo chip. Este empaquetamiento permitió construir computadoras más pequeñas, más rápidas y menos costosas que sus predecesores transistorizados. A continuación se describen algunas de las computadoras más importantes de esta generación.

Para 1964 IBM era el principal fabricante de computadoras y tenía un problema grave con sus dos máquinas de mayor éxito, la 7094 y la 1401: eran absolutamente incompatibles. Una era un triturador de números de alta velocidad que empleaba aritmética binaria en paralelo con registros de 36 bits, y la otra era un procesador de entrada/salida que parecía hacer más de lo que en verdad hacía y que utilizaba aritmética decimal en serie con palabras de longitud variable en la memoria. Muchos de sus clientes corporativos tenían ambas máquinas y no les gustaba la idea de mantener dos departamentos de programación distintos que nada tenían en común.

Cuando llegó la hora de sustituir estas dos series, IBM dio un paso radical: introdujo una sola línea de productos, la System/360, basada en circuitos integrados, diseñada para computación tanto científica como comercial. La System/360 contenía muchas innovaciones, siendo la más importante que era una familia de cerca de media docena de máquinas con el mismo lenguaje ensamblador y con tamaño y potencia crecientes. Una compañía podía reemplazar su 1401 por una 360 Model 30 y su 7094 por una 360 Model 75. La Model 75 era más grande y rápida (y más costosa), pero el software escrito para ella podía, en principio, ejecutarse en la Model 30. En la práctica, el software escrito para un modelo inferior se ejecutaba en un modelo superior sin problemas, pero al pasar de una máquina superior a una inferior existía la posibilidad de que el programa no cupiera en la memoria. Aun así, esto representó un avance importante respecto a la situación que prevalecía con la 7094 y la 1401. La idea de las familias de máquinas de inmediato fue bien recibida, y en unos cuantos años casi todos los fabricantes de computadoras tenían una familia de máquinas similares que cubrían una

amplia gama de precio y desempeño. En la figura 1-7 se muestran algunas características de la familia 360 inicial. Posteriormente se introdujeron otros modelos.

Propiedad	Modelo 30	Modelo 40	Modelo 50	Modelo 65
Desempeño relativo	1	3.5	10	21
Tiempo de ciclo (ns)	1000	625	500	250
Memoria máxima (KB)	64	256	256	512
Bytes obtenidos por ciclo	1	2	4	16
Número máximo de canales de datos	3	3	4	6

Figura 1-7. La oferta inicial de la línea de productos IBM 360.

Otra importante innovación de la 360 fue la **multiprogramación**: tener varios programas en la memoria a la vez de modo que mientras uno estaba esperando el término de una operación de entrada/salida otro podía realizar cálculos.

La 360 también fue la primera máquina que podía emular (simular) otras computadoras. Los modelos más pequeños podían emular la 1401, y los más grandes podían emular la 7094, para que los clientes pudieran seguir usando sus antiguos programas binarios sin modificación en tanto efectuaban la conversión a la 360. Algunos modelos ejecutaban los programas de la 1401 con tal rapidez (mucho mayor que en la 1401 misma) que muchos clientes nunca convirtieron sus programas.

La emulación era fácil en la 360 porque todos los modelos iniciales y casi todos los modelos posteriores eran microprogramados. Lo único que tenía que hacer IBM era escribir tres microprogramas: para el conjunto de instrucciones nativo de la 360, para el conjunto de instrucciones de la 1401 y para el conjunto de instrucciones de la 7094. Esta flexibilidad fue una de las razones principales por las que se introdujo la microprogramación.

La 360 resolvió el dilema de binaria paralela *versus* decimal en serie con un término medio: la máquina tenía 16 registros de 32 bits para la aritmética binaria, pero su memoria estaba organizada en bytes, como la de la 1401. Además, contaba con instrucciones seriales tipo 1401 para transferir registros de tamaño variable en la memoria.

Otra característica importante de la 360 fue un espacio de direcciones enorme (para esa época) de 2^{24} bytes (16 megabytes). Puesto que la memoria costaba varios dólares por byte en esos días, 16 megabytes parecía casi infinito. Desafortunadamente, la serie 360 fue seguida más adelante por la serie 370, la 4300, la 3080 y la 3090, todas las cuales empleaban la misma arquitectura. Para mediados de los ochenta, el límite de 16 megabytes se había convertido en un verdadero problema, e IBM tuvo que abandonar parcialmente la compatibilidad al cambiar a las direcciones de 32 bits que se necesitaban para direccionar la nueva memoria de 2^{32} bytes.

En retrospectiva, puede argumentarse que como esas máquinas tenían palabras y registros de 32 bits, probablemente debían haber tenido también direcciones de 32 bits, pero en esa época nadie podía imaginarse una máquina de 16 megabytes. Criticar a IBM por esta falta de visión es como criticar a un moderno fabricante de computadoras personales por tener direcciones de sólo 32 bits. En unos cuantos años las computadoras personales tal vez necesiten más de 4 GB de memoria, y entonces las direcciones de 32 bits serán intolerablemente pequeñas.

El mundo de las minicomputadoras también dio un gran paso en la tercera generación cuando DEC introdujo la serie PDP-11, un sucesor de la PDP-8 de 16 bits. En muchos sentidos, la serie PDP-11 era como el hermano menor de la serie 360, igual que la PDP-1 había sido como un hermanito de la 7094. Tanto la 360 como la PDP-11 tenían registros orientados hacia palabras y memoria orientada hacia bytes, y ambas abarcaban una gama muy amplia de precio y desempeño. La PDP-11 tuvo un éxito enorme, sobre todo en las universidades, y mantuvo la ventaja de DEC sobre los otros fabricantes de minicomputadoras.

1.2.5 La cuarta generación – integración a muy grande escala (1980-?)

Para los años ochenta la **VLSI** (**integración a muy grande escala**, *Very Large Scale Integration*) había hecho posible colocar primero decenas de miles, luego centenares de miles y por último millones de transistores en un solo chip. Este avance dio pie a computadoras más pequeñas y más rápidas. Antes de la PDP-1, las computadoras eran tan grandes y costosas que las compañías y universidades necesitaban departamentos especiales llamados **centros de cómputo** para operarlas. Con la llegada de la minicomputadora, un departamento podía comprar su propia computadora. Para 1980 los precios habían bajado tanto que una persona podía tener su propia computadora. Se había iniciado la era de la computadora personal.

Las computadoras personales se usaban de forma muy diferente que las computadoras grandes; se utilizaban para procesamiento de texto, hojas de cálculo y muchas otras aplicaciones altamente interactivas que las computadoras grandes no podían manejar bien.

Las primeras computadoras personales solían venderse como "kits". Cada kit contenía una tarjeta de circuitos impresos, un puñado de chips, que típicamente incluían un Intel 8080, varios cables, una fuente de potencia y tal vez una unidad de disco flexible de 8 pulgadas. Al comprador correspondía armar las piezas para crear una computadora. No se proporcionaba software. Si usted quería software, lo escribía. Más adelante se popularizó el sistema operativo CP/M para el 8080, escrito por Gary Kildall. Era un verdadero sistema operativo para disco (flexible), con un sistema de archivos y comandos de usuario que se introducían con el teclado.

Otra de las primeras computadoras personales fue la Apple y posteriormente la Apple II, diseñadas por Steve Jobs y Steve Wozniak en la proverbial cochera. Esta máquina se popularizó mucho entre los usuarios caseros y en las escuelas y convirtió a Apple de la noche a la mañana en protagonista.

Después de muchas deliberaciones y de observar lo que otras compañías estaban haciendo, IBM, que entonces era la fuerza dominante en la industria de la computación, por fin decidió que quería ingresar en el negocio de las computadoras personales. En lugar de diseñar una máquina desde cero, empleando únicamente componentes de IBM, lo cual habría tardado mucho tiempo, IBM hizo algo totalmente fuera de carácter: entregó a uno de sus ejecutivos, Philip Estridge, una gran bolsa de dinero y le ordenó irse a algún lugar lejos de los entrometidos burócratas de las oficinas corporativas en Armonk, N.Y., y que no volviera

antes de tener una computadora personal funcional. Estridge se estableció lejos de la oficina central, en Boca Raton, Florida, escogió el Intel 8088 como su CPU, y construyó la IBM Personal Computer a partir de componentes comerciales. La máquina se introdujo en 1981 y se convirtió de inmediato en la computadora más vendida de la historia.

IBM hizo otra cosa fuera de carácter de la que más tarde se arrepentiría. En lugar de mantener totalmente en secreto el diseño de la máquina (o al menos protegido por una barrera de patentes), como normalmente hacía, publicó los planos completos, incluidos todos los diagramas de circuitos, en un libro que vendió a 49 dólares. Lo que se buscaba es que otras compañías fabricaran tarjetas que pudieran insertarse en la IBM PC (plug-ins) a fin de aumentar su flexibilidad y popularidad. Por desgracia para IBM, dado que el diseño ya era totalmente público y todas las piezas se podían conseguir fácilmente de proveedores comerciales con facilidad, muchas otras compañías comenzaron a fabricar **clones** de la PC, casi siempre a un costo mucho menor que lo que IBM cobraba. Fue así como nació toda una industria.

Aunque otras compañías fabricaron computadoras personales utilizando unas CPU que no eran de Intel, como las Commodore, Apple, Amiga y Atari, el ímpetu de la industria de la IBM PC era tan grande que las demás fueron arrolladas. Sólo unas cuantas sobrevivieron, aunque estuvieron restringidas a ciertos nichos del mercado, como estaciones de trabajo de ingeniería o supercomputadoras.

La versión inicial de la IBM PC venía equipada con el sistema operativo MS-DOS provisto por la entonces pequeña Microsoft Corporation. Cuando Intel logró producir CPU cada vez más potentes, IBM y Microsoft pudieron producir un sucesor del MS-DOS llamado OS/2, que contaba con una interfaz gráfica con el usuario similar a la de la Apple Macintosh. Mientras tanto, Microsoft creó también su propio sistema operativo Windows que se ejecutaba encima del MS-DOS, por si acaso el OS/2 no tenía aceptación. Sin abundar en la historia, el OS/2 no experimentó una buena acogida, IBM y Microsoft tuvieron una gran pelea extremadamente pública, y Microsoft se dedicó a hacer de Windows un enorme éxito. La forma en que la diminuta Intel y la todavía más diminuta Microsoft lograron destronar a IBM, una de las corporaciones más grandes, ricas y poderosas en la historia del mundo, es una parábola que sin duda se relata con lujo de detalle en escuelas de negocios de todo el mundo.

Para mediados de los ochenta una nueva idea llamada RISC comenzó a dominar, reemplazando arquitecturas complejas (CISC) por otras mucho más sencillas pero más rápidas. En los noventa comenzaron a aparecer las CPU superescalares. Estas máquinas podían ejecutar varias instrucciones al mismo tiempo, a menudo en un orden distinto del que tenían en el programa. Presentaremos los conceptos de CISC, RISC y superescalar en el capítulo 2 y los trataremos con detalle en todo el libro.

1.3 EL ZOOLÓGICO DE LAS COMPUTADORAS

En la sección anterior presentamos una muy breve historia de los sistemas de cómputo. En ésta examinaremos el presente y miraremos hacia el futuro. Aunque las computadoras personales son las computadoras más conocidas, hay otras clases de máquinas en uso actualmente, y vale la pena dar un vistazo a esas otras computadoras.

1.3.1 Fuerzas tecnológicas y económicas

La industria de la computación está avanzando como ninguna otra. La fue
mordial es la capacidad de los fabricantes de chips para empacar cada vez n
un chip. Un mayor número de transistores, que son diminutos interrupto
implica memorias más grandes y procesadores más potentes.

La rapidez del progreso tecnológico puede modelarse de acuerdo con una observación llamada **ley de Moore**, descubierta por Gordon Moore, cofundador y director de Intel, en 1965. Mientras preparaba un discurso para un grupo de la industria, Moore se dio cuenta de que cada nueva generación de chips de memoria se estaba introduciendo tres años después de la anterior. Puesto que cada nueva generación tenía cuatro veces más memoria que su predecesora, Moore se percató de que el número de transistores en un chip estaba aumentando de forma constante y predijo que este crecimiento continuaría durante varias décadas. Hoy día, la ley de Moore suele expresarse como que el número de transistores se duplica cada 18 meses. Cabe señalar que esto equivale a un incremento de cerca del 60% en el número de transistores cada año. Los tamaños de los chips de memoria y sus fechas de introducción, que se muestran en la figura 1-8, confirman que la ley de Moore se sigue cumpliendo.

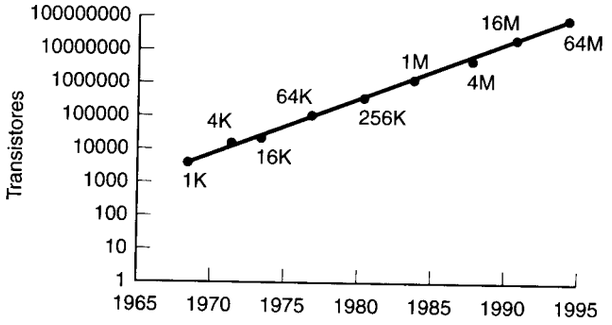


Figura 1-8. La ley de Moore predice un incremento anual de 60% en el número de transistores que se pueden colocar en un chip. Los puntos de datos dados en esta figura son tamaños de memoria, en bits.

Desde luego, la ley de Moore no es realmente una ley, sino sólo una observación empírica acerca de la rapidez con que los físicos de estado sólido y los ingenieros de proceso están empujando hacia adelante las fronteras tecnológicas, y una predicción de que seguirán haciéndolo al mismo ritmo en el futuro. Muchos observadores de la industria esperan que la ley de Moore se seguirá cumpliendo ya entrado el siglo XXI, posiblemente hasta 2020. En ese punto los transistores consistirán en un número demasiado reducido de átomos como para ser confiables, aunque los adelantos en la computación cuántica podrían hacer posible para entonces el almacenamiento de un bit utilizando el espín de un solo electrón.

La ley de Moore ha creado lo que algunos economistas llaman un **círculo virtuoso**. Los adelantos en la tecnología (transistores por chip) dan pie a mejores productos y precios más bajos. Los precios bajos dan pie a nuevas aplicaciones (nadie estaba creando juegos de video

antes de tener una computadora personal funcional. Estridge se estableció lejos de la oficina central, en Boca Raton, Florida, escogió el Intel 8088 como su CPU, y construyó la IBM Personal Computer a partir de componentes comerciales. La máquina se introdujo en 1981 y se convirtió de inmediato en la computadora más vendida de la historia.

IBM hizo otra cosa fuera de carácter de la que más tarde se arrepentiría. En lugar de mantener totalmente en secreto el diseño de la máquina (o al menos protegido por una barrera de patentes), como normalmente hacía, publicó los planos completos, incluidos todos los diagramas de circuitos, en un libro que vendió a 49 dólares. Lo que se buscaba es que otras compañías fabricaran tarjetas que pudieran insertarse en la IBM PC (plug-ins) a fin de aumentar su flexibilidad y popularidad. Por desgracia para IBM, dado que el diseño ya era totalmente público y todas las piezas se podían conseguir fácilmente de proveedores comerciales con facilidad, muchas otras compañías comenzaron a fabricar **clones** de la PC, casi siempre a un costo mucho menor que lo que IBM cobraba. Fue así como nació toda una industria.

Aunque otras compañías fabricaron computadoras personales utilizando unas CPU que no eran de Intel, como las Commodore, Apple, Amiga y Atari, el ímpetu de la industria de la IBM PC era tan grande que las demás fueron arrolladas. Sólo unas cuantas sobrevivieron, aunque estuvieron restringidas a ciertos nichos del mercado, como estaciones de trabajo de ingeniería o supercomputadoras.

La versión inicial de la IBM PC venía equipada con el sistema operativo MS-DOS provisto por la entonces pequeña Microsoft Corporation. Cuando Intel logró producir CPU cada vez más potentes, IBM y Microsoft pudieron producir un sucesor del MS-DOS llamado OS/2, que contaba con una interfaz gráfica con el usuario similar a la de la Apple Macintosh. Mientras tanto, Microsoft creó también su propio sistema operativo Windows que se ejecutaba encima del MS-DOS, por si acaso el OS/2 no tenía aceptación. Sin abundar en la historia, el OS/2 no experimentó una buena acogida, IBM y Microsoft tuvieron una gran pelea extremadamente pública, y Microsoft se dedicó a hacer de Windows un enorme éxito. La forma en que la diminuta Intel y la todavía más diminuta Microsoft lograron destronar a IBM, una de las corporaciones más grandes, ricas y poderosas en la historia del mundo, es una parábola que sin duda se relata con lujo de detalle en escuelas de negocios de todo el mundo.

Para mediados de los ochenta una nueva idea llamada RISC comenzó a dominar, reemplazando arquitecturas complejas (CISC) por otras mucho más sencillas pero más rápidas. En los noventa comenzaron a aparecer las CPU superescalares. Estas máquinas podían ejecutar varias instrucciones al mismo tiempo, a menudo en un orden distinto del que tenían en el programa. Presentaremos los conceptos de CISC, RISC y superescalar en el capítulo 2 y los trataremos con detalle en todo el libro.

1.3 EL ZOOLÓGICO DE LAS COMPUTADORAS

En la sección anterior presentamos una muy breve historia de los sistemas de cómputo. En ésta examinaremos el presente y miraremos hacia el futuro. Aunque las computadoras personales son las computadoras más conocidas, hay otras clases de máquinas en uso actualmente, y vale la pena dar un vistazo a esas otras computadoras.

1.3.1 Fuerzas tecnológicas y económicas

La industria de la computación está avanzando como ninguna otra. La fuerza impulsora primordial es la capacidad de los fabricantes de chips para empacar cada vez más transistores en un chip. Un mayor número de transistores, que son diminutos interruptores electrónicos, implica memorias más grandes y procesadores más potentes.

La rapidez del progreso tecnológico puede modelarse de acuerdo con una observación llamada **ley de Moore**, descubierta por Gordon Moore, cofundador y director de Intel, en 1965. Mientras preparaba un discurso para un grupo de la industria, Moore se dio cuenta de que cada nueva generación de chips de memoria se estaba introduciendo tres años después de la anterior. Puesto que cada nueva generación tenía cuatro veces más memoria que su predecesora, Moore se percató de que el número de transistores en un chip estaba aumentando de forma constante y predijo que este crecimiento continuaría durante varias décadas. Hoy día, la ley de Moore suele expresarse como que el número de transistores se duplica cada 18 meses. Cabe señalar que esto equivale a un incremento de cerca del 60% en el número de transistores cada año. Los tamaños de los chips de memoria y sus fechas de introducción, que se muestran en la figura 1-8, confirman que la ley de Moore se sigue cumpliendo.

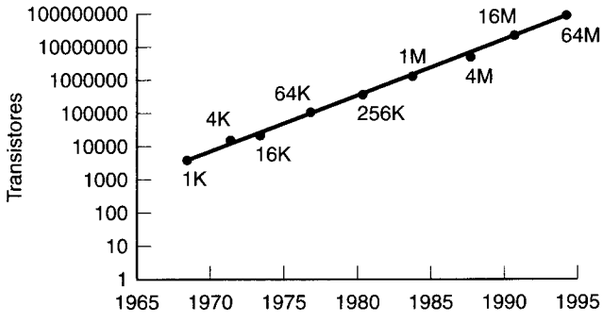


Figura 1-8. La ley de Moore predice un incremento anual de 60% en el número de transistores que se pueden colocar en un chip. Los puntos de datos dados en esta figura son tamaños de memoria, en bits.

Desde luego, la ley de Moore no es realmente una ley, sino sólo una observación empírica acerca de la rapidez con que los físicos de estado sólido y los ingenieros de proceso están empujando hacia adelante las fronteras tecnológicas, y una predicción de que seguirán haciéndolo al mismo ritmo en el futuro. Muchos observadores de la industria esperan que la ley de Moore se seguirá cumpliendo ya entrado el siglo XXI, posiblemente hasta 2020. En ese punto los transistores consistirán en un número demasiado reducido de átomos como para ser confiables, aunque los adelantos en la computación cuántica podrían hacer posible para entonces el almacenamiento de un bit utilizando el espín de un solo electrón.

La ley de Moore ha creado lo que algunos economistas llaman un **círculo virtuoso**. Los adelantos en la tecnología (transistores por chip) dan pie a mejores productos y precios más bajos. Los precios bajos dan pie a nuevas aplicaciones (nadie estaba creando juegos de video

para computadoras cuando éstas costaban 10 millones de dólares cada una). Las nuevas aplicaciones dan pie a nuevos mercados y nuevas compañías que surgen para aprovecharlos. La existencia de todas estas compañías da pie a una competencia que, a su vez, crea una demanda económica de mejores tecnologías con las cuales superar a los demás. Esto completa el círculo.

Otro factor que impulsa el mejoramiento tecnológico es la primera ley del software de Nathan (debida a Nathan Myhrvold, un alto ejecutivo de Microsoft) que dice: "El software es un gas; se expande hasta llenar el recipiente que lo contiene." En los años ochenta el procesamiento de textos se efectuaba con programas como Troff (que se utilizó para este libro). Troff ocupa unas cuantas decenas de kilobytes de memoria. Los modernos procesadores de textos ocupan decenas de megabytes. Los programas futuros sin duda requerirán decenas de gigabytes. El software que continuamente añade funciones (como los cascos de los barcos que sin cesar adquieren bálanos) crea una demanda constante de procesadores más rápidos, memorias más grandes y mayor capacidad de entrada/salida.

Si bien el aumento en el número de transistores por chip con el paso de los años ha sido impresionante, los avances en otras tecnologías de cómputo no se han quedado muy atrás. Por ejemplo, en 1982 se introdujo la IBM PC/XT con un disco duro de 10 MB. Por regla general, los sistemas de escritorio actuales tienen discos un tanto mayores. Medir el mejoramiento de los discos no es tan sencillo, porque implica varios parámetros (capacidad, tasa de datos, precio, etc.), pero casi cualquier métrica indicará que las capacidades han aumentado desde entonces a razón de 50% anual, por lo menos.

Otras áreas que han sido testigo de avances espectaculares son las telecomunicaciones y las redes. En menos de dos décadas hemos pasado de módems de 300 bits/s a módems analógicos de 56 kbps, líneas telefónicas ISDN de 2×64 kbps y redes de fibra óptica de más de un gigabit/s. Los cables telefónicos trasatlánticos de fibras ópticas, como TAT-12/13, cuestan unos 700 millones de dólares, duran 10 años y pueden transportar 300,000 llamadas simultáneas, lo que da un costo de menos de un centavo de dólar por una llamada intercontinental de 10 minutos. En el laboratorio se ha demostrado la factibilidad de sistemas de comunicación ópticos que operan a 1 terabit/s (10^{12} bits/s) a distancias de más de 100 km sin amplificadores. El crecimiento exponencial de Internet es tan evidente que no hace falta mencionarlo aquí.

1.3.2 La gama de las computadoras

Richard Hamming, antiguo investigador de Bell Labs, cierta vez observó que un cambio de un orden de magnitud en la cantidad causa un cambio en la calidad. Así, un automóvil que puede correr a 1000 km/h en el desierto de Nevada tiene un tipo de máquina fundamentalmente distinto de la de un automóvil normal que viaja a 100 km/h en una autopista. Así mismo, un rascacielos de 100 pisos no es sólo un edificio de apartamentos de 10 pisos a mayor escala. Y con las computadoras no estamos hablando de factores de 10, sino de factores de un millón en el curso de tres décadas.

Los aumentos predichos por la ley de Moore se pueden utilizar de varias maneras. Una de ellas es construir computadoras cada vez más potentes que no cuesten más. Otro enfoque es construir la misma computadora con menos y menos dinero cada año. La industria de la computación ha hecho ambas cosas y más, y el resultado es la amplia variedad de computadoras

que se venden actualmente. En la figura 1-9 se presenta una clasificación muy burda de las computadoras actuales.

Tipo	Precio	Ejemplo de aplicación
Computadora desechable	1	Tarjetas de felicitación
Computadora incorporada	10	Relojes, automóviles, aparatos
Computadora de juegos	100	Juegos de video caseros
Computadora personal	1 K	Computadora de escritorio o portátil
Servidor	10 K	Servidor de red
Colección de estaciones de trabajo	100 K	Minisupercomputadora
Mainframe (macrocomputadora)	1 M	Procesamiento de datos por lotes en un banco
Supercomputadora	10 M	Predicción del tiempo a largo plazo

Figura 1-9. Gama de computadoras disponibles en la actualidad. Los precios no deben tomarse muy en serio.

En el extremo inferior encontramos chips individuales pegados al interior de tarjetas de felicitación para tocar "Feliz cumpleaños", la "Marcha nupcial" o alguna otra tonadilla igualmente abominable. El autor todavía no ha encontrado una tarjeta de condolencias que toque una marcha fúnebre, pero después de haber sacado esta idea al dominio público, se cree que no tardará en aparecer. Para cualquier persona que haya crecido con mainframes de millones de dólares, la idea de una computadora desechable es tan lógica como la de un avión desechable. No obstante, es indudable que las computadoras desechables llegaron para quedarse. (¿Qué tal bolsas para la basura parlantes que nos recuerden reciclar las latas de aluminio?)

En el siguiente peldaño tenemos las computadoras que están incorporadas en teléfonos, televisores, hornos de microondas, reproductores de CD, juguetes, muñecas y miles de aparatos más. En unos cuantos años, todo artículo eléctrico incluirá una computadora. El número de computadoras incorporadas se medirá en miles de millones, rebasando por órdenes de magnitud a todas las demás computadoras combinadas. Estas computadoras contienen un procesador, menos de un megabyte de memoria, y cierta capacidad de E/S, todo en un solo chip que cuesta unos cuantos dólares.

En el siguiente peldaño están las máquinas de videojuegos. Se trata de computadoras normales, con capacidades gráficas especiales, pero software limitado y casi ninguna extensibilidad. En este intervalo de precios también están los organizadores personales, los asistentes digitales portátiles y otros dispositivos de computación de bolsillo similares, así como las computadoras de red y las terminales de Web. Lo que estas computadoras tienen en común es que contienen un procesador, unos cuantos megabytes de memoria, algún tipo de pantalla (posiblemente un televisor) y poco más. Eso es lo que los hace tan baratos.

A continuación vienen las computadoras personales en las que la mayoría de la gente piensa cuando oye la palabra "computadora". Éstas incluyen modelos de escritorio y portátiles. Por lo regular incluyen varios megabytes de memoria, un disco duro con unos cuantos gigabytes

de datos, una unidad de CD-ROM, un módem, una tarjeta de sonido y otros periféricos. Estas máquinas cuentan con sistemas operativos complejos, muchas opciones de expansión y un enorme surtido de software comercial. Las que tienen una CPU Intel suelen llamarse "computadoras personales", mientras que las que tienen otro tipo de CPU a menudo se llaman "estaciones de trabajo", aunque en lo conceptual casi no hay diferencias entre los dos tipos.

Ciertas computadoras personales o estaciones de trabajo mejoradas suelen utilizarse como servidores de red, tanto para redes de área local (casi siempre dentro de una sola compañía) como para Internet. Éstas tienen configuraciones de un solo procesador o de múltiples procesadores, tienen unos cuantos gigabytes de memoria, muchos gigabytes de espacio en disco duro, y capacidad de trabajo en red de alta velocidad. Algunas de ellas pueden manejar docenas o cientos de llamadas a la vez.

Más allá de los servidores multiprocesador pequeños encontramos sistemas conocidos como sistemas **NOW (redes de estaciones de trabajo**, *Networks of Workstations*) o **COW (cúmulos de estaciones de trabajo**, *Clusters of Workstations*). Éstos son computadores personales o estaciones de trabajo estándar conectadas por redes de gigabits/s y que ejecutan programas especiales que permiten a todas las máquinas colaborar en la solución de un solo problema, que a menudo es científico o de ingeniería. Es fácil cambiar la escala de estos sistemas, desde un puñado de máquinas hasta miles de ellas. Gracias al bajo precio de sus componentes, los departamentos independientes pueden poseer tales máquinas, que de hecho son minisupercomputadoras.

Ahora llegamos a las mainframes, computadoras que llenan una habitación y que se remontan a los sesenta. En muchos casos, estas máquinas son descendientes directos de mainframes IBM 360 adquiridas décadas atrás. En su mayor parte, estas computadoras no son mucho más rápidas que servidores potentes, pero suelen tener mayor capacidad de E/S y están equipadas con grandes "granjas" de discos que llegan a contener un terabyte de datos o más (1 TB = 10^{12} bytes). Aunque son extremadamente costosas, estas máquinas a menudo se mantienen funcionando en vista de la enorme inversión en software, datos, procedimientos operativos y personal que representan. Muchas compañías consideran que es más económico pagar unos cuantos millones de dólares de vez en cuando por una nueva, que siquiera contemplar el esfuerzo necesario para reprogramar todas sus aplicaciones para máquinas más pequeñas.

Es esta clase de computadoras la que dio pie al problema del Año 2000 de triste fama, el cual es culpa de los programadores en COBOL de los sesenta y setenta que representaban el año como dos dígitos decimales (para ahorrar memoria). Ellos nunca imaginaron que su software duraría tres o cuatro décadas. Muchas compañías han repetido el mismo error al limitarse a agregar dos dígitos más al año. El autor aprovecha la ocasión para predecir el fin de la civilización que conocemos a la media noche del 31 de diciembre de 9999, cuando 8000 años de viejos programas en COBOL dejarán de funcionar simultáneamente.

Después de las mainframes vienen las verdaderas supercomputadoras, cuyas CPU son inconcebiblemente rápidas, tienen muchos gigabytes de memoria principal, y sus discos y sus redes muy rápidos. En años recientes, muchas de las supercomputadoras se han convertido en máquinas altamente paralelas, no muy diferentes de los diseños COW, pero con componentes más rápidos y numerosos. Las supercomputadoras se emplean para resolver problemas que requieren muchos cálculos en ciencia e ingeniería, como simular el choque de galaxias, sintetizar nuevas medicinas o modelar el flujo del aire alrededor de un ala de avión.

1.4 EJEMPLOS DE FAMILIAS DE COMPUTADORAS

En esta sección presentaremos una breve introducción a las tres computadoras que usaremos como ejemplos en el resto del libro: la Pentium II, la UltraSPARC II y la picoJava II [sic].

1.4.1 Introducción al Pentium II

En 1968 Robert Noyce, inventor del circuito integrado de silicio; Gordon Moore, cuya ley ya es famosa, y Arthur Rock, un inversionista de San Francisco, formaron la Intel Corporation para fabricar chips de memoria. En su primer año de operación, Intel sólo vendió 3000 dólares de chips, pero el negocio ha prosperado desde entonces.

A fines de los años sesenta, las calculadoras eran grandes máquinas electromecánicas del tamaño de una moderna impresora láser y con un peso de unos 20 kg. En septiembre de 1969 una compañía japonesa, Busicom, se acercó a Intel para pedirle que fabricara 12 chips a la medida para una calculadora electrónica propuesta. El ingeniero de Intel asignado a este proyecto, Ted Hoff, examinó el plan y se percató de que podía poner una CPU de 4 bits de propósito general en un solo chip, y que esto efectuaría el mismo trabajo y sería además más simple y económico. Fue así como en 1970 nació la primera CPU en un solo chip, la 4004 de 2300 transistores (Faggin *et al.*, 1996).

Vale la pena señalar que ni Intel ni Busicom tenían la menor idea de lo que acababan de hacer. Cuando Intel decidió que valdría la pena hacer el intento de usar la 4004 en otros proyectos, ofreció comprar a Busicom los derechos del nuevo chip devolviéndole los 60,000 dólares que ésta había pagado a Intel por desarrollarlo. Busicom aceptó rápidamente el ofrecimiento de Intel, y éste comenzó a trabajar en una versión de ocho bits del chip, el 8008, introducido en 1972.

Intel no esperaba mucha demanda por el 8008, así que estableció una línea de producción de bajo volumen. Para asombro de todos, el chip despertó un interés enorme, por lo que Intel se dedicó a diseñar un nuevo chip de CPU que superara la limitación de 16K de memoria del 8008 (impuesta por el número de agujas en el chip). Este diseño dio como resultado el 8080, una CPU pequeña, de propósito general, introducido en 1974. Como había hecho la PDP-8, este producto tomó a la industria por asalto y de la noche a la mañana se convirtió en un artículo de mercado masivo. Sólo que en lugar de vender millares, como había hecho DEC, Intel vendió millones.

En 1978 llegó el 8086, una verdadera CPU de 16 bits en un solo chip. El 8086 tenía un diseño similar al del 8080, pero no era totalmente compatible con éste. Al 8086 siguió el 8088, que tenía la misma arquitectura que el 8086 y ejecutaba los mismos programas, pero tenía un bus de 8 bits en lugar de uno de 16 bits, lo que lo hacía más lento pero más económico que el 8086. Cuando IBM escogió el 8088 como CPU para la IBM PC original, este chip rápidamente se convirtió en el estándar para la industria de las computadoras personales.

Ni el 8088 ni el 8086 podían direccionar más de un megabyte de memoria. Para principios de los ochenta esto se convirtió en un problema cada vez más grave, por lo que Intel diseñó el 80286, una versión del 8086 compatible hacia arriba. El conjunto de instruc-

ciones básico era casi el mismo que el del 8086 y el 8088, pero la organización de la memoria era muy diferente y un tanto torpe, debido al requisito de compatibilidad con los chips anteriores. El 80286 se usó en la IBM PC/AT y en los modelos PS/2 de mediano precio. Al igual que el 8088, este chip tuvo un éxito enorme, principalmente porque la gente lo veía como un 8088 más rápido.

El siguiente paso lógico era una verdadera CPU de 32 bits en un chip, el 80386, que salió al mercado en 1985. Al igual que el 80286, éste era más o menos compatible con todos los chips anteriores, hasta el 8080. Ser compatible con lo existente era una bendición para la gente que necesitaba ejecutar software viejo, pero un fastidio para quienes habrían preferido una arquitectura sencilla, limpia y moderna que no cargara con los errores y la tecnología del pasado.

Cuatro años después salió el 80486. Éste era básicamente una versión más rápida del 80386 que además tenía una unidad de punto flotante y 8K de memoria caché en el chip. La **memoria caché** sirve para retener las palabras de memoria más comúnmente usadas dentro o cerca de la CPU, para evitar los (lentos) accesos a la memoria principal. El 80386 también contaba con apoyo de multiprocesador integrado, que permitía a los fabricantes construir sistemas con múltiples CPU.

En este punto Intel se enteró por las malas (al perder una demanda legal de violación de marca comercial) que los números (como 80486) no pueden registrarse como marcas, así que la nueva generación recibió un nombre: Pentium (del vocablo griego penta, cinco). A diferencia del 80486, que tenía una fila de procesamiento, el Pentium tenía dos, y esto lo ayudaba a ser dos veces más rápido (trataremos las filas de procesamiento con detalle en el capítulo 2).

Cuando apareció la siguiente generación, la gente que esperaba el Sexium (*sex* es seis en latín) sufrió una decepción. El nombre Pentium ya era tan conocido que la gente de comercialización quería conservarlo, así que el nuevo chip recibió el nombre de Pentium Pro. A pesar de lo trivial del cambio en el nombre respecto al de su predecesor, este procesador representó una importante ruptura con el pasado. En lugar de tener dos o más filas de procesamiento, el Pentium Pro tenía una organización interna muy distinta y podía ejecutar hasta cinco instrucciones a la vez.

Otra innovación del Pentium Pro fue una memoria caché de dos niveles. El chip procesador mismo tenía 8 KB de memoria para retener las instrucciones de uso común, y 8 KB de memoria para retener datos de uso común. En la misma cavidad dentro del paquete Pentium Pro (pero no en el chip mismo) había una segunda memoria caché de 256 KB.

El siguiente procesador Intel fue el Pentium II, que era básicamente un Pentium Pro con la adición de extensiones multimedia especiales (llamadas **MMX**). Estas instrucciones estaban diseñadas para acelerar los cálculos requeridos para procesar audio y video, lo que hacía innecesaria la adición de coprocesadores de multimedia especiales. Estas instrucciones también estuvieron disponibles en procesadores Pentium posteriores, pero no en el Pentium Pro, de modo que el Pentium II combinaba las ventajas del Pentium Pro con multimedios.

A principios de 1998 Intel introdujo una nueva línea de productos llamada **Celeron**, que es básicamente una versión de bajo costo y bajo desempeño del Pentium II proyectada para PC del extremo inferior. Puesto que el Celeron tiene la misma arquitectura que el Pentium II, no hablaremos más de él en este libro. En junio de 1998 Intel introdujo una versión especial del Pentium II para el extremo superior del mercado. Este procesador, llamado **Xeon**, tiene

un caché más grande, un bus más rápido y mejor apoyo multiprocesador, pero por lo demás es un Pentium II normal, así que tampoco lo trataremos individualmente. La familia Intel se muestra en la figura 1-10.

Chip	Fecha	MHz	Transistores	Memoria	Notas
4004	4/1971	0.108	2,300	640	Primer microprocesador en un chip
8008	4/1972	0.108	3,500	16 KB	Primer microprocesador de 8 bits
8080	4/1974	2	6,000	64 KB	Primera CPU de propósito general en un bit
8086	6/1978	5-10	29,000	1 MB	Primera CPU de 16 bits en un chip
8088	6/1979	5-8	29,000	1 MB	Se usó en la IBM PC
80286	2/1982	8-12	134,000	16 BM	Cunta con protección de memoria
80386	10/1985	16-33	275,000	4 GB	Primer CPU de 32 bits
80486	4/1989	25-100	1.2M	4 GB	Memoria caché de 8K integrada
Pentium	3/1993	60-233	3.1M	4 GB	Dos conductos; modelos posteriores tenían MMX
Pentium Pro	3/1995	150-200	5.5M	4 GB	Dos niveles de caché integrados
Pentium II	5/1997	233-400	7.5M	4 GB	Pentium Pro más MMX

Figura 1-10. La familia de CPU Intel. Las velocidades de reloj se miden en MHz (megahertz), donde 1 MHz es 1 millón de ciclos/s.

Todos los chips Intel son compatibles con sus predecesores hasta el 8086. En otras palabras, un Pentium II puede ejecutar programas del 8086 sin modificación. Esta compatibilidad siempre ha sido un requisito de diseño para Intel, a fin de que los usuarios puedan mantener su inversión en software existente. Desde luego, el Pentium II es 250 veces más complejo que el 8086, así que puede hacer una buena cantidad de cosas que el 8086 nunca pudo hacer. Estas extensiones incrementales han dado pie a una arquitectura menos elegante de lo que podría haber sido si alguien hubiera entregado a los arquitectos del Pentium II 7.5 millones de transistores e instrucciones para comenzar otra vez desde el principio.

Resulta interesante que si bien la ley de Moore ha estado asociada desde hace mucho al número de bits de una memoria, aplica igualmente bien a los chips de CPU. Si graficamos los números de transistores dados en la figura 1-10 contra la fecha de introducción de cada chip en una escala semilogarítmica, vemos que la ley de Moore también se cumple. La gráfica correspondiente se presenta en la figura 1-11.

1.4.2 Introducción al UltraSPARC II

En los setenta UNIX era popular en las universidades, pero ninguna computadora personal lo ejecutaba, así que los amantes de UNIX tenían que usar minicomputadoras de tiempo compar-

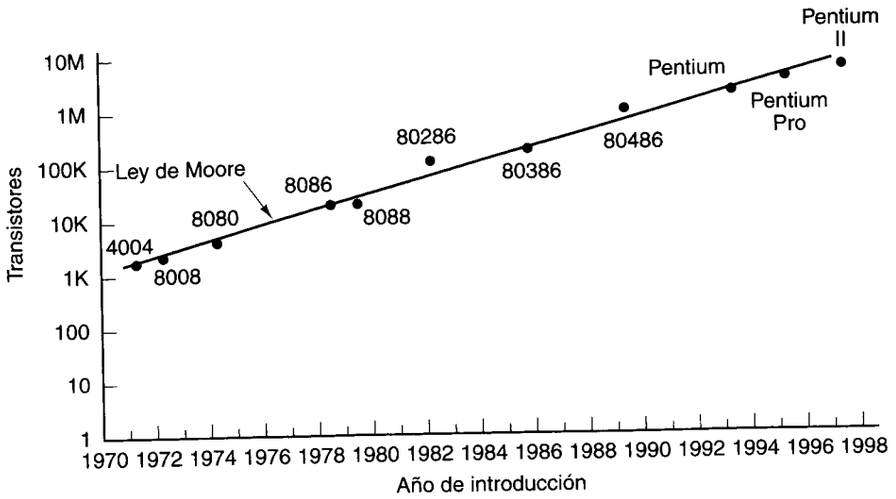


Figura 1-11. Ley de Moore para chips de CPU.

tido (a menudo sobrecargadas) como la PDP-11 y VAX. En 1981 un estudiante de posgrado alemán en Stanford, Andy Bechtolsheim, quien se sentía frustrado por tener que acudir al centro de cómputo para usar UNIX, decidió resolver el problema construyendo para su uso personal una estación de trabajo UNIX utilizando componentes comerciales, y la llamó SUN-1 (Stanford University Network).

Bechtolsheim pronto atrajo la atención de Vinod Khosla, un hindú de 27 años que tenía el deseo ferviente de retirarse como millonario antes de cumplir los 30 años. Khosla convenció a Bechtolsheim de formar una compañía para construir y vender estaciones de trabajo Sun. Luego, Khosla contrató a Scott McNealy, otro estudiante de posgrado de Stanford, para encargarse del área de producción. Para escribir el software contrataron a Bill Joy, el principal arquitecto del Berkeley UNIX. Los cuatro fundaron Sun Microsystems en 1982.

El primer producto de Sun, la Sun-1, basada en la CPU Motorola 68020, tuvo un éxito inmediato, lo mismo que las máquinas subsecuentes Sun-2 y Sun-3, que también usaban CPU Motorola. A diferencia de otras computadoras personales de la época, estas máquinas eran mucho más potentes (de ahí la designación "estación de trabajo") y estaban diseñadas desde un principio para operar en una red. Cada estación de trabajo Sun venía equipada con una conexión Ethernet y con software TCP/IP para conectarse con ARPANET, la precursora de Internet.

Para 1987 Sun, que ya estaba vendiendo 500 millones de dólares al año en sistemas, decidió diseñar su propia CPU basándola en un nuevo y revolucionario diseño de la University of California at Berkeley (el RISC II). Esta CPU, llamada **SPARC** (**arquitectura de procesador escalable**, *Scalable Processor ARChitecture*), fue la base para la estación de trabajo Sun-4. En poco tiempo, todos los productos de Sun utilizaban la CPU SPARC.

A diferencia de muchas otras compañías de computadoras, Sun decidió no fabricar el chip de CPU SPARC ella misma. En vez de ello, otorgó licencias a varios fabricantes de semiconductores para producirlo, esperando que la competencia entre ellos hiciera subir el

desempeño y bajar los precios. Los proveedores fabricaron varios chips distintos, basados en diferentes tecnologías, que operaban a diferente velocidad de reloj, y con distintos precios. Estos chips incluyeron MicroSPARC, HyperSPARC, SuperSPARC y TurboSPARC. Aunque estas CPU diferían en aspectos menores, todas eran compatibles en lo binario y ejecutaban los mismos programas de usuario sin modificación.

Sun siempre quiso que SPARC fuera una arquitectura abierta, con muchos proveedores de componentes y sistemas, a fin de forjar una industria capaz de competir en un mundo de computadoras personales ya dominado por las CPU basadas en Intel. Para ganarse la confianza de compañías que estaban interesadas en SPARC pero no querían invertir en un producto controlado por un competidor, Sun creó un consorcio de la industria, SPARC International, para controlar el desarrollo de versiones futuras de la arquitectura SPARC. Por ello es importante distinguir entre la arquitectura SPARC, que es una especificación del conjunto de instrucciones y otras características visibles para el programador, y una implementación específica de ella. En este libro estudiaremos la arquitectura SPARC genérica y, al hablar de los chips de CPU en los capítulos 3 y 4, de un chip SPARC específico empleado en estaciones de trabajo Sun.

El SPARC inicial era una máquina completa de 32 bits que operaba a 36 MHz. La CPU, llamada **IU** (**unidad de enteros**, *Integer Unit*), era magra y feroz, con sólo tres formatos de instrucción principales y sólo 55 instrucciones en total. Además, una unidad de punto flotante aportaba otras 14 instrucciones. Esta historia puede contrastarse con la línea Intel, que comenzó con chips de 8 y 16 bits (8088, 8086, 80286) y por fin se convirtió en un chip de 32 bits con el 80386.

La primera ruptura de SPARC con el pasado ocurrió en 1995, con el desarrollo de la versión 9 de la arquitectura SPARC, una arquitectura cabal de 64 bits, con direcciones de 64 bits y registros de 64 bits. La primera estación de trabajo Sun que implementó la arquitectura V9 (versión 9) fue la **UltraSPARC I**, introducida en 1995 (Tremblay y O'Connor, 1996). A pesar de ser una máquina de 64 bits, era totalmente compatible en lo binario con los SPARC de 32 bits existentes.

El propósito de UltraSPARC era explorar nuevos terrenos. Mientras que máquinas anteriores estaban diseñadas para manejar datos alfanuméricos, ejecutar programas como procesadores de textos y hojas de cálculo, la UltraSPARC se diseñó desde el principio para manejar imágenes, audio, video y multimedia en general. Entre otras innovaciones además de la arquitectura de 64 bits había 23 instrucciones nuevas, incluidas algunas para empacar y desempacar pixeles de palabras de 64 bits, cambiar la escala de imágenes y girarlas, transferir bloques y realizar compresión y descompresión de video en tiempo real. Estas instrucciones, llamadas **VIS** (**conjunto de instrucciones visuales**, *Visual Instruction Set*) tenían por objeto ofrecer una capacidad multimedia general, análoga a las instrucciones MMX de Intel.

El UltraSPARC estaba dirigido a aplicaciones del extremo superior, como servidores de Web multiprocesador grandes, con docenas de CPU y memorias físicas de hasta 2 TB [1 TB (terabyte) = 10^{12} bytes]. Sin embargo, versiones más pequeñas pueden usarse también en computadoras portátiles.

Los sucesores del UltraSPARC I fueron UltraSPARC II y UltraSPARC III. Estos modelos difieren primordialmente en cuanto a velocidad de reloj, pero también se añadieron algu-

nas funciones nuevas en cada iteración. En este libro, cuando tratemos la arquitectura SPARC, utilizaremos el UltraSPARC II V9 de 64 bits como ejemplo.

1.4.3 Introducción al picoJava II

El lenguaje de programación C fue inventado por Dennis Ritchie de Bell Labs para usarlo con el sistema operativo UNIX. Gracias a su diseño económico y la popularidad de UNIX, C pronto se convirtió en el lenguaje de programación dominante en el mundo para programación de sistemas. Algunos años después, Bjarne Stroustrup, también de Bell Labs, añadió a C ideas del mundo de la programación orientada a objetos para crear C++, que también se volvió muy popular.

A mediados de los noventa, investigadores de Sun Microsystems estaban explorando formas de lograr que los usuarios obtuvieran programas binarios por Internet y los ejecutaran como parte de páginas de la World Wide Web. A ellos les gustaba C++, excepto que no era seguro. En otras palabras, un programa binario en C++ recién bajado de Internet fácilmente podía espiar la máquina que lo había adquirido e interferirla de otras maneras. Su solución a este problema fue inventar un nuevo lenguaje de programación, Java, inspirado en C++ pero sin los problemas de seguridad de éste. Java es un lenguaje orientado a objetos seguro, que cada vez se usa más para muchas aplicaciones. Por ser un lenguaje popular y elegante, lo usaremos en los ejemplos de programación de este libro.

Puesto que Java no es más que un lenguaje de programación, es posible escribir compiladores de él para la arquitectura Pentium, SPARC o cualquier otra. Existen tales compiladores, pero el principal objetivo de Sun al introducir Java era poder intercambiar programas ejecutables Java entre computadoras de Internet y ejecutarlos sin modificación. Si un programa Java compilado en una máquina SPARC se enviara por Internet a una Pentium, no se ejecutaría, y no se alcanzaría la meta de poder enviar un programa binario a cualquier sitio y ejecutarlo ahí.

Para poder transportar programas binarios entre máquinas distintas, Sun definió una arquitectura de máquina virtual llamada **JVM (Java Virtual Machine)**. Esta máquina tiene una memoria formada por palabras de 32 bits, y puede ejecutar 226 instrucciones. Casi todas estas instrucciones son sencillas, pero unas cuantas son muy complejas y requieren varios ciclos de memoria.

A fin de hacer portátil a Java, Sun escribió un compilador de Java para la JVM, y también escribió un intérprete JVM para ejecutar programas binarios Java. Éste intérprete se escribió en C y por tanto puede compilarse y ejecutarse en cualquier máquina que posea un compilador de C, lo que en la práctica incluye a casi todas las máquinas del mundo. Por consiguiente, lo único que el dueño de una máquina necesita hacer para que ésta pueda ejecutar programas binarios Java es conseguir el programa binario ejecutable del intérprete de Java para esa plataforma (digamos, Pentium II y Windows 98, SPARC y UNIX, etc.) junto con ciertos programas y bibliotecas de apoyo asociados. Además, casi todos los navegadores de Internet contienen un intérprete de JVM para poder ejecutar fácilmente **applets**, que son pequeños programas binarios Java asociados a páginas de la World Wide Web. Muchos de estos applets proporcionan animación y sonido.

La interpretación de programas para JVM (o de cualesquier otros programas, para el caso) es lenta. Una estrategia alternativa a ejecutar un applet u otro programa JVM recién recibido es compilarlo primero para la máquina en cuestión y luego ejecutar el programa compilado. Esta estrategia requiere tener un compilador de JVM a lenguaje de máquina dentro del navegador y poder activarlo automáticamente cuando se necesite. Tales compiladores, llamados compiladores **JIT (justo a tiempo**, *Just In Time*), existen y son comunes, pero son grandes e introducen un retraso entre la llegada del programa JVM y su ejecución mientras el programa se compila a lenguaje de máquina.

Además de las implementaciones en software de la máquina virtual Java (intérpretes de JVM y compiladores JIT), Sun y otras compañías han diseñado chips de JVM. Éstos son diseños de CPU que ejecutan directamente programas binarios JVM, sin necesidad de una capa de interpretación por software o compilación JIT. Las arquitecturas iniciales, picoJava I (O'Connor y Tremblay, 1997) y picoJava II (McGhan y O'Connor, 1998), estaban pensadas para el mercado de los sistemas incorporados. Este mercado requiere chips potentes, flexibles y sobre todo de bajo costo (menos de 50 dólares, y a veces mucho menos) que se incorporan en tarjetas inteligentes, televisores, teléfonos y otros aparatos, sobre todo los que necesitan comunicarse con el mundo exterior. Los licenciatarios de Sun pueden fabricar sus propios chips utilizando el diseño picoJava, adaptándolos hasta cierto punto incluyendo o quitando la unidad de punto flotante, ajustando el tamaño de los cachés, etcétera.

El valor de un chip Java para el mercado de los sistemas incorporados es que un dispositivo puede alterar su funcionalidad mientras está operando. Por ejemplo, considere un ejecutivo de negocios que tiene un teléfono celular basado en Java y que nunca vislumbró la necesidad de leer faxes en la diminuta pantalla del teléfono, pero que repentinamente necesita hacerlo. Si llama a su proveedor celular, el ejecutivo puede bajar a su teléfono un applet para visualización de faxes y añadir esta funcionalidad al dispositivo. Las necesidades de velocidad excluyen la posibilidad de interpretar los applets Java, y la falta de memoria en el teléfono impide la compilación JIT. Ésta es una situación en la que un chip JVM es útil.

Aunque el picoJava II no es un chip concreto (no puede ir a una tienda y comprar uno), es la base de varios chips, como la CPU microJava 701 de Sun y muchos más de otros licenciatarios de Sun. Utilizaremos el diseño picoJava II como uno de nuestros ejemplos constantes en todo el libro porque es muy diferente de las CPU Pentium II y UltraSPARC, y está dirigido a un área de aplicación muy diferente. Esta CPU resulta especialmente interesante para nuestros propósitos porque en el capítulo 4 presentaremos un diseño para implementar un subconjunto de JVM empleando microprogramación. Luego podremos contrastar nuestro diseño microprogramado con un verdadero diseño en hardware.

El picoJava II tiene dos unidades opcionales: un caché y una unidad de punto flotante, que cada fabricante de chips puede incluir o eliminar, a voluntad. Por sencillez, nos referiremos al picoJava II como si fuera un chip en lugar de un diseño de chip. Cuando sea importante (en contadas ocasiones), seremos específicos y hablaremos del chip microJava 701 que implementa el diseño picoJava II. Aun cuando no mencionemos el microJava 701 específicamente, el material aplicará a él y a todos los demás chips Java de otros fabricantes basados en este diseño.

Al utilizar Pentium II, UltraSPARC II y picoJava II como ejemplos, podremos estudiar tres tipos distintos de CPU. Éstos son, respectivamente, una arquitectura CISC tradicional implementada con tecnología superescalar moderna, una verdadera arquitectura RISC implementada con tecnología superescalar, y un chip Java dedicado para usarse en sistemas incorporados. Estos tres procesadores son muy diferentes entre sí, y esto nos da la oportunidad de explorar mejor el espacio de diseño y ver qué tipos de concesiones pueden hacerse para procesadores dirigidos a públicos distintos.

1.5 BOSQUEJO DEL LIBRO

Este libro trata las computadoras multinivel (término que abarca casi todas las computadoras modernas) y su organización. Examinaremos cuatro niveles con gran detalle: el nivel de lógica digital, el nivel de microarquitectura, el nivel de ISA y el nivel de sistema operativo. Entre las cuestiones básicas que examinaremos están el diseño general del nivel (y por qué se diseñó así), los tipos de instrucciones y datos con que se cuenta, la organización y direccionamiento de la memoria, y el método de implementación del nivel. El estudio de estos temas, y otros similares, se denomina organización de computadoras o arquitectura de computadoras.

Lo que más nos interesa son los conceptos, no los detalles ni las matemáticas formales. Por esa razón, algunos de los ejemplos estarán muy simplificados para hacer hincapié en las ideas centrales y no los detalles.

A fin de entender cómo se aplican en la práctica los principios presentados en este libro, utilizaremos el Pentium II, el UltraSPARC II y el picoJava II como ejemplos constantes en todo el libro. Se escogieron estos diseños por varias razones. Primera, todos se usan ampliamente y es probable que el lector tenga acceso a por lo menos uno de ellos. Segunda, cada uno tiene su propia arquitectura singular, lo que permite efectuar comparaciones y fomenta una actitud de "¿qué alternativas hay?". Los libros que se ocupan de una sola máquina a menudo dejan al lector con la sensación de que les ha sido revelada "la máquina verdadera", cosa que es absurda en vista de las múltiples concesiones y decisiones arbitrarias que los diseñadores se ven obligados a hacer. Recomendamos al lector estudiar éstas y todas las demás computadoras con un ojo crítico y tratar de entender por qué las cosas son como son y cómo podrían haber sido diferentes, en lugar de limitarse a aceptarlas tal cual.

Debe quedar claro desde el principio que éste no es un libro acerca de cómo programar el Pentium II, el UltraSPARC II o el picoJava II. Utilizaremos estas máquinas como ilustraciones en los puntos apropiados, pero no pretendemos presentar un tratamiento completo. Los lectores que deseen una introducción exhaustiva a cualquiera de ellas deberán consultar las publicaciones del fabricante.

El capítulo 2 es una introducción a los componentes básicos de una computadora: procesadores, memoria y equipo de entrada/salida. Su propósito es presentar un panorama general de la arquitectura de los sistemas y una introducción a los capítulos subsecuentes.

Los capítulos 3, 4, 5 y 6 se ocupan cada uno de un nivel específico de los que se muestran en la figura 1-2. Nuestro tratamiento es ascendente, porque así es como se han diseñado tradicionalmente las máquinas. El diseño del nivel k está determinado en gran medida por las propiedades del nivel $k - 1$, de modo que es difícil entender cualquier nivel si no se tiene ya una buena comprensión del nivel subyacente que lo motivó. Además, es recomendable desde el punto de vista educativo proceder de los niveles inferiores, más sencillos, hasta los niveles superiores, más complejos, y no al revés.

El capítulo 3 se ocupa del nivel de lógica digital, el verdadero hardware de la máquina. Se explica qué son las compuertas y cómo pueden combinarse en circuitos útiles. También se presenta el álgebra booleana, una herramienta para analizar circuitos digitales. Se describen los buses de computadora, sobre todo el bus PCI tan utilizado. El capítulo examina numerosos ejemplos de la industria, incluidos los tres ejemplos constantes antes mencionados.

El capítulo 4 introduce la organización del nivel de microarquitectura y su control. Puesto que la función de este nivel es interpretar las instrucciones de nivel 2 de la capa que está arriba, nos concentraremos en este tema y lo ilustraremos con ejemplos. El capítulo también contiene descripciones del nivel de microarquitectura de algunas máquinas reales.

El capítulo 5 trata el nivel de ISA, el que la mayoría de los fabricantes de computadoras anuncia como el lenguaje de máquina. Aquí examinaremos detalladamente nuestras máquinas de ejemplo.

El capítulo 6 cubre algunas de las instrucciones, organización de memoria y mecanismos de control presentes en el nivel de sistema operativo de la máquina. Los ejemplos que se usan aquí son Windows NT (popular en sistemas servidores Pentium II del extremo superior) y UNIX, empleado en el UltraSPARC II.

El capítulo 7 habla del nivel de lenguaje ensamblador. Se cubre tanto el lenguaje ensamblador como el proceso de ensamblado. Otro tema que surge aquí es el de enlazado.

El capítulo 8 se ocupa de las computadoras paralelas, un tema cada vez más importante en nuestros días. Algunas de estas computadoras paralelas tienen múltiples CPU que comparten una misma memoria. Otras tienen varias CPU sin memoria común. Algunas son supercomputadoras; otras son COW. Todo será cubierto en este capítulo.

El capítulo 9 contiene una lista de lecturas sugeridas con anotaciones, organizada por tema, y una lista alfabética de citas bibliográficas. Éste es el capítulo más importante del libro. Úselo.

PROBLEMAS

1. Explique los siguientes términos en sus propias palabras:
   a. Traductor.
   b. Intérprete.
   c. Máquina virtual.
2. ¿Qué diferencia hay entre interpretación y traducción?
3. ¿Es concebible que un compilador genere salidas para el nivel de microarquitectura en lugar del nivel ISA? Comente las ventajas y las desventajas de esta propuesta.

4. ¿Puede imaginar una computadora multinivel en la que el nivel de dispositivos y el de lógica digital no son los niveles más bajos? Explique.

5. Considere una computadora con intérpretes idénticos en los niveles 1, 2 y 3. Un intérprete requiere n instrucciones para ir a buscar, examinar y ejecutar una instrucción. Una instrucción del nivel 1 tarda k nanosegundos en ejecutarse. ¿Cuánto tarda una instrucción en los niveles 2, 3 y 4?

6. Considere una computadora multinivel en la que todos los niveles son diferentes. Cada nivel tiene instrucciones que son m veces más potentes que las del nivel inmediato inferior; es decir, una instrucción del nivel r puede efectuar el trabajo de m instrucciones del nivel $r - 1$. Si un programa del nivel 1 tarda k segundos en ejecutarse, ¿cuánto tardarían programas equivalentes en los niveles 2, 3 y 4, suponiendo que se requieren m instrucciones del nivel r para interpretar una sola instrucción del nivel $r+1$?

7. Algunas instrucciones en el nivel de sistema operativo son idénticas a instrucciones en lenguaje ISA. El microprograma ejecuta directamente esas instrucciones, en lugar de que lo haga el sistema operativo. A la luz de su respuesta al problema anterior, ¿por qué cree que esto sea así?

8. ¿En qué sentido son equivalentes el hardware y el software? ¿En qué sentido no son equivalentes?

9. Una de las consecuencias de la idea de von Neumann de almacenar el programa en la memoria es que los programas pueden modificarse, igual que los datos. ¿Se le ocurre algún ejemplo en el que esta posibilidad podría haber sido útil? (Sugerencia: piense en realizar operaciones aritméticas con arreglos.)

10. El coeficiente de desempeño de la 360 modelo 75 es de 50 veces el de la 360 modelo 30, pero el tiempo de ciclo es sólo cinco veces más corto. ¿Cómo explica esta discrepancia?

11. En las figuras 1-5 y 1-6 se muestran dos diseños básicos de sistemas. Describa cómo podría efectuarse entrada/salida en cada sistema. ¿Cuál podría tener un mejor desempeño global?

12. En cierto momento, un transistor de un microprocesador tenía un diámetro de 1 micra. Según la ley de Moore, ¿qué tamaño tendría un transistor en el modelo del año siguiente?

2

ORGANIZACIÓN DE LOS SISTEMAS DE COMPUTADORA

Una computadora digital consiste en un sistema de procesadores interconectados, memorias y dispositivos de entrada/salida. Este capítulo es una introducción a estos tres componentes y a su interconexión, como antecedentes para el examen detallado de niveles específicos en los cinco capítulos subsecuentes. Los conceptos de procesador, memoria y entrada/salida son fundamentales y estarán presentes en todos los niveles, por lo que iniciaremos nuestro estudio de la arquitectura de las computadoras examinándolos uno por uno.

2.1 PROCESADORES

La organización de una computadora sencilla orientada hacia los buses se muestra en la figura 2-1. La **CPU** (**unidad central de procesamiento**, *Central Processing Unit*) es el "cerebro" de la computadora. Su función es ejecutar programas almacenados en la memoria principal buscando sus instrucciones y examinándolas para después ejecutarlas una tras otra. Los componentes están conectados por un **bus**, que es una colección de alambres paralelos para transmitir direcciones, datos y señales de control. Los buses pueden ser externos a la CPU, cuando la conectan a la memoria y a los dispositivos de E/S, pero también internos, como veremos en breve.

La CPU se compone de varias partes. La unidad de control se encarga de buscar instrucciones de la memoria principal y determinar su tipo. La unidad de aritmética y lógica realiza operaciones como suma y AND booleano necesarias para ejecutar las instrucciones.

La CPU también contiene una memoria pequeña y de alta velocidad que sirve para almacenar resultados temporales y cierta información de control. Esta memoria se compone de

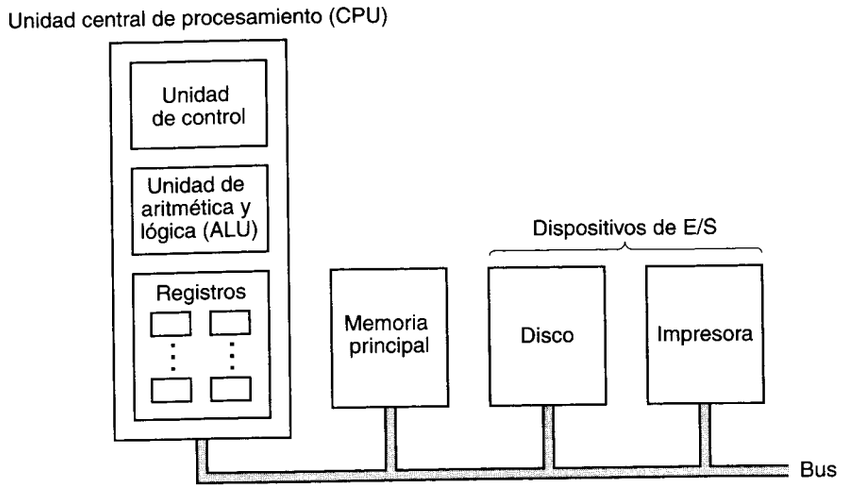


Figura 2-1. Organización de una computadora sencilla con una CPU y dos dispositivos de E/S.

varios registros, cada uno de los cuales tiene cierto tamaño y función. Por lo regular, todos los registros tienen el mismo tamaño. Cada registro puede contener un número, hasta algún máximo determinado por el tamaño del registro. Los registros pueden leerse y escribirse a alta velocidad porque están dentro de la CPU.

El registro más importante es el **contador de programa** (**PC**, *Program Counter*), que apunta a la siguiente instrucción que debe buscarse para ejecutarse. El nombre "contador de programa" es un tanto engañoso porque no tiene nada que ver con *contar*, pero es un término de uso universal. Otro registro importante es el **registro de instrucciones** (**IR**, *Instruction Register*), que contiene la instrucción que se está ejecutando. Casi todas las computadoras tienen varios registros más, algunos de propósito general y otros para fines específicos.

2.1.1 Organización de la CPU

En la figura 2-2 se muestra con más detalle la organización interna de una parte de una CPU von Neumann típica. Esta parte se llama **camino de datos** y consiste en los registros (generalmente del 1 al 32), la **ALU** (**unidad de aritmética y lógica**, *Arithmetic Logic Unit*) y varios buses que conectan los componentes. Los registros alimentan dos registros de entrada de la ALU, rotulados *A* y *B* en la figura. Estos registros contienen las entradas de la ALU mientras ésta está calculando. El camino de datos es muy importante en todas las máquinas y lo examinaremos con gran detalle a todo lo largo del libro.

La ALU suma, resta y realiza otras operaciones simples con sus entradas, y produce un resultado en el registro de salida. El contenido de este registro de salida se envía a un registro, que posteriormente se escribe (es decir, se guarda) en la memoria, si se

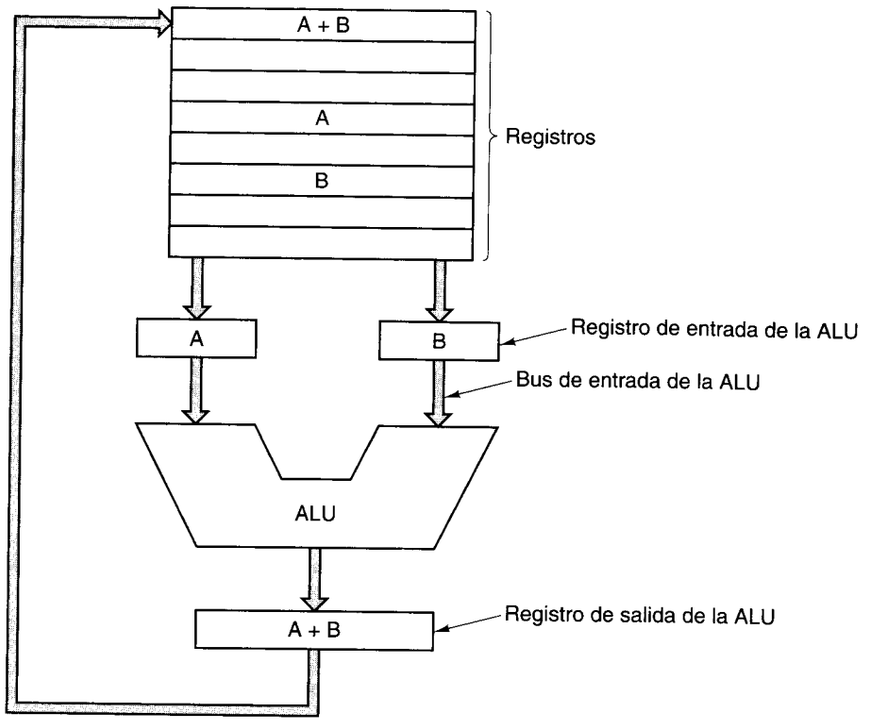


Figura 2-2. Camino de datos de una máquina von Neumann típica.

desea. No todos los diseños tienen los registros *A*, *B* y de salida. En el ejemplo se ilustra una suma.

Casi todas las instrucciones pueden dividirse en una de dos categorías: registro-memoria o registro-registro. Las instrucciones registro-memoria permiten buscar palabras de la memoria a los registros, donde pueden utilizarse como entradas de la ALU en instrucciones subsecuentes, por ejemplo. (Las "palabras" son las unidades de datos que se transfieren entre la memoria y los registros. Una palabra podría ser un entero. Trataremos la organización de la memoria más adelante en este capítulo.) Otras instrucciones registro-memoria permiten almacenar el contenido de un registro en la memoria.

La otra clase de instrucción es la de registro-registro. Una instrucción registro-registro típica busca dos operandos de los registros, los coloca en los registros de entrada de la ALU, realiza alguna operación con ellos (por ejemplo suma o AND booleano) y coloca el resultado en uno de los registros. El proceso de hacer pasar dos operandos por la ALU y almacenar el resultado se llama **ciclo del camino de datos** y es el corazón de casi todas las CPU. En gran medida, este ciclo define lo que la máquina puede hacer. Cuanto más rápido es el ciclo del camino de datos, más rápidamente opera la máquina.

2.1.2 EJECUCIÓN DE INSTRUCCIONES

La CPU ejecuta cada instrucción en una serie de pasos pequeños. A grandes rasgos, los pasos son los siguientes:

1. Buscar la siguiente instrucción de la memoria y colocarla en el registro de instrucciones.
2. Modificar el contador de programa de modo que apunte a la siguiente instrucción.
3. Determinar el tipo de la instrucción que se trajo.
4. Si la instrucción utiliza una palabra de la memoria, determinar dónde está.
5. Buscar la palabra, si es necesario, y colocarla en un registro de la CPU.
6. Ejecutar la instrucción.
7. Volver al paso 1 para comenzar a ejecutar la siguiente instrucción.

Esta sucesión de pasos se conoce como el ciclo de **búsqueda-decodificación-ejecución**, y es fundamental para el funcionamiento de todas las computadoras.

Esta descripción de cómo funciona una CPU se parece mucho a un programa escrito en español. La figura 2-3 muestra este programa informal reescrito como método Java (o sea, procedimiento) llamado *interpretar*. La máquina que se está interpretando tiene dos registros que los programas de usuario pueden ver: el contador de programa (PC), para seguir la pista a la dirección de la siguiente instrucción que debe buscarse, y el acumulador (AC), para acumular resultados aritméticos. También tiene registros internos para retener la instrucción en curso durante su ejecución (instr), el tipo de la instrucción en curso (instr_type), la dirección del operando de la instrucción (data_loc) y el operando mismo (data). Se supone que las instrucciones contienen una sola dirección de memoria. La posición de memoria direccionada contiene el operando; por ejemplo, el dato que debe sumarse al acumulador.

El simple hecho de que sea posible escribir un programa capaz de imitar la función de una CPU demuestra que un programa no tiene que ejecutarse con una CPU de "hardware" que consista en una caja llena de circuitos. En vez de ello, el programa puede ejecutarse haciendo que otro programa busque, examine y ejecute sus instrucciones. Un programa (como el de la figura 2-3) que busca, examina y ejecuta las instrucciones de otro programa se llama **intérprete**, como mencionamos en el capítulo 1.

Esta equivalencia entre procesadores en hardware e intérpretes tiene importantes implicaciones para la organización de las computadoras y el diseño de los sistemas de computación. Después de haber especificado el lenguaje de máquina, *L*, para una computadora nueva, el equipo de diseño puede decidir si quiere construir un procesador en hardware que ejecute directamente los programas en *L* o si prefieren escribir un intérprete que interprete programas en *L*. Si optan por escribir un intérprete, también deberán crear una máquina en hardware que ejecute el intérprete. También son posibles ciertas construcciones híbridas, con ejecución parcial en hardware y también algo de interpretación en software.


```
public class Interp {
   static int PC;                          // contador de programa; contiene dir.de la sig. instruc.
   static int AC;                          // acumulador; registro para efectuar op. aritméticas
   static int instr;                       // registro para retener la instrucción en curso
   static int tipo_instr;                  // el tipo de instrucción (código de operación)
   static int pos_datos;                   // la dirección de los datos, o –1 si no hay datos
   static int datos;                       // contiene el operando en curso
   static boolean bit_ejec = true;         // bit que puede apagarse para detener la máquina

   public static void interpretar(int memoria[], int dir_inicio) {
     // Este procedimiento interpreta programas para una máquina sencilla cuyas instrucciones
     // tienen un operando en la memoria. La máquina tiene un registro AC (acumulador) que se
     // usa en las operaciones aritméticas. La instrucción ADD suma un entero de la memoria al
     // AC, por ejemplo. El intérprete continúa ejecutándose hasta que la instrucción HALT
     // apaga el bit de ejecución. El estado de un proceso que se ejecuta en esta máquina
     // consiste en la memoria, el contador de programa, el bit de ejecución y el AC. Los parámetros
     // de entrada consisten en la imagen de memoria y la dirección de inicio.

     PC = dir_inicio;
     while (bit_ejec) {
       instr = memoria[PC];                          // traer siguiente instrucción a instr
       PC = PC + 1 ;                                 // incrementar contador de programa
       tipo_instr = obt_tipo_instr(instr);           // determinar tipo de instrucción
       pos_datos = encontrar_datos(instr, tipo_instr); // localizar los datos (–1 si no hay)
       if (pos_datos >= 0)                           // si pos_datos es –1 no hay operando
       datos = memoria[pos_datos];                   // traer los datos
       ejecutar(tipo_instr, datos);                  // ejecutar la instrucción
       }

   }

   private static int obt_tipo_instr(int direc) {...}
   private static int encontrar_datos(int instr, int tipo) {...}
   private static void ejecutar(int tipo, int datos) {...}

}
```

Figura 2-3. Intérprete para una computadora simple (escrito en Java).

Un intérprete descompone las instrucciones de su máquina objetivo en pasos pequeños. Por consiguiente, la máquina en la que el intérprete se ejecuta puede ser mucho más simple y menos costosa que un procesador en hardware para la máquina objetivo. Este ahorro es más importante aún si la máquina objetivo tiene un gran número de instrucciones y éstas son complicadas, con muchas opciones. El ahorro se logra básicamente por la sustitución de hardware por software (el intérprete).

Las primeras computadoras tenían conjuntos de instrucciones pequeños y sencillos, pero la búsqueda de computadoras más potentes llevó, entre otras cosas, a instrucciones individuales más potentes. Casi desde el principio se descubrió que con instrucciones más comple-

jas a menudo era posible acelerar la ejecución de los programas, aunque las instrucciones individuales tardaran más en ejecutarse. Una instrucción de punto flotante es un ejemplo de instrucción más compleja. Otro ejemplo es el apoyo directo para acceder a los elementos de un arreglo. A veces la cosa era tan sencilla como observar que con frecuencia ocurrían las mismas dos instrucciones en sucesión, de modo que una sola instrucción podía realizar el trabajo de ambas.

Las instrucciones más complejas eran mejores porque a veces hacían posible traslapar la ejecución de operaciones individuales o ejecutarlas en paralelo empleando diferente hardware. En las costosas computadoras de alto rendimiento el costo de este hardware adicional podía justificarse fácilmente. Fue así como las computadoras de alto precio y mejor rendimiento comenzaron a tener mucho más instrucciones que las de bajo costo. Sin embargo, el creciente costo de la creación de software y las necesidades de compatibilidad de las instrucciones hicieron necesario implementar instrucciones complejas incluso en las computadoras de bajo costo en las que el precio era más importante que la velocidad.

Hacia fines de los cincuenta, IBM (que entonces era la compañía que dominaba el campo de las computadoras) se había dado cuenta de las muchas ventajas, tanto para IBM como para sus clientes, que tenía que manejar una sola familia de máquinas, todas las cuales ejecutaban las mismas instrucciones. IBM introdujo el término **arquitectura** para describir este nivel de compatibilidad. Una nueva familia de computadoras tenía una sola arquitectura pero muchas implementaciones diferentes, todas las cuales podían ejecutar el mismo programa, la única diferencia era el precio y la velocidad. Pero, ¿cómo construir una computadora de bajo costo que pudiera ejecutar todas las complejas instrucciones de las máquinas de alto costo y mayor rendimiento?

La respuesta fue la interpretación. Esta técnica, inicialmente sugerida por Wilkes (1951), permitió diseñar computadoras sencillas, de bajo costo, capaces de ejecutar un gran número de instrucciones. El resultado fue la arquitectura IBM System/360, una familia de computadoras compatibles que abarcaba casi dos órdenes de magnitud, tanto en precio como en capacidad. Sólo se usaba una implementación directa en hardware (es decir, no interpretada) en los modelos más caros.

Las computadoras sencillas con instrucciones interpretadas también tenían otros beneficios. Los más importantes eran:

1. La capacidad para corregir en el campo instrucciones mal implementadas, o incluso subsanar deficiencias de diseño en el hardware básico.
2. La oportunidad de añadir nuevas instrucciones con un costo mínimo, aun después de haber entregado la máquina.
3. Diseño estructurado que permite crear, probar y documentar instrucciones complejas de forma eficiente.

Cuando el mercado de las computadoras hizo explosión en la década de los setenta, y la capacidad de las máquinas creció rápidamente, la demanda de computadoras de bajo costo favoreció los diseños que utilizaban intérpretes. La capacidad para adaptar el hardware y el intérprete a un conjunto de instrucciones específico surgió como un diseño mucho muy económico para los procesadores. Al avanzar a pasos agigantados la tecnología subyacente de los semiconductores, las ventajas de costo sobrepasaron las oportunidades de lograr mayor rendimiento, y las arquitecturas basadas en intérpretes se convirtieron en la forma convencional de diseñar computadoras. Casi todas las computadoras nuevas diseñadas en los setenta, desde las minicomputadoras hasta las mainframes, se basaron en interpretación.

Hacia finales de esa década, el uso de procesadores simples que ejecutaban intérpretes se había extendido a todos los modelos, con excepción de los más costosos y rápidos como la Cray-1 y la serie Cyber de Control Data. El uso de un intérprete eliminaba las limitaciones de costo inherentes a las instrucciones complejas, y las arquitecturas comenzaron a explorar instrucciones más complejas, sobre todo las formas de especificar los operandos a utilizar.

La tendencia llegó a su cenit con la computadora VAX de Digital Equipment Corporation, que tenía varios cientos de instrucciones y más de 200 formas distintas de especificar los operandos a usar en cada instrucción. Desafortunadamente, la arquitectura VAX se concibió desde el principio para implementarse con un intérprete, sin pensar mucho en la implementación de un modelo de alto rendimiento. Esta filosofía llevó a la inclusión de un gran número de instrucciones de valor marginal que eran difíciles de ejecutar de manera directa. Esta omisión resultó fatal para VAX, y en última instancia también para DEC (Compaq compró a DEC en 1998).

Si bien los primeros microprocesadores de 8 bits eran máquinas muy sencillas con conjuntos de instrucciones muy simples, para fines de la década de los años setenta incluso los microprocesadores habían adoptado diseños basados en intérpretes. Durante este periodo, uno de los principales retos que los diseñadores de microprocesadores enfrentaban era controlar la creciente complejidad que los circuitos integrados hacían posible. Una ventaja importante del enfoque basado en intérpretes era la posibilidad de diseñar un procesador sencillo, dejando casi todo lo complejo para la memoria que contenía el intérprete. Así, un diseño de hardware complejo podía convertirse en un diseño de software complejo.

El éxito del Motorola 68000, que tenía un conjunto de instrucciones grande interpretado, y el fracaso concurrente del Zilog Z8000 (que tenía un conjunto de instrucciones igualmente grande, pero sin intérprete) demostró las ventajas de los intérpretes para sacar al mercado rápidamente un microprocesador nuevo. Este éxito fue más sorprendente en vista de la ventaja inicial de Zilog (el predecesor del Z8000, el Z80, era mucho más popular que el predecesor del 68000, el 6800). Desde luego, otros factores influyeron en este desenlace, como la larga historia de Motorola como fabricante de chips y la larga historia de Exxon (el dueño de Zilog) como compañía petrolera, no como fabricante de chips.

Otro factor que actuaba en favor de la interpretación durante esa era fue la existencia de memorias sólo de lectura rápidas, llamadas **almacenes de control**, para contener los intérpretes. Supongamos que una instrucción interpretada representativa del 68000 requiriera 10 instrucciones del intérprete, llamadas **microinstrucciones**, a 100 ns cada una, y dos referencias a la memoria principal, a 500 ns cada una. El tiempo de ejecución total era entonces de 2000 ns, sólo el doble de la rapidez máxima que podía lograrse con ejecución directa. Si no se contara con almacén de control, la instrucción habría tardado 6000 ns. Una reducción en la velocidad por un factor de 6 es mucho más difícil de aceptar que una reducción por un factor de 2.

2.1.3 RISC *versus* CISC

A finales de los setenta se efectuaron muchos experimentos con instrucciones muy complejas, que eran posibles gracias al intérprete. Los diseñadores trataron de salvar la "brecha semántica" entre lo que las máquinas podían hacer y lo que los lenguajes de programación de alto nivel requerían. A casi nadie se le ocurría diseñar máquinas más sencillas, como ahora casi nadie piensa en diseñar sistemas operativos, redes, procesadores de textos, etc. menos potentes (lo cual tal vez es lamentable).

Un grupo que se opuso a la tendencia y trató de incorporar algunas de las ideas de Seymour Cray en una minicomputadora de alto rendimiento fue el encabezado por John Cocke en IBM. Estos trabajos tuvieron como resultado una minicomputadora experimental llamada **801**. Aunque IBM nunca comercializó esta máquina y los resultados no se publicaron sino hasta varios años después (Radin, 1982), la información se filtró y otras personas comenzaron a investigar arquitecturas similares.

En 1980, un grupo de Berkeley dirigido por David Patterson y Carlo Séquin comenzó a diseñar chips de CPU VLSI que no utilizaban interpretación (Patterson, 1985; Patterson y Séquin, 1982). Ellos acuñaron el término **RISC** para este concepto y llamaron a su chip de CPU RISC I, el cual fue seguido en poco tiempo por el RISC II. Un poco después, en 1981, del otro lado de la Bahía de San Francisco en Stanford, John Hennessy diseñó y fabricó un chip un tanto diferente al que llamó **MIPS** (Hennessy, 1984). Estos chips llegaron a convertirse en productos importantes desde el punto de vista comercial: SPARC y MIPS, respectivamente.

Estos nuevos procesadores tenían diferencias significativas respecto a los procesadores comerciales de la época. Dado que estas nuevas CPU no tenían que ser compatibles con productos existentes, sus diseñadores estaban en libertad de escoger nuevos conjuntos de instrucciones que maximizaran el rendimiento total del sistema. Si bien el hincapié inicial fue en instrucciones simples que pudieran ejecutarse con rapidez, pronto se vio que la clave para un buen desempeño era diseñar instrucciones que pudieran **emitirse** (iniciarse) rápidamente. El tiempo real que una instrucción tardaba era menos importante que el número de instrucciones que podían iniciarse por segundo.

En la época en que se estaban diseñando por primera vez estos sencillos procesadores la característica que llamó la atención de todo mundo fue el número relativamente pequeño de instrucciones disponibles, por lo regular unas 50. Este número era mucho menor que las 200 a 300 que tenían computadoras establecidas como la DEC VAX y las grandes mainframes de IBM. De hecho, el acrónimo RISC significa **computadora de conjunto de instrucciones reducido** (*Reduced Instruction Set Computer*), lo que contrasta con CISC, que significa **computadora de conjunto de instrucciones complejo** (*Complex Instruction Set Computer*) (una referencia poco sutil al VAX, que dominaba los departamentos de ciencias de la computación universitarios en ese entonces). Hoy día, poca gente piensa que el tamaño del conjunto de instrucciones sea crucial, pero el nombre ha persistido.

Para no hacer muy larga la historia, se desencadenó una gran guerra de religión, con los partidarios de RISC atacando el orden establecido (VAX, Intel, las grandes mainframes de IBM). Ellos aseguraban que la mejor forma de diseñar una computadora era tener un número

reducido de instrucciones simples que se ejecutaran en un ciclo del camino de datos de la figura 2-2, buscar dos registros, combinarlos de alguna manera (por ejemplo, suma o AND) y almacenar el resultado en un registro nuevamente. Su argumento era que incluso si una máquina RISC requería cuatro o cinco instrucciones para hacer lo que una máquina CISC hace en una instrucción, si las instrucciones RISC son 10 veces más rápidas (porque no se interpretan), RISC gana. También vale la pena señalar que para esas fechas la rapidez de las memorias principales había alcanzado a la rapidez de los almacenes de control sólo de lectura, así que el castigo por interpretación se había incrementado considerablemente, lo que favorecía a las máquinas RISC.

Podríamos pensar que en vista de las ventajas de velocidad de la tecnología RISC, las máquinas RISC (como la Alpha de DEC) habrían sacado a las máquinas CISC (como la Intel Pentium) del mercado. No ha sucedido tal cosa. ¿Por qué?

En primer lugar, está la cuestión de la compatibilidad con lo existente y los miles de millones de dólares que las compañías han invertido en software para la línea Intel. Segundo, aunque parezca sorprendente, Intel ha logrado aplicar las mismas ideas incluso en una arquitectura CISC. A partir del 486, las CPU de Intel contienen un núcleo RISC que ejecuta las instrucciones más simples (y casi siempre las más comunes) en un solo ciclo del camino de datos, al tiempo que interpreta las instrucciones más complejas de la forma CISC acostumbrada. El resultado neto es que las instrucciones comunes son rápidas, y las menos comunes son lentas. Si bien este enfoque híbrido no es tan rápido como un diseño RISC puro, ofrece un desempeño global competitivo y al mismo tiempo permite la ejecución de software viejo sin modificación.

2.1.4 Principios de diseño de las computadoras modernas

Ahora, más de una década después de la introducción de las primeras máquinas RISC, ciertos principios de diseño gozan de aceptación como una buena forma de diseñar computadoras dado el estado actual de la tecnología de hardware. Si ocurre un cambio importante en la tecnología (digamos que un nuevo proceso de fabricación hace que un ciclo de memoria tarde 10 veces menos que un ciclo de CPU), cualquier cosa podría suceder. Por tanto, los diseñadores de máquinas siempre deben estar pendientes de los avances tecnológicos que podrían afectar el equilibrio entre los componentes.

Habiendo dicho eso, existe un conjunto de principios de diseño, a veces llamados **principios de diseño RISC**, que los arquitectos de las CPU de propósito general hacen lo posible por seguir. Restricciones externas, como el requisito de ser compatible con alguna arquitectura existente, a menudo requieren concesiones ocasionales, pero estos principios son metas que la mayoría de los diseñadores se fija. A continuación analizaremos los principios más importantes.

Todas las instrucciones se ejecutan directamente en hardware

El hardware ejecuta directamente todas las instrucciones comunes; éstas no se interpretan con microinstrucciones. La eliminación de un nivel de interpretación hace que la mayor parte

de las instrucciones sean rápidas. En las computadoras que implementan conjuntos de instrucciones CISC, las instrucciones más complejas pueden dividirse en partes discretas que luego pueden ejecutarse como una sucesión de microinstrucciones. Este paso adicional hace más lenta a la máquina, pero podría ser aceptable en el caso de instrucciones que ocurren con poca frecuencia.

Maximizar el ritmo con que se emiten instrucciones

Las computadoras modernas recurren a muchos trucos para maximizar su desempeño, y el principal de ellos es tratar de emitir el mayor número posible de instrucciones por segundo. Después de todo, si podemos emitir 500 millones de instrucciones por segundo, habremos construido un procesador de 500 MIPS, sin importar cuánto tarden en ejecutarse realmente las instrucciones. (**MIPS** significa Millones de Instrucciones Por Segundo; el procesador MIPS pretende hacer un juego de palabras con este acrónimo.) Este principio sugiere que el paralelismo puede desempeñar un papel importante en el mejoramiento del desempeño, ya que sólo es posible emitir un gran número de instrucciones lentas en un periodo corto si varias instrucciones pueden ejecutarse simultáneamente.

Aunque las instrucciones siempre se encuentran en el orden que dicta el programa, no siempre se emiten en ese orden (porque algún recurso necesario podría estar ocupado) y no tienen que terminar en el orden del programa. Desde luego, si la instrucción 1 asigna un valor a un registro y la instrucción 2 usa ese registro, debe tenerse mucho cuidado de que la instrucción 2 no lea el registro antes de que éste contenga el valor correcto. Para ello se requiere un control estricto, pero el potencial para incrementar la rapidez ejecutando varias instrucciones simultáneamente es grande.

Las instrucciones deben ser fáciles de decodificar

Un límite crítico de la rapidez con que se emiten las instrucciones es la decodificación de instrucciones individuales para determinar qué recursos necesitan. Todo lo que pueda agilizar este proceso es útil, e incluye hacer que las instrucciones tengan una longitud fija, con un número pequeño de campos. Cuanto menor sea el números de formatos de instrucciones distintos, mejor.

Sólo las operaciones de carga y almacenamiento deben hacer referencia a la memoria

Una de las formas más sencillas de desglosar las operaciones en pasos individuales es requerir que los operandos de casi todas las instrucciones provengan de —y regresen a—, registros. La operación de transferir operandos entre la memoria y los registros puede realizarse con instrucciones diferentes. Puesto que el acceso a la memoria puede tardar mucho, y el retraso es impredecible, la mejor manera de traslapar estas instrucciones con otras es cerciorarse de que no hagan más que transferir operandos entre los registros y la memoria. Esta observación implica que sólo las instrucciones LOAD y STORE deben hacer referencia a la memoria.

Incluir abundantes registros

Puesto que el acceso a la memoria es relativamente lento, es necesario contar con muchos registros (al menos 32) para que, una vez que se ha obtenido una palabra, se pueda mantener en un registro hasta que ya no se necesite. Quedarse sin registros y tener que escribirlos en la memoria sólo para volverlos a cargar después es poco recomendable y debe evitarse en la medida de lo posible. La mejor manera de lograr esto es tener suficientes registros.

2.1.5 Paralelismo en el nivel de instrucciones

Los arquitectos de computadoras se esfuerzan constantemente por mejorar el desempeño de las máquinas que diseñan. Una forma de hacerlo es aumentar la velocidad de reloj de los chips para que operen con mayor rapidez, pero para cada nuevo diseño existe un límite para lo que es posible lograr con los recursos disponibles en ese momento histórico. Por consiguiente, casi todos los arquitectos de computadoras recurren al paralelismo (hacer dos o más cosas al mismo tiempo) para sacar el mayor provecho posible a una velocidad de reloj dada.

El paralelismo adopta dos formas generales: paralelismo en el nivel de instrucciones y paralelismo en el nivel de procesador. En el primero, se aprovecha el paralelismo dentro de las instrucciones individuales para lograr que la máquina ejecute más instrucciones por segundo. En el segundo, múltiples CPU trabajan juntas en el mismo problema. Cada enfoque tiene sus propios méritos. En esta sección examinaremos el paralelismo en el nivel de instrucciones; en la que sigue veremos el paralelismo en el nivel de procesadores.

Filas de procesamiento

Se ha sabido desde hace años que la obtención de instrucciones de la memoria es un importante cuello de botella que afecta la rapidez de ejecución de las instrucciones. Para aliviar este problema, computadoras que se remontan por lo menos a la IBM Stretch (1959) han contado con la capacidad de buscar instrucciones de la memoria por adelantado, a fin de tenerlas disponibles en el momento en que se necesitan. Esas instrucciones se almacenaban en una serie de registros llamados **buffer de prebúsqueda**. Así, cuando se requería una instrucción, casi siempre podía tomarse del buffer de prebúsqueda en lugar de esperar a que terminara una lectura de la memoria.

Efectivamente, la prebúsqueda divide la ejecución de instrucciones en dos partes: búsqueda y ejecución propiamente dicha. El concepto de **fila de procesamiento** (*pipeline*) lleva esta estrategia mucho más lejos. En lugar de dividir la ejecución de instrucciones en sólo dos partes, a menudo se divide en muchas partes, cada una de las cuales se maneja con un componente de hardware dedicado, y todos estos componentes pueden operar en paralelo.

La figura 2-4(a) ilustra una fila de procesamiento con cinco unidades, también llamadas **etapas**. La etapa 1 busca la instrucción de la memoria y la coloca en un buffer hasta que se necesita. La etapa 2 decodifica la instrucción, determinando de qué tipo es y qué operandos necesita. La etapa 3 localiza y busca los operandos, sea de registros o de la memoria. La etapa 4 se encarga de ejecutar propiamente la instrucción, casi siempre haciendo pasar los operandos

por el camino de datos de la figura 2-2. Por último, la etapa 5 escribe el resultado en el registro apropiado.

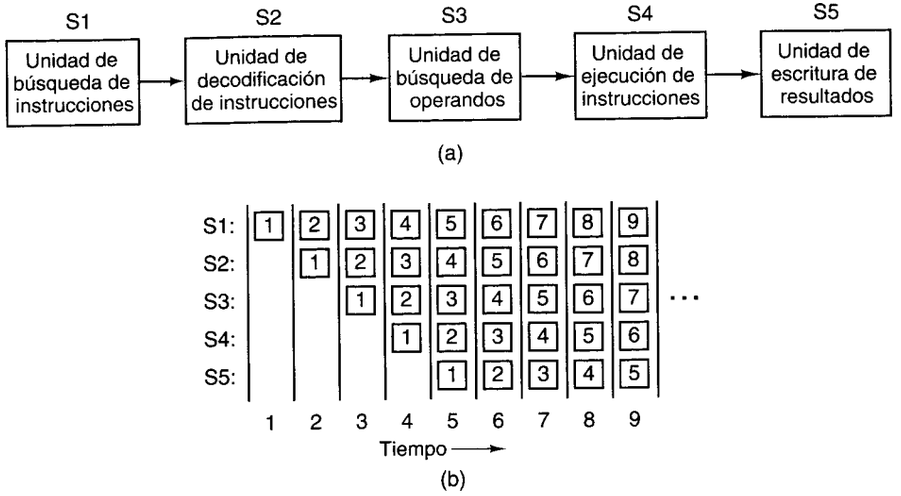


Figura 2-4. (a) Fila de procesamiento de cinco etapas. (b) Estado de cada etapa en función del tiempo. Se ilustran nueve ciclos de reloj.

En la figura 2-4(b) vemos cómo la fila de procesamiento opera como una función del tiempo. Durante el ciclo de reloj 1, la etapa S1 está trabajando en la instrucción 1, buscándola en la memoria. Durante el ciclo 2, la etapa S2 decodifica la instrucción 1, mientras la etapa S1 busca la instrucción 2. Durante el ciclo 3, la etapa S3 busca los operandos para la instrucción 1, la etapa S2 decodifica la instrucción 2 y la etapa S1 busca la tercera instrucción. Durante el ciclo 4, la etapa S4 ejecuta la instrucción 1, S3 busca los operandos para la instrucción 2, S2 decodifica la instrucción 3 y S1 busca la instrucción 4. Por último, durante el ciclo 5, S5 escribe el resultado de la instrucción 1, mientras las demás etapas trabajan con las instrucciones subsecuentes.

Consideremos una analogía para aclarar el concepto de fila de procesamiento. Imagine una fábrica de pasteles en la que se separa la preparación de los pasteles y el empacado de éstos para su distribución. Supongamos que el departamento de empacado tiene una larga banda transportadora con cinco trabajadores (unidades de procesamiento) formados junto a ella. Cada 10 segundos (el ciclo de reloj), el trabajador 1 coloca una caja de pastel vacía en la banda. La caja es transportada hasta el trabajador 2, quien coloca un pastel en ella. Poco después, la caja llega al puesto del trabajador 3, donde se le cierra y sella. Luego, la caja continúa hasta donde está el trabajador 4, quien adhiere una etiqueta a la caja. Por último, el trabajador 5 retira la caja de la banda y la coloca en un contenedor grande que después se enviará a un supermercado. Básicamente, ésta es la forma en que funcionan también las filas de procesamiento de una computadora: cada instrucción (pastel) pasa por varios pasos de procesamiento antes de salir completada por el otro extremo.

Volviendo a nuestra fila de procesamiento de la figura 2-4, supongamos que el tiempo de ciclo de esta máquina es de 2 ns. Entonces, se requieren 10 ns para que una instrucción pase por las cinco etapas de la fila de procesamiento. A primera vista, si una instrucción tarda 10 ns, podría parecer que la máquina puede operar a 100 MIPS, pero en realidad trabaja a una velocidad mucho mayor. En cada ciclo de reloj (2 ns), se completa una instrucción, así que la rapidez de procesamiento real es de 500 MIPS, no de 100 MIPS.

El uso de filas de procesamiento permite balancear la **latencia** (el tiempo que tarda en ejecutarse una instrucción) y el **ancho de banda del procesador** (cuántas MIPS puede ejecutar la CPU). Con un tiempo de ciclo de T ns, y n etapas en la fila de procesamiento, esta latencia es de nT ns y el ancho de banda es de $1000/T$ MIPS (lógicamente, dado que estamos midiendo tiempos en nanosegundos, deberíamos medir el ancho de banda de la CPU en MMIPS o GIPS, pero nadie lo hace, así que tampoco lo haremos aquí).

Arquitecturas superescalares

Si una fila de procesamiento es buena, entonces seguramente dos serán mejores. En la figura 2-5 se muestra un posible diseño para una CPU con fila de procesamiento dual basado en la figura 2-4. Aquí una sola unidad de búsqueda de instrucciones trae pares de instrucciones y coloca cada una en su propia fila de procesamiento, que cuenta con su propia ALU para poder operar en paralelo. Esto requiere que las dos instrucciones no compitan por el uso de recursos (por ejemplo, registros) y también que una no dependa del resultado de la otra. Al igual que con una solo fila de procesamiento, el compilador debe garantizar que se cumpla esta situación (lo que implica que el hardware no verifica que las instrucciones sean compatibles, y produce resultados incorrectos si no lo son), o bien los conflictos se detectan y eliminan durante la ejecución utilizando hardware adicional.

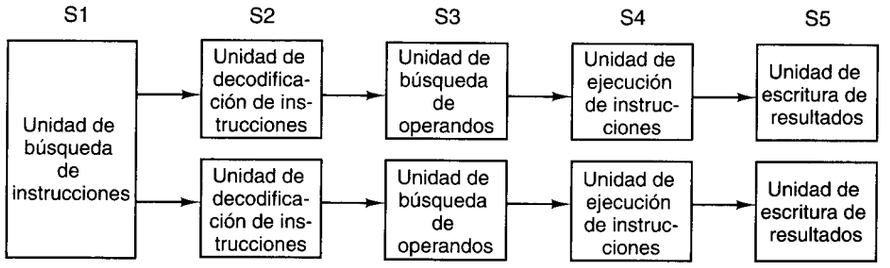


Figura 2-5. Filas de procesamiento dual de cinco etapas con unidad de búsqueda de instrucciones común.

Aunque las filas de procesamiento, sean sencillas o dobles, se usan principalmente en las máquinas RISC (el 386 y sus predecesores no los tenían), a partir del 486 Intel comenzó a introducir filas de procesamiento en sus CPU. El 486 tenía una fila de procesamiento y el Pentium tenía dos filas de procesamiento de cinco etapas más o menos como en la figura 2-5, aunque la división exacta del trabajo entre las etapas 2 y 3 (llamadas decodificar-1 y decodificar-2) era un poco diferente de nuestro ejemplo. La fila de procesamiento principal, llamada **fila de**

procesamiento u, podía ejecutar una instrucción Pentium arbitraria. La segunda fila de procesamiento, llamada **fila de procesamiento v**, sólo podía ejecutar instrucciones enteras simples (y una instrucción de punto flotante sencilla, FXCH).

Unas reglas complejas determinaban si un par de instrucciones eran compatibles y podían ejecutarse en paralelo. Si las instrucciones de un par no eran suficientemente simples o eran incompatibles, sólo se ejecutaba la primera (en la fila de procesamiento u). La segunda se retenía y se apareaba con la siguiente instrucción. Las instrucciones siempre se ejecutaban en orden. Así, compiladores específicos para Pentium que producían pares compatibles podían generar programas de más rápida ejecución que los compiladores más antiguos. Mediciones efectuadas revelaron que un Pentium que ejecutaba código optimado para él era exactamente dos veces más rápido con programas de manipulación de enteros que un 486 que operaba con la misma rapidez de reloj (Pountain, 1993). Esta ganancia podía atribuirse totalmente a la segunda fila de procesamiento.

Es concebible usar cuatro filas de procesamiento, pero ello implica duplicar demasiado hardware (los computólogos, a diferencia de los especialistas en el folklore, no creen en el número tres). En vez de ello, en las CPU de extremo superior se sigue una estrategia distinta. La idea básica es tener una sola fila de procesamiento pero proporcionarle varias unidades funcionales, como se muestra en la figura 2-6. Por ejemplo, el Pentium II tiene una estructura similar a esta figura, y la veremos en el capítulo 4. Se acuñó el término **arquitectura superescalar** en 1987 para describir este enfoque (Agerwala y Cocke, 1987). Sin embargo, sus raíces se remontan a hace más de 30 años con la computadora CDC 6600. La 6600 obtenía una instrucción cada 100 ns y la transfería a una de 10 unidades funcionales para su ejecución en paralelo mientras la CPU iba en busca de la siguiente instrucción.

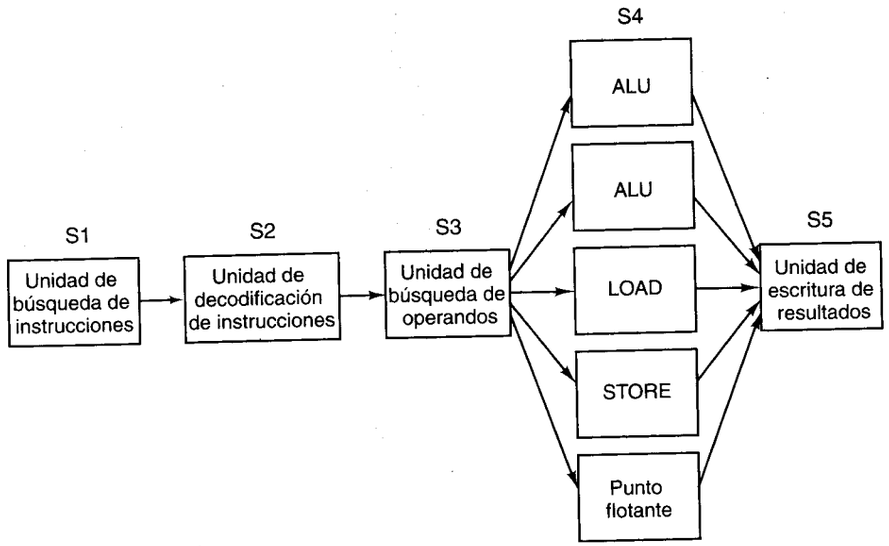


Figura 2-6. Procesador superescalar con cinco unidades funcionales.

El concepto de procesador superescalar lleva implícita la idea de que la etapa S3 puede emitir instrucciones con mucho mayor rapidez de la que la etapa S4 puede ejecutarlas. Si la etapa S3 emitiera una instrucción cada 10 ns y todas las unidades funcionales pudieran efectuar su trabajo en 10 ns, sólo una estaría ocupada en un momento dado, y la idea no representaría ninguna ventaja. En realidad, casi todas las unidades funcionales de la etapa 4 tardan mucho más que un ciclo de reloj en ejecutarse, sobre todo las que acceden a la memoria o realizan aritmética de punto flotante. Como puede verse en la figura, es posible tener varios ALU en la etapa 4.

2.1.6 Paralelismo en el nivel de procesador

La demanda de computadoras más y más rápidas parece insaciable. Los astrónomos quieren simular qué sucedió en el primer microsegundo después del *big bang*, los economistas quieren modelar la economía mundial, y los adolescentes quieren tener juegos multimedia interactivos en 3D por Internet con sus amigos virtuales. Si bien las CPU son cada vez más rápidas, llegará el momento en que se toparán con la barrera de la velocidad de la luz, que con toda seguridad no podrá rebasar los 20 cm/ns en alambre de cobre o fibra óptica, por más ingeniosos que sean los ingenieros de Intel. Los chips más rápidos también producen más calor, y su disipación es un problema.

El paralelismo en el nivel de instrucciones ayuda un poco, pero las filas de procesamiento y el funcionamiento superescalar casi nunca ganan más que un factor de 5 o 10. Para obtener ganancias de 50, 100 o más, el único camino es diseñar computadoras con múltiples CPU, por lo que ahora veremos cómo están organizadas algunas de ellas.

Computadoras de matriz

Muchos problemas de las ciencia física e ingeniería implican matrices o tienen una estructura altamente regular en algún sentido. En muchos casos los mismos cálculos se efectúan con muchos conjuntos de datos distintos al mismo tiempo. La regularidad y estructura de estos programas los convierte en candidatos idóneos para una aceleración por ejecución en paralelo. Se han utilizado dos métodos para ejecutar programas científicos grandes rápidamente. Si bien estos dos sistemas son muy similares en casi todos los sentidos, resulta irónico que uno de ellos se considere como una extensión de un solo procesador, mientras que el otro se considera como una computadora paralela.

Un **arreglo de procesadores** consiste en un gran número de procesadores idénticos que ejecutan la misma secuencia de instrucciones con diferentes conjuntos de datos. El primer arreglo de procesadores del mundo fue la computadora ILLIAC IV de la University of Illinois, que se ilustra en la figura 2-7 (Bouknight *et al.*, 1972). El plan original era construir una máquina constituida por cuatro cuadrantes, cada uno de los cuales tenía una cuadrícula 8×8 de elementos procesador/memoria. Una sola unidad de control por cuadrante transmitía instrucciones que eran ejecutadas simultáneamente por todos los procesadores, cada uno de los cuales utilizaba sus propios datos de su propia memoria (que se cargaba durante la fase de inicialización). Debido a que el presupuesto se excedió en un factor de cuatro, sólo llegó a

construirse un cuadrante, pero éste logró un desempeño de 50 megaflops (millones de operaciones de punto flotante por segundo). Se dice que si se hubiera completado la máquina original y hubiera alcanzado la meta de desempeño fijada originalmente (1 gigaflop), habría duplicado la capacidad de cómputo de todo el mundo.

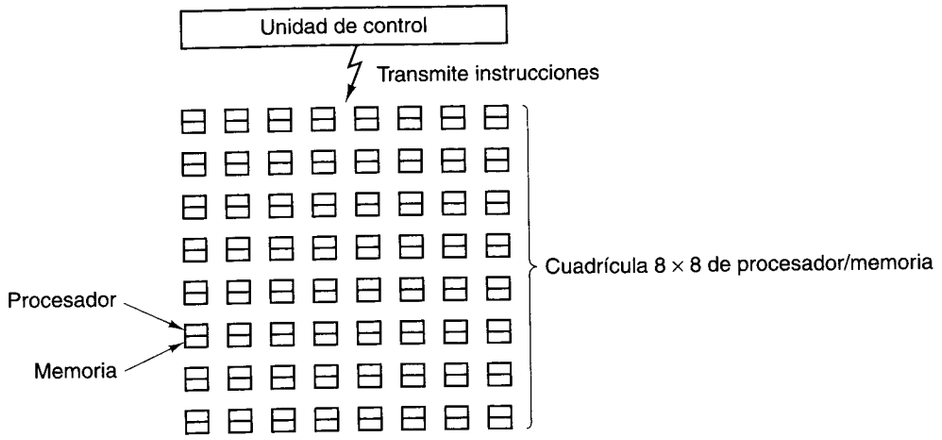


Figura 2-7. Arreglo de procesadores del tipo ILLIAC IV.

Para el programador, un **procesador vectorial** es muy parecido a un arreglo de procesadores. Al igual que éste, el procesador vectorial es muy eficiente al ejecutar una secuencia de operaciones con pares de elementos de datos. Pero, a diferencia del arreglo de procesadores, todas las operaciones de suma se efectúan en un solo sumador provisto de muchas fila de procesamientos. La compañía fundada por Seymour Cray, Cray Research, produjo muchos procesadores vectoriales, comenzando con la Cray-1 en 1974 y continuando hasta los modelos actuales (Cray Research forma parte ahora de SGI).

Tanto los arreglos de procesadores como los procesadores vectoriales trabajan con arreglos de datos. Ambos ejecutan instrucciones individuales que, por ejemplo, suman los elementos de dos vectores en pares. Pero mientras que el arreglo de procesadores lo hace teniendo tantos sumadores como elementos tiene el vector, el procesador vectorial tiene el concepto de **registro vectorial**, que consiste en un conjunto de registros convencionales que pueden cargarse desde la memoria con una sola instrucción, la cual en realidad los carga desde la memoria en serie. Luego una instrucción de suma vectorial realiza la suma por pares de los elementos de dos vectores de este tipo alimentándolos a un sumador con filas de procesamiento desde los dos registros vectoriales. El resultado del sumador es otro vector, que puede almacenarse en un registro vectorial o utilizarse directamente como operando de otra operación vectorial.

Se siguen fabricando arreglos de procesadores, pero ocupan un nicho cada vez más pequeño del mercado, ya que sólo funcionan bien con problemas que requieren realizar el mismo cálculo con muchos conjuntos de datos simultáneamente. Los arreglos de procesadores pueden realizar algunas operaciones de forma más eficiente que las computadoras vectoriales, pero requieren mucho más hardware y su programación es notoriamente difícil. Los

procesadores vectoriales, en cambio, pueden añadirse a un procesador convencional. El resultado es que las partes del programa susceptibles de **vectorizarse** pueden ejecutarse rápidamente aprovechando la unidad vectorial, mientras que el resto del programa puede ejecutarse en un procesador único convencional.

Multiprocesadores

Los elementos de procesamiento de un arreglo de procesadores no son CPU independientes, ya que todos ellos comparten la misma unidad de control. Nuestro primer sistema paralelo constituido por CPU propiamente dichas es el **multiprocesador**, un sistema con varias CPU que comparten una memoria común, como un grupo de personas en un salón que comparten un pizarrón común. Puesto que cada CPU puede leer o escribir en cualquier parte de la memoria, deben coordinarse (en software) para no estorbarse mutuamente.

Hay varios posibles esquemas de implementación. El más sencillo consiste en tener un solo bus con varias CPU y una memoria conectadas a él. En la figura 2-8(a) se muestra un diagrama de semejante multiprocesador basado en bus. Muchas compañías fabrican este tipo de sistemas.

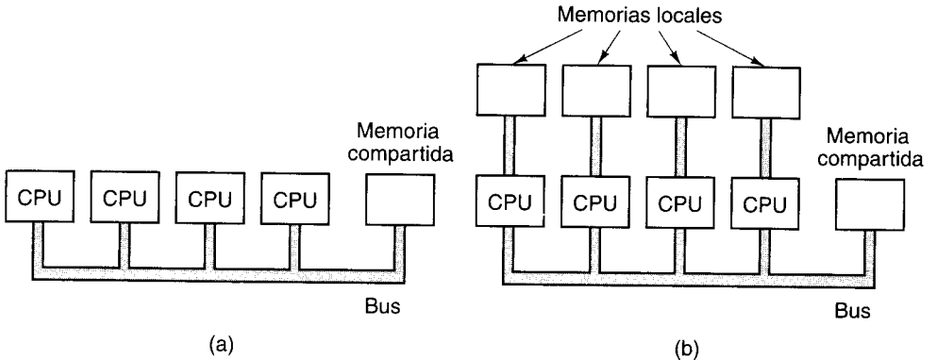


Figura 2-8. (a) Multiprocesador con un solo bus. (b) Multicomputadora con memorias locales.

No se necesita mucha imaginación para darse cuenta de que si hay un gran número de procesadores rápidos que constantemente están tratando de acceder a la memoria por el mismo bus, habrá conflictos. Los diseñadores de multiprocesadores han ideado varios esquemas para reducir esta contención y mejorar el desempeño. Un diseño, que se muestra en la figura 2-8(b), proporciona a cada procesador un poco de memoria local propia, inaccesible para los demás. Esta memoria puede utilizarse para código de programa y datos que no es necesario que se compartan. El acceso a esta memoria privada no usa el bus principal, lo que reduce considerablemente el tráfico en el bus. Hay otros posibles esquemas (por ejemplo, uso de cachés).

Los multiprocesadores tienen la ventaja, respecto a otros tipos de computadoras paralelas, de que es fácil trabajar con el modelo de programación de una sola memoria compartida. Por ejemplo, imagine un programa que busca células cancerosas en una fotografía de algún tejido

tomada a través de un microscopio. La fotografía digitalizada podría guardarse en la memoria común, y a cada procesador se le asignaría alguna región de la fotografía para su búsqueda. Puesto que cada procesador tiene acceso a toda la memoria, uno de ellos no tiene problemas para estudiar una célula que comienza en la región que se le asignó pero que rebasa la frontera con la siguiente región.

Multicomputadoras

Aunque los multiprocesadores con un número reducido de procesadores (≤ 64) son relativamente fáciles de construir, las dificultades se multiplican de forma sorprendente cuando aumenta el número de procesadores. Lo difícil es conectar todos los procesadores a la memoria. Para superar estos problemas, muchos diseñadores simplemente han abandonado la idea de tener una memoria compartida y se han limitado a construir sistemas que consisten en un gran número de computadoras interconectadas, cada una de las cuales tiene su propia memoria, sin que haya una memoria común. Estos sistemas se llaman **multicomputadoras**.

Las CPU de una multicomputadora se comunican enviándose mutuamente mensajes, parecidos al correo electrónico pero mucho más rápidos. En sistemas grandes, tener cada computadora conectada a todas las demás no resulta práctico, y se utilizan topologías como retículas bi y tridimensionales, árboles y anillos. El resultado es que los mensajes de una computadora a otra a menudo tienen que pasar por una o más computadoras intermedias o conmutadores para llegar del origen al destino. No obstante, es posible alcanzar sin mucha dificultad tiempos de paso de mensajes del orden de unos cuantos microsegundos. Se han construido multicomputadoras con casi 10,000 CPU, y ya están funcionando.

Puesto que los multiprocesadores son más fáciles de programar y las multicomputadoras son más fáciles de construir, se ha investigado asiduamente la posibilidad de diseñar sistemas híbridos que combinen las propiedades más atractivas de cada uno. Tales computadoras tratan de presentar la ilusión de una memoria compartida, sin hacer el gasto de construirla realmente. Estudiaremos con detalle los multiprocesadores y las multicomputadoras en el capítulo 8.

2.2 MEMORIA PRIMARIA

La **memoria** es la parte de la computadora en la que se almacenan programas y datos. Algunos especialistas en computación (sobre todo los británicos) emplean el término **almacén** (store) o **almacenamiento** (storage) en lugar de memoria (memory), aunque cada vez se extiende más el uso de "almacenamiento" para referirse al almacenamiento en disco. Sin una memoria en la cual los procesadores puedan leer y escribir información, no existirían las computadoras digitales de programa almacenado.

2.2.1 Bits

La unidad básica de memoria es el dígito binario, llamado **bit**. Un bit puede contener un 0 o un 1; es la unidad más simple posible. (Un dispositivo que sólo pudiera almacenar ceros difícilmente podría servir para construir un sistema de memoria; se requieren por lo menos dos valores.)

Suele decirse que las computadoras emplean aritmética binaria porque es "eficiente". Lo que quiere decirse (aunque pocas veces se es consciente de ello) es que es posible almacenar información digital distinguiendo entre diferentes valores de alguna cantidad física continua, como un voltaje o corriente. Cuantos más valores sea necesario distinguir, y menor separación haya entre valores adyacentes, menos confiable será la memoria. El sistema de numeración binario sólo requiere distinguir entre dos valores; por tanto, es el método más confiable para codificar información digital. Si no está familiarizado con los números binarios, vea el apéndice A.

En la publicidad de algunas computadoras, como las mainframes IBM grandes, se alardea de que tienen aritmética decimal además de binaria. Este truco se logra utilizando 4 bits para almacenar un dígito decimal empleando un código llamado **BCD (decimal codificado en binario**, *Binary Coded Decimal*). Cuatro bits se pueden combinar de 16 formas, diez de las cuales se usan para los dígitos del 0 al 9; seis combinaciones no se usan. A continuación se muestra el número 1944 codificado en decimal y en binario puro, empleando 16 bits en cada ejemplo:

decimal: 0001 1001 0100 0100 binario: 0000011110011000

Dieciséis bits en el formato decimal pueden almacenar los números del 0 al 9999, lo que da sólo 10,000 combinaciones, mientras que un número binario puro de 16 bits puede almacenar 65,536 combinaciones distintas. Es por esto que decimos que el almacenamiento binario es más eficiente.

Sin embargo, considere lo que sucedería si algún joven y brillante ingeniero eléctrico inventara un dispositivo electrónico altamente confiable que pudiera almacenar directamente los dígitos del 0 al 9 dividiendo la región de 0 a 10 volts en 10 intervalos. Cuatro de estos dispositivos podrían almacenar cualquier número decimal del 0 al 9999, y podrían formar 10,000 combinaciones. También podrían usarse cuatro de esos dispositivos para almacenar números binarios, pero sólo empleando 0 y 1. En tal caso, sólo podrían almacenar 16 combinaciones. Con tales dispositivos, el sistema decimal obviamente es más eficiente.

2.2.2 Direcciones de memoria

Las memorias consisten en varias **celdas** (o **localidades**), cada una de las cuales puede almacenar un elemento de información. Cada celda tiene un número, su **dirección**, con el cual los programas pueden referirse a ella. Si una memoria tiene n celdas, tendrán las direcciones 0 a $n - 1$. Todas las celdas de una memoria contienen el mismo número de bits. Si una celda consta de k bits, podrá contener cualquiera de 2^k combinaciones de bits distintas. En la figura 2-9 se muestran tres diferentes organizaciones para una memoria de 96 bits. Observe que celdas adyacentes tienen direcciones consecutivas (por definición).

Las computadoras que emplean el sistema de numeración binario (incluidas las notaciones octal y hexadecimal para números binarios) expresan las direcciones de memoria como números binarios. Si una dirección tiene m bits, el número máximo de celdas direccionables es 2^m. Por ejemplo, una dirección empleada para referirse a la memoria de la figura 2-9(a) necesita al menos 4 bits para expresar todos los números del 0 al 11. En cambio, basta una dirección de 3 bits para las figuras 2-9(b) y (c). El número de bits de la dirección determina el número máximo de celdas direccionables directamente en la memoria y es independiente

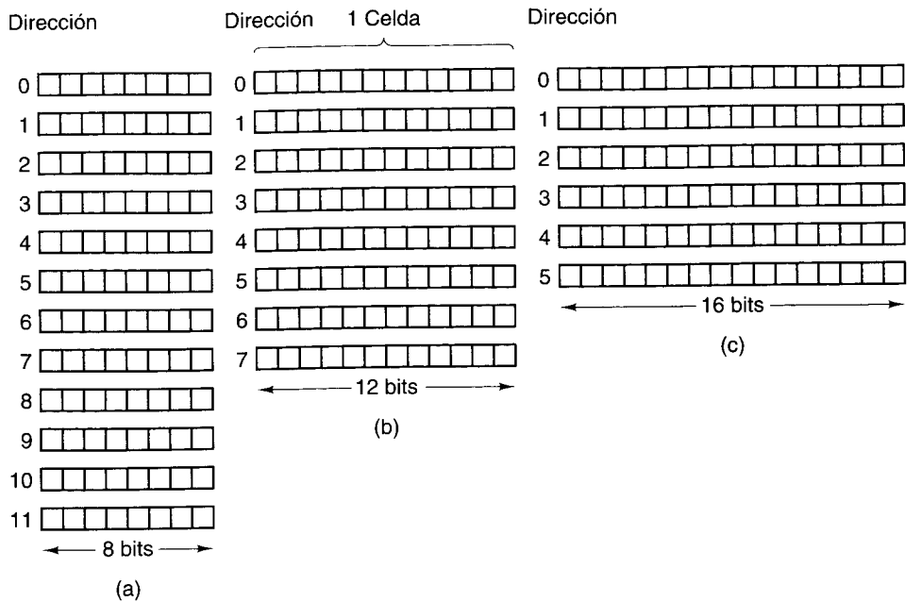


Figura 2-9. Tres formas de organizar una memoria de 96 bits.

del número de bits por celda. Una memoria que tiene 2^{12} celdas de 8 bits cada una, y una que tiene 2^{12} celdas de 64 bits cada una, necesitan ambas direcciones de 12 bits.

En la figura 2-10 se da el número de bits por celda de algunas computadoras que se han vendido comercialmente.

La importancia de la celda es que es la unidad direccionable más pequeña. En años recientes casi todos los fabricantes de computadoras han adoptado como estándar una celda de 8 bits, que recibe el nombre de **byte**. Los bytes se agrupan en **palabras**. Una computadora con palabras de 32 bits tiene 4 bytes/palabra, mientras que una con palabras de 64 bits tiene 8 bytes/palabra. La importancia de las palabras es que casi todas las instrucciones operan con palabras enteras; por ejemplo, suman dos palabras. Así, una máquina de 32 bits tiene registros de 32 bits e instrucciones para manipular palabras de 32 bits, mientras que una máquina de 64 bits tiene registros de 64 bits e instrucciones para transferir, sumar, restar y manipular de otras maneras palabras de 64 bits.

2.2.3 Ordenamiento de bytes

Los bytes de una palabra pueden numerarse de izquierda a derecha o de derecha a izquierda. A primera vista podría parecer que esta decisión carece de importancia pero, como veremos en breve, tiene implicaciones importantes. En la figura 2-11(a) se muestra parte de la memoria de una computadora de 32 bits cuyos bytes se numeran de izquierda a derecha, como la SPARC o las grandes mainframes de IBM. En la figura 2-11(b) se aprecia la representación análoga de una computadora de 32 bits que usa numeración de derecha a izquierda, como la

Computadora	Bits/celda
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Figura 2-10. Número de bits por celda en algunas computadoras comerciales de importancia histórica.

familia Intel. El primer sistema, en el que la numeración comienza por el extremo "grande" (es decir, de orden alto), se llama computadora **big endian**, en contraste con la **little endian** de la figura 2-11(b). Estos términos provienen de la obra *Los viajes de Gulliver*, donde Jonathan Swift hacía una sátira de los políticos que declararon una guerra a causa de su disputa respecto a si los huevos debían romperse por el extremo ancho (*big end*, en la obra original en inglés) o por el extremo angosto (*little end*). Los términos se usaron por primera vez aplicados a la arquitectura de computadoras en un delicioso artículo escrito por Cohen (1981).

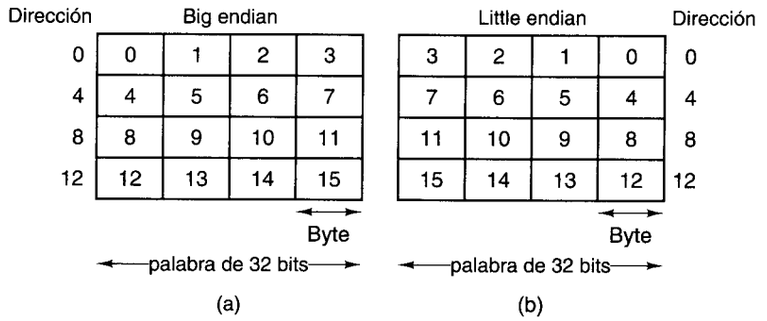


Figura 2-11. (a) Memoria big endian. (b) Memoria little endian.

Es importante entender que tanto en los sistemas big endian como en los little endian, un entero de 32 bits con el valor numérico de, digamos, 6, se representa con los bits 110 en los tres bits de la extrema derecha (de orden bajo) de una palabra, con ceros en los 29 bits de la izquierda. En el esquema big endian, los bits 110 están en el byte 3 (o 7, u 11, etc.), mientras

que en el esquema little endian están en el byte 0 (o 4, u 8, etc.). En ambos casos, la palabra que contiene este entero tiene la dirección 0.

Si las computadoras sólo almacenaran enteros, no habría problemas, pero muchas aplicaciones requieren una mezcla de enteros, cadenas de caracteres y otros tipos de datos. Considere, por ejemplo, un sencillo registro de personal que consiste en una cadena (nombre de empleado) y dos enteros (edad y número de departamento). La cadena termina con uno o más bytes 0 para rellenar una palabra. La representación big endian para Jim Smith, de 21 años, del departamento 260 ($1 \times 256 + 4 = 260$) se muestra en la figura 2.12(a); la little endian, en la figura 2-12(b).

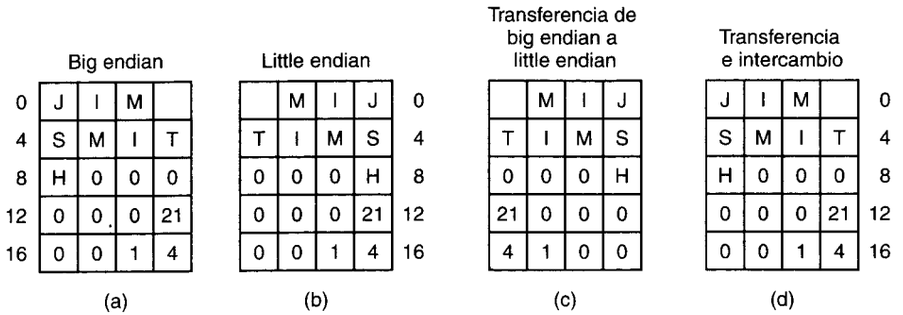


Figura 2-12. (a) Registro de personal en una máquina big endian. (b) El mismo registro en una máquina little endian. (c) Resultado de transferir el regitro de una big endian a una little endian. (d) Resultado de intercambiar los bytes de (c).

Ambas representaciones son excelentes y son congruentes internamente. Los problemas comienzan cuando una de las máquinas trata de enviar el registro a la otra por una red. Supongamos que la big endian envía el registro a la little endian byte por byte, comenzando con el byte 0 y terminando con el byte 19. (Seremos optimistas y supondremos que los bits de los bytes no se invierten durante la transmisión, pues ya tenemos suficientes problemas.) Así, el byte 0 de la big endian se coloca en el byte 0 de la memoria de la little endian, y así sucesivamente, como se muestra en la figura 2-12(c).

Cuando la little endian trata de imprimir el nombre, no hay problema, pero la edad aparece como 21×2^{24}, y el departamento es igualmente absurdo. Esta situación surge porque la transmisión invirtió el orden de los caracteres de una palabra, como debería, pero también invirtió los bytes de los enteros, cosa que no debió hacer.

Una solución obvia es hacer que el software invierta los bytes de una palabra después de efectuar el copiado. Esto conduce a la figura 2-12(d) y hace que los dos enteros queden bien pero convierte la cadena en " MIJTIMS" "H". Esta inversión de la cadena ocurre porque al leerla, la computadora primero lee el byte 0 (un espacio), luego el byte 1 (M), etcétera.

No existe una solución sencilla. Un remedio que funciona, pero es ineficiente, es incluir una cabecera frente a cada dato que indique el tipo de datos que sigue (cadena, entero u otro) y qué longitud tiene. Esto permite al receptor efectuar sólo las conversiones que sean necesarias. En todo caso, es evidente que la falta de un estándar para el ordenamiento de los bytes genera problemas innecesarios al intercamiar datos entre diferentes máquinas.

2.2.4 Códigos para corrección de errores

Las memorias de las computadoras pueden cometer errores ocasionales a causa de picos de voltaje en la línea de alimentación u otras causas. Para protegerse contra tales errores, algunas memorias emplean códigos para detección o corrección de errores. Cuando se usan esos códigos, se añaden bits extra de una forma especial a cada palabra de la memoria. Cuando se lee una palabra de la memoria, se verifican los bits adicionales para ver si ha ocurrido algún error.

Para entender cómo pueden manejarse los errores, es necesario examinar de cerca la naturaleza de los mismos. Supongamos que una palabra de memoria consiste en m bits de datos a los cuales añadiremos r bits redundantes, o de verificación. Sea la longitud total n (es decir, $n = m + r$). Una unidad de n bits que contiene m bits de datos y r bits de verificación se conoce como **palabra de código** de n bits.

Dadas dos palabras de código cualesquiera, digamos 10001001 y 10110001, es posible determinar cuántos bits correspondientes difieren. En este caso, tres bits difieren. Para determinar cuántos bits difieren basta calcular el OR EXCLUSIVO booleano bit por bit de las dos palabras de código, y contar el número de bits 1 en el resultado. El número de posiciones de bit en las que dos palabras de código difieren se denomina **distancia de Hamming** (Hamming, 1950). La importancia de esta cifra es que si dos palabras de código están separadas por una distancia de Hamming d, se requieren d errores de un solo bit para convertir una palabra en la otra. Por ejemplo, las palabras de código 11110001 y 00110000 están separadas por una distancia de Hamming de 3 porque se requieren tres errores de un solo bit para convertir una en la otra.

En el caso de una palabra de memoria de m bits, están permitidos los 2^m patrones de bits, pero a causa de la forma en que se calculan los bits de verificación, sólo 2^m de las 2^n palabras de código son válidas. Si al leer la memoria se obtiene una palabra de código no válida, la computadora sabrá que ocurrió un error de memoria. Si se conoce el algoritmo para calcular los bits de verificación, es posible construir una lista completa de las palabras de código permitidas, y encontrar en esta lista las dos palabras de código cuya distancia de Hamming es mínima. Esta distancia es la distancia de Hamming de todo el código.

Las propiedades de detección y corrección de errores de un código dependen de su distancia de Hamming. Para detectar d errores de un solo bit se necesita un código con una distancia de $d + 1$, porque con un código así no es posible que d errores de un solo bit conviertan una palabra de código válida en otra palabra de código válida. De forma similar, para corregir d errores de un solo bit se requiere un código con una distancia $2d + 1$ porque así las palabras de código válidas están tan lejanas unas de otras que aun con d cambios la palabra de código original está más cerca de la palabra con errores que cualquier otra palabra de código válida, y es posible determinarla de forma inequívoca.

Como ejemplo sencillo de código con detección de errores, considere un código en el que se anexa un solo **bit de paridad** a los datos. El bit de paridad se escoge de modo que el número de bits 1 en la palabra de código sea siempre par (o impar). Un código así tiene una distancia de 2, ya que cualquier error de un solo bit produce una palabra de código con la paridad equivocada. En otras palabras, se requieren dos errores de un solo bit para pasar de una a otra palabras de código válidas, y el código puede servir para detectar errores sencillos.

Siempre que se lee de la memoria una palabra con la paridad incorrecta, se indica una condición de error. El programa no puede continuar, pero al menos no se calculan resultados incorrectos.

Como ejemplo sencillo de código con corrección de errores, considere un código que sólo tiene cuatro palabras de código válidas:

0000000000, 0000011111, 1111100000 y 1111111111

Este código tiene una distancia de 5, lo que implica que puede corregir errores dobles. Si llega la palabra de código 0000000111, el receptor sabe que la palabra original tiene que haber sido 0000011111 (si no hubo más de dos errores). Por otra parte, si un triple error convierte 0000000000 en 0000000111, el error no podrá corregirse.

Imagine que quiere diseñar un código con m bits de datos y r bits de verificación que permita corregir todos los errores de un solo bit. Cada una de las 2^m palabras de memoria válidas tiene n palabras de código no válidas a una distancia de 1. Éstas se forman invirtiendo por turno cada uno de los n bits de la palabra de código de n bits que se forma a partir de la palabra de memoria válida. Así, cada una de las 2^m palabras de memoria válidas requiere $n + 1$ patrones de bits dedicados a ella (para los n posibles errores y el patrón correcto). Puesto que el número total de patrones de bits es 2^n, necesitaremos que $(n + 1)2^m \leq 2^n$. Si usamos $n = m + r$, este requisito se convierte en $(m + r + 1) \leq 2^r$. Dado m, esto fija un límite inferior para el número de bits de verificación que se requieren para corregir errores sencillos. En la figura 2-13 se muestra el número de bits de verificación necesarios para diversos tamaños de palabra de memoria.

Tamaño de palabra	Bits de	Tamaño total	Porcentaje de gasto e xtra verificación
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Figura 2-13. Número de bits de verificación para un código que puede corregir un solo error.

Este límite inferior teórico puede lograrse empleando un método ideado por Richard Hamming (1950). Antes de examinar el algoritmo de Hamming, veamos una sencilla representación gráfica que ilustra claramente la idea de un código de corrección de errores para palabras de 4 bits. El diagrama de Venn de la figura 2-14(a) contiene tres círculos, *A*, *B* y *C*, que juntos forman siete regiones. Como ejemplo, codifiquemos la palabra de memoria de 4 bits 1100 en las regiones *AB*, *ABC*, *AC* y *BC*, un bit por región (en orden alfabético). Esta codificación se muestra en la figura 2-14(a).

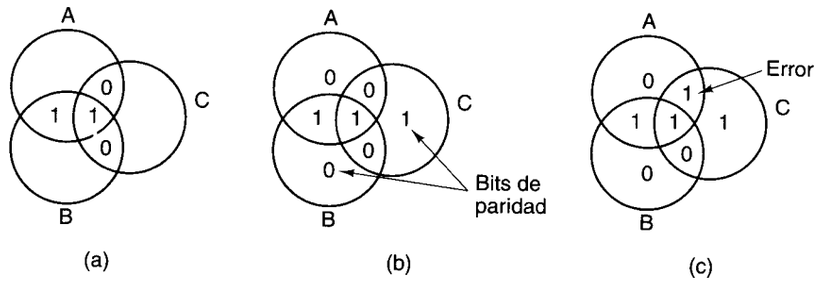


Figura 2-14. (a) Codificación de 1100. (b) Adición de paridad par. (c) Error en *AC*.

Ahora añadimos un bit de paridad a cada una de las tres regiones vacías a modo de producir una paridad par, como se ilustra en la figura 2-14(b). Por definición, la suma de los bits de cada uno de los tres círculos, *A*, *B* y *C*, es ahora un número par. En el círculo *A* tenemos los cuatro números 0, 0, 1 y 1, que suman 2, un número par. En el círculo *B*, los números son 1, 1, 0 y 0, que también suman 2, un número par. Por último, en el círculo *C* tenemos la misma situación. En este ejemplo da la casualidad que todos los círculos son iguales, pero en otros ejemplos podría haber también sumas de 0 y 4. Esta figura corresponde a una palabra de código con 4 bits de datos y 3 bits de paridad.

Supongamos ahora que el bit de la región *AC* es erróneo y cambia de 0 a 1, como se muestra en la figura 2-14(c). La computadora detecta que los círculos *A* y *C* tienen la paridad equivocada (impar). El único cambio de un solo bit que corrige la situación es restaurar *AC* a 0, lo que corrige el error. De este modo, la computadora puede reparar errores de memoria de un solo bit automáticamente.

Veamos ahora cómo podemos usar el algoritmo de Hamming para construir códigos con corrección de errores para palabras de memoria de cualquier tamaño. En un código de Hamming, se añaden r bits de paridad a una palabra de m bits para formar una nueva palabra de $m + r$ bits. Los bits se numeran a partir de 1, no de 0, siendo el bit 1 el de la extrema izquierda (bit de orden alto). Todos los bits cuyo número es una potencia de 2 son bits de paridad; los demás se utilizan para datos. Por ejemplo, con palabras de 16 bits se añaden cinco bits de paridad. Los bits 1, 2, 4, 8 y 16 son bits de paridad, y todos los demás son bits de datos. En total, la palabra de memoria tiene 21 bits (16 de datos, 5 de paridad). Usaremos (arbitrariamente) paridad par en este ejemplo.

Cada bit de paridad examina posiciones de bit específicas; el bit de paridad se ajusta de modo que el número total de unos en las posiciones verificadas sea par. Las posiciones de bit que verifica cada bit de paridad son

Bit 1 verifica los bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.

Bit 2 verifica los bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.

Bit 4 verifica los bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.

Bit 8 verifica los bits 8, 9, 10, 11, 12, 13, 14, 15.

Bit 16 verifica los bits 16, 17, 18, 19, 20, 21.

En general, el bit b es verificado por los bits $b_1, b_2, ..., b_j$ tales que $b_1 + b_2 + ... + b_j = b$. Por ejemplo, el bit 5 es verificado por los bits 1 y 4 porque 1 + 4 = 5. El bit 6 es verificado por los bits 2 y 4 porque 2 + 4 = 6, etcétera.

La figura 2-15 muestra la construcción de un código de Hamming para la palabra de memoria de 16 bits 1111000010101110. La palabra de código de 21 bits es 001011100000101101110. Para ver cómo funciona la corrección de errores, consideremos qué sucedería si el bit 5 se invirtiera por un pico en la alimentación eléctrica. La nueva palabra de código sería 001001100000101101110 en lugar de 001011100000101101110. Verificamos los cinco bits de paridad, con los siguientes resultados:

Paridad del bit 1 incorrecta (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 contienen cinco unos).

Paridad del bit 2 correcta (2, 3, 6, 7, 10, 11, 14, 15, 18, 19 contienen seis unos).

Paridad del bit 4 incorrecta (4, 5, 6, 7, 12, 13, 14, 15, 20, 21 contienen cinco unos).

Paridad del bit 8 correcta (8, 9, 10, 11, 12, 13, 14, 15 contienen dos unos).

Paridad del bit 16 correcta (16, 17, 18, 19, 20, 21 contienen cuatro unos).

El número total de unos de los bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 y 21 debería ser par, porque estamos usando paridad par. El bit incorrecto debe ser uno de los bits verificados por el bit de paridad 1: el bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 o 21. El bit de paridad 4 es incorrecto, lo que implica que uno de los bits 4, 5, 6, 7, 12, 13, 14, 15, 20 o 21 es incorrecto. El error debe estar en uno de los bits de ambas listas, o sea 5, 7, 13, 15 o 21. Sin embargo, el bit 2 es correcto, lo que elimina al 7 y al 15. Así mismo, el bit 8 es correcto, lo que elimina al 13. Por último, el bit 16 es correcto, lo que elimina al 21. El único bit que queda es el 5, que es el que tuvo el error. Puesto que se leyó como 1, debería ser 0. Es así como se corrigen los errores.

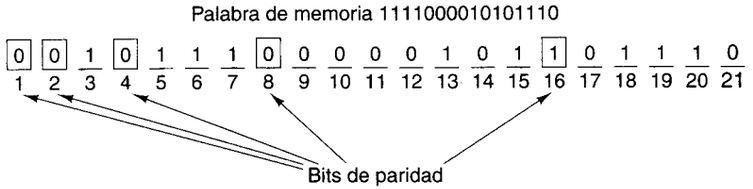


Figura 2-15. Construcción del código Hamming para la palabra de memoria 1111000010101110 añadiendo cinco bits de verificación a los 16 bits de datos.

Un método sencillo para encontrar el bit incorrecto consiste en calcular primero todos los bits de paridad. Si todos son correctos, no hubo error (o hubo más de uno). Luego se suman todos los bits de paridad incorrectos, contando 1 por el bit 1, 2 por el bit 2, 4 por el bit 4, etc. La suma que se obtiene es la posición del bit incorrecto. Por ejemplo, si los bits de paridad 1 y 4 son incorrectos pero los 2, 8 y 16 son correctos, el bit 5 (1 + 4) se habrá invertido.

2.2.5 Memoria caché

Históricamente, las CPU siempre han sido más rápidas que las memorias. Al mejorar las memorias, también han mejorado las CPU, y la diferencia ha persistido. De hecho, al aumentar el número de circuitos que es posible incluir en un chip, los diseñadores de CPU han aprovechado esos recursos para crear filas de procesamiento y operaciones superescalares, lo que hace a las CPU aún más rápidas. Los diseñadores de memorias generalmente han utilizado la tecnología nueva para aumentar la capacidad de sus chips, no la velocidad, así que al parecer el problema está empeorando con el paso del tiempo. Lo que esta diferencia implica en la práctica es que, después de que la CPU emite una solicitud a la memoria, pasan muchos ciclos antes de que reciba la palabra que necesita. Cuanto más lenta es la memoria, más ciclos tiene que esperar la CPU.

Como señalamos antes, hay dos formas de resolver este problema. La más sencilla es iniciar las instrucciones de lectura de memoria (READ) cuando se llega a ellas, pero continuar la ejecución y detener la CPU si una instrucción trata de usar la palabra de memoria antes de que llegue. Cuanto más lenta sea la memoria, con mayor frecuencia ocurrirá este problema y mayor será el retraso cuando ocurra. Por ejemplo, si el retraso de memoria es de 10 ciclos, es muy probable que una de las siguientes 10 instrucciones trate de usar la palabra que se leyó.

La otra solución es tener máquinas que no se detengan, y que exijan a los compiladores no generar código que use palabras antes de que hayan llegado. El problema es que no es nada fácil llevar esto a la práctica. Es común que después de un LOAD no haya nada más que hacer, y el compilador se vea obligado a insertar instrucciones NOP (no hay operación) que no hacen nada más que ocupar una ranura y desperdiciar tiempo. En realidad, esto equivale a detenerse, sólo que en software en lugar de en hardware, pero la degradación del desempeño es la misma.

De hecho, el problema no es la tecnología, sino la economía. Los ingenieros saben cómo construir memorias que sean tan rápidas como las CPU, pero para operar a su máxima velocidad tienen que estar dentro del chip de la CPU (porque ir por el bus hasta la memoria es muy lento). La colocación de una memoria grande en el chip de la CPU lo hace más grande y por tanto más costoso, pero aun si el costo no fuera importante existen límites para el tamaño que un chip de CPU puede tener. Así, la decisión se reduce a tener una cantidad pequeña de memoria rápida o una cantidad grande de memoria lenta. Lo que preferiríamos es una gran cantidad de memoria rápida a un precio bajo.

Lo interesante es que se conocen técnicas para combinar una cantidad pequeña de memoria rápida con una cantidad grande de memoria lenta para obtener la velocidad de la memoria rápida (casi) y la capacidad de la memoria grande a un precio moderado. La memoria pequeña y rápida se llama **caché** (del francés *cacher*, que significa guardar o esconder). A continuación describiremos brevemente cómo se usan los cachés y como funcionan. En el capítulo 4 se dará una explicación más detallada.

La idea fundamental en que se basa la memoria caché es sencilla: las palabras de memoria de mayor uso se mantienen en el caché. Cuando la CPU necesita una palabra, primero la busca en el caché. Sólo si la palabra no está ahí recurre a la memoria principal. Si una fracción sustancial de las palabras está en el caché, el tiempo de acceso promedio puede reducirse considerablemente.

Así pues, el éxito o el fracaso dependen de qué fracción de las palabras está en el caché. Desde hace años, se sabe que los programas no acceden a la memoria de forma totalmente aleatoria. Si una referencia a la memoria dada es a la dirección A, es probable que la siguiente referencia sea a una dirección cercana a A. Un ejemplo sencillo es el programa mismo. Con excepción de las ramificaciones y las llamadas a procedimientos, los instrucciones se traen de posiciones consecutivas de la memoria. Además, casi todo el tiempo de ejecución de los programas se invierte en realizar ciclos (*loops*), en los que un número limitado de instrucciones se ejecutan una y otra vez. Así mismo, un programa que manipula matrices probablemente hará muchas referencias a la misma matriz antes de pasar a otra cosa.

La observación de que las referencias a la memoria que se hacen en cualquier periodo corto tienden a usar una fracción pequeña de la memoria total se denomina **principio de localidad** y constituye la base de todos los sistemas de caché. La idea general es que cuando se hace referencia a una palabra, ella y algunas de sus vecinas se traen de la memoria grande lenta al caché, para que la siguiente vez que se use el acceso a ella sea rápido. Un acomodo común de la CPU, el caché y la memoria principal se ilustra en la figura 2-16. Si una palabra se lee o se escribe k veces en un intervalo corto, la computadora necesitará una referencia a la memoria lenta y $k - 1$ referencias a la memoria rápida. Cuanto mayor sea k, mejor será el desempeño global.

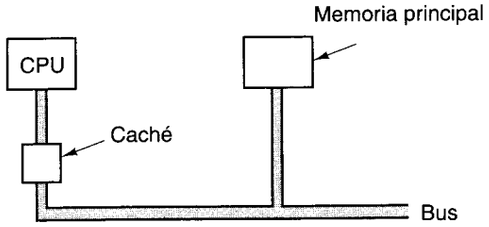


Figura 2-16. El caché se encuentra lógicamente entre la CPU y la memoria principal. Físicamente, hay varios lugares donde podría colocarse.

Podemos formalizar este cálculo introduciendo c, el tiempo de acceso al caché, m, el tiempo de acceso a la memoria principal y h, la **tasa de aciertos**, que es la fracción de todas las referencias que pueden satisfacerse con el caché. En nuestro pequeño ejemplo del párrafo anterior, $h = (k - 1)/k$. Algunos autores también definen la **tasa de fallos**, que es $1 - h$.

Con estas definiciones, podemos calcular el tiempo de acceso medio así:

$$\text{tiempo de acceso medio} = c + (1 - h)m$$

A medida que $h \rightarrow 1$, todas las referencias pueden satisfacerse con el caché, y el tiempo de acceso se aproxima a c. Por otro lado, si $h \rightarrow 0$, se requerirá una referencia a la memoria en cada ocasión, y el tiempo de acceso se aproximará a $c + m$: primero un tiempo c para examinar el caché (sin éxito) y luego un tiempo m para efectuar la referencia a la memoria. En algunos sistemas, la referencia a la memoria puede iniciarse en paralelo con la búsqueda en el caché, de modo que si no hay acierto en el caché, el ciclo de memoria ya se habrá iniciado.

Sin embargo, esta estrategia requiere que la búsqueda en la memoria pueda suspenderse si hay un acierto en el caché, y esto complica la implementación.

Con el principio de localidad como guía, las memorias principales y los cachés se dividen en bloques de tamaño fijo. Al hablar de estos bloques dentro del caché, es común usar el término **líneas de caché**. Cuando hay un fallo de caché, toda la línea de caché se carga de la memoria principal al caché, no sólo la palabra que se necesita. Por ejemplo, si las líneas son de 64 bytes, una referencia a la dirección de memoria 260 traerá la línea que consiste en los bytes 256 a 319 a una línea de caché. Con un poco de suerte, pronto se necesitarán también algunas de las otras palabras de esa línea de caché. Operar así es más eficiente que buscar palabras individuales porque es más rápido traer k palabras a la vez que traer una palabra k veces. Además, si las entradas del caché son más grandes que una palabra habrá menos de ellas, y el gasto extra para manejarlas será menor.

El diseño de cachés es un tema cada vez más importante para el diseño de las CPU de alto rendimiento. Un aspecto es el tamaño del caché. Cuanto más grande es el caché, mejor funciona, pero también cuesta más. Un segundo aspecto es el tamaño de la línea de caché. Un caché de 16 KB puede dividirse en 1 K líneas de 16 bytes, 2 K líneas de 8 bytes, y otras combinaciones. Un tercer aspecto es la organización del caché, es decir, ¿cómo hace el caché para saber qué palabras de memoria contiene en un momento dado? Examinaremos los cachés con detalle en el capítulo 4.

Un cuarto aspecto de diseño es si las instrucciones y los datos se guardan en el mismo caché o en cachés distintos. Un diseño de **caché unificado** (instrucciones y datos en el mismo caché) es más sencillo y automáticamente equilibra las obtenciones de instrucciones frente a las obtenciones de datos. No obstante, la tendencia actual es hacia los **cachés divididos**, con instrucciones en un caché y datos en otro. Este diseño también se denomina **arquitectura Harvard**, referencia que se remonta a la computadora Mark III de Howard Aiken, que tenía diferentes memorias para las instrucciones y los datos. Lo que está impulsando a los diseñadores en esta dirección es el uso tan extendido de las CPU con filas de procesamiento. La unidad de búsqueda de instrucciones necesita acceder a instrucciones al mismo tiempo que la unidad de búsqueda de operandos requiere acceder a datos. Un caché dividido permite accesos en paralelo; un caché unificado no. Además, puesto que las instrucciones normalmente no se modifican durante la ejecución, el contenido del caché de instrucciones nunca tiene que volver a escribirse en la memoria.

Por último, un quinto aspecto es el número de cachés. Hoy día no es raro tener chips con un caché primario en el chip de CPU, un caché secundario fuera del chip pero en el mismo paquete, y un tercer caché todavía más lejos.

2.2.6 Empaquetamiento y tipos de memoria

Desde los albores de las memorias de semiconductores hasta principios de los noventa, la memoria se fabricaba, se compraba y se instalaba como chips individuales. Las densidades de los chips iban desde 1K bits hasta 1M bits o más, pero cada chip era una unidad discreta. Las primeras PC a menudo tenían zócalos vacíos en los que podían insertarse chips de memoria adicionales, si el comprador llegaba a necesitarlos.

Actualmente se acostumbra usar una disposición diferente. Un grupo de chips, por lo regular 8 o 16, se monta en una diminuta tarjeta de circuitos integrados y se vende como unidad. Esta unidad se llama **SIMM (módulo de memoria individual en línea**, *Single Inline Memory Module*) o **DIMM (módulo de memoria dual en línea**, *Dual Inline Memory Module*), dependiendo de si tiene una fila de conectores en un lado o en ambos lados de la tarjeta. En la figura 2-17 se ilustra un ejemplo de SIMM.

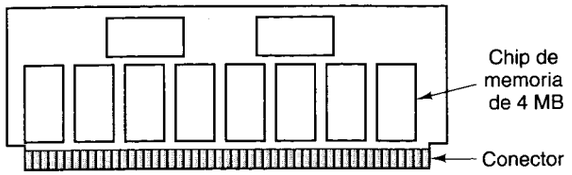


Figura 2-17. Módulo individual de memoria en línea (SIMM) que contiene 32 MB. Dos de los chips controlan el SIMM.

Una configuración de SIMM típica podría tener ocho chips con 32 megabits (4 MB) cada uno en el SIMM. Así, todo el módulo contendría 32 MB. Muchas computadoras tienen espacio para cuatro módulos, lo que da una capacidad total de 128 MB si son empleados SIMM de 32 MB. En muchos casos estos SIMM pueden sustituirse posteriormente por SIMM de 64 MB o más grandes si es necesario.

Los primeros SIMM tenían 30 conectores y alimentaban 8 bits a la vez. Los demás conectores eran para direccionamiento y control. SIMM posteriores tenían 72 conectores y alimentaban 32 bits a la vez. Para una máquina como la Pentium, que esperaba 64 bits a la vez, los SIMM de 72 conectores se apareaban, y cada uno alimentaba la mitad de los bits requeridos. Actualmente los DIMM son la forma estándar de empaquetar la memoria. Cada DIMM tiene 84 conectores chapeados con oro en cada lado, para un total de 168 conectores. Un DIMM puede alimentar 64 bits a la vez. Las capacidades típicas de los DIMM son de 64 MB o más. Un DIMM físicamente más pequeño, llamado **SO-DIMM (DIMM de contorno pequeño**, *Small Outline DIMM*), se usa en computadoras portátiles. Es posible añadir a los SIMM y DIMM un bit de paridad o corrección de errores, pero puesto que la tasa media de errores de un módulo es de un error cada 10 años, en casi todas las computadoras ordinarias se ha omitido la detección y corrección de errores.

2.3 MEMORIA SECUNDARIA

Por más grande que sea la memoria principal, siempre es demasiado pequeña. La gente siempre quiere almacenar más información de la que cabe en la memoria, primordialmente porque, a medida que la tecnología mejora, comienzan a pensar en almacenar cosas que antes sólo se les ocurrían a los escritores de ciencia-ficción. Por ejemplo, a medida que la disciplina presupuestaria del gobierno de Estados Unidos obliga a sus dependencias a generar sus propias ganancias, podríamos imaginar que la Biblioteca del Congreso decidiera digitalizar y vender todo su acervo como artículo de consumo ("Todo el Conocimiento Humano por sólo

49 dólares"). A muy grandes rasgos, serían unos 50 millones de libros, cada uno con 1 MB de texto y 1 MB de ilustraciones comprimidas, y necesitaríamos almacenar 10^{14} bytes = 100 terabytes. Almacenar las casi 50,000 películas que se han filmado es una tarea de magnitud comparable. Esta cantidad de información no va a caber en la memoria principal, al menos no en las próximas décadas.

2.3.1 Jerarquías de memoria

La solución tradicional para almacenar una gran cantidad de datos es una jerarquía de memoria, como se ilustra en la figura 2-18. En la cúspide están los registros de la CPU, a los que puede tenerse acceso a la velocidad máxima de la CPU. Luego viene la memoria caché, que actualmente es del orden de 32 KB a unos cuantos megabytes. Sigue la memoria principal, con tamaños que actualmente van de 16 MB para los sistemas más económicos hasta decenas de gigabytes en el extremo superior. Después vienen los discos magnéticos, el "caballito de batalla" actual para el almacenamiento permanente. Por último, tenemos la cinta magnética y los discos ópticos para el almacenamiento de archivos.

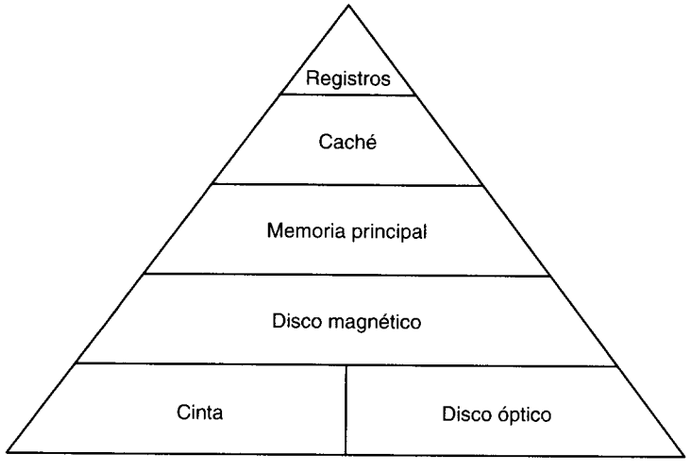


Figura 2-18. Jerarquía de memoria de cinco niveles.

Al bajar por la jerarquía, tres parámetros clave crecen. Primero, el tiempo de acceso se alarga. Los registros de la CPU pueden accederse en unos cuantos nanosegundos. Las memorias caché requieren un múltiplo pequeño del tiempo de acceso de los registros. Los accesos a la memoria principal suelen ser de unas cuantas decenas de nanosegundos. Luego viene una brecha grande, pues los tiempos de acceso a disco son de por lo menos 10 ms, y el acceso a cinta o disco óptico puede medirse en segundos si es preciso traer los medios e insertarlos en una unidad.

Segundo, la capacidad de almacenamiento aumenta al bajar por la jerarquía. Los registros de la CPU pueden contener tal vez 128 bytes; los cachés unos cuantos megabytes; las memorias principales decenas o miles de megabytes; los discos magnéticos de unos cuantos

gigabytes a decenas de gigabytes. Las cintas y los discos ópticos generalmente se guardan fuera de línea, así que su capacidad está limitada sólo por el presupuesto del propietario.

Tercero, el número de bits que se obtiene por dólar invertido aumenta al bajar por la jerarquía. Aunque los precios actuales cambian rápidamente, la memoria principal se mide en dólares/megabyte; el almacenamiento en disco magnético en centavos de dólar/megabyte, y la cinta magnética en dólares/gigabyte o menos.

Ya examinamos los registros, el caché y la memoria principal. En las secciones que siguen estudiaremos los discos magnéticos; después nos ocuparemos de los discos ópticos. No estudiaremos las cintas porque se usan casi exclusivamente para copias de seguridad, y de todos modos no hay mucho que hablar al respecto.

2.3.2 Discos magnéticos

Un disco magnético consiste en uno o más platos de aluminio con un recubrimiento magnetizable. Originalmente estos platos llegaron a tener 50 cm de diámetro, pero en la actualidad suelen ser de 3 a 12 cm, y algunos discos para computadoras portátiles tienen diámetros de menos de 3 cm y se siguen encogiendo. Una cabeza de disco que contiene una bobina de inducción flota sobre la superficie y muy cerca de ella, descansando en un colchón de aire (excepto en los discos flexibles, en los que toca la superficie). Cuando una corriente positiva o negativa pasa por la cabeza, magnetiza la superficie justo debajo de ella, alineando las partículas magnéticas hacia la izquierda o hacia la derecha, dependiendo de la polaridad de la corriente de la unidad. Cuando la cabeza pasa sobre un área magnetizada, se induce una corriente positiva o negativa en la cabeza, lo que permite leer los bits almacenados previamente. Así, el plato gira bajo la cabeza, puede escribirse una serie de bits y luego leerse. La geometría de una pista de disco se muestra en la figura 2-19.

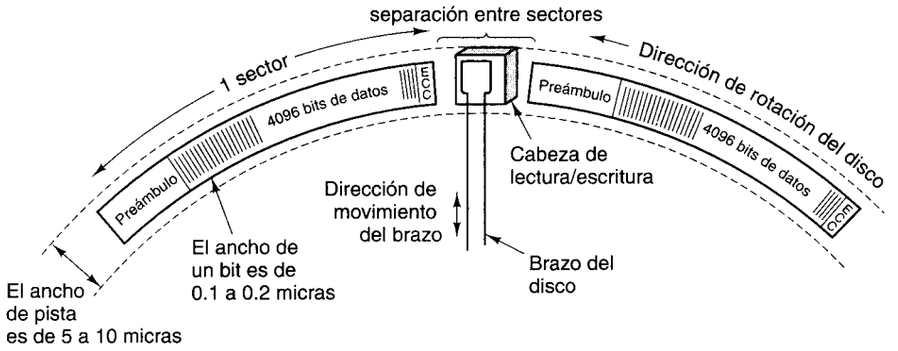


Figura 2-19. Porción de una pista de disco. Se ilustran dos sectores.

La secuencia circular de bits que se escriben cuando el disco efectúa una rotación completa se llama **pista**. Cada pista se divide en **sectores** de longitud fija, que típicamente contienen 512 bytes de datos, precedidos por un **preámbulo** que permite a la cabeza sincronizarse antes de leer o escribir. Después de los datos viene un código de corrección de errores (ECC, *Error-Correcting Code*), que puede ser un código de Hamming o, más comúnmente, un código capaz de corregir múltiples errores llamado **código Reed-Solomon**. Entre sectores consecutivos hay una pequeña **separación entre sectores**. Algunos fabricantes citan la capacidad de sus discos en un estado sin formato (como si cada pista contuviera únicamente datos), pero una medición más honesta es la capacidad formateada, que no cuenta los preámbulos, ECC ni separaciones como datos. La capacidad formateada suele ser 15% menor que la capacidad sin formato.

Todos los discos tienen brazos móviles que pueden desplazarse hacia adentro y hacia afuera hasta diferentes distancias radiales del eje sobre el que gira el plato. A cada distancia radial puede escribirse una pista diferente. Así, las pistas son una serie de círculos concéntricos alrededor del eje. La anchura de una pista depende del tamaño de la cabeza y de la exactitud con que ésta puede ubicarse radialmente. Con la tecnología actual, los discos tienen entre 800 y 2000 pistas por centímetro, lo que da anchos de pista del orden de 5 a 10 micras (1 micra = 1/1000 mm). Cabe señalar que una pista no es un surco físico en la superficie, sino sólo un anillo de material magnetizado, con delgadas áreas protectoras que lo separan de las pistas que están más adentro y más afuera.

La densidad lineal de bits alrededor de la circunferencia de la pista es diferente de la radial, y en buena medida está determinada por la pureza de la superficie y la calidad del aire. Los discos actuales alcanzan densidades de 50,000 a 100,000 bits/cm. Por tanto, un bit es unas 50 veces más grande en la dirección radial que a lo largo de la circunferencia. A fin de elevar la calidad de la superficie y del aire, casi todos los discos se sellan en la fábrica para impedir la entrada de polvo. Tales unidades se llaman **discos Winchester**. Las primeras unidades (creadas por IBM) tenían 30 MB de almacenamiento fijo sellado y 30 MB de almacenamiento removible. Supuestamente, estos discos 30-30 recordaban a la gente los rifles Winchester 30-30 que tan importante papel desempeñaron en la frontera de Estados Unidos, por lo que se conservó el nombre "Winchester".

Casi todos los discos consisten en varios platos apilados verticalmente, como se muestra en la figura 2-20. Cada superficie tiene su propio brazo y cabeza. Todos los brazos están soldados juntos de modo que se mueven simultáneamente a una posición radial dada. El conjunto de todas las pistas en una posición radial dada se llama **cilindro**.

El desempeño de los discos depende de diversos factores. Para leer o escribir un sector, primero el brazo debe moverse a la posición radial correcta. Esta acción se denomina **búsqueda** (*seek*). Los tiempos de búsqueda promedio (entre pistas al azar) son del orden de 5 a 15 ms, aunque se ha logrado bajar el tiempo de búsqueda entre pistas consecutivas a menos de 1 ms. Una vez que la cabeza está ubicada radialmente, hay un retraso, llamado **latencia rotacional** en lo que el sector deseado gira hasta quedar bajo la cabeza. Casi todos los discos giran a 3600, 5400 o 7200 rpm, así que el retardo medio (media rotación) es de 4 a 8 ms. En el extremo superior hay discos que giran a 10,800 rpm (o sea, a 180 rotaciones/s). El tiempo de transferencia depende de la densidad lineal y de la velocidad de rotación. Con las tasas de transferencia típicas de 5 a 20 MB/s, un sector de 512 bytes tarda entre 25 y 100 μs. Por consiguiente, el tiempo de búsqueda y la latencia rotacional dominan el tiempo de transferencia. La lectura de sectores al azar dispersos por todo el disco es obviamente una forma poco eficiente de operar.

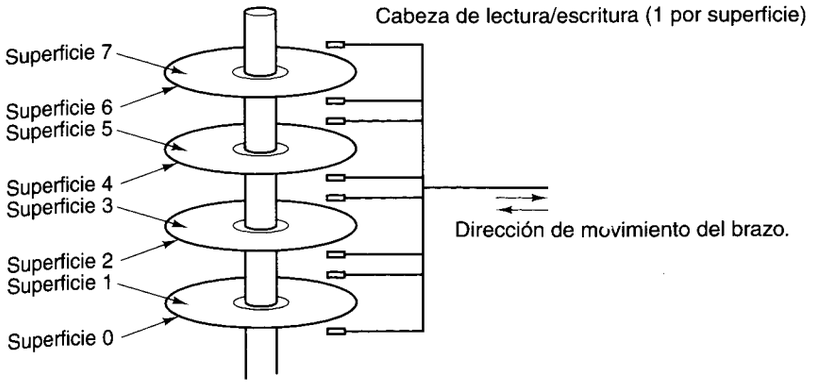


Figura 2-20. Disco con cuatro platos.

Vale la pena mencionar que, a causa de los preámbulos, los ECC, las separaciones entre sectores, los tiempos de búsqueda y las latencias rotacionales, hay una gran diferencia entre la rapidez máxima de ráfaga de una unidad y su rapidez máxima sostenida. La rapidez máxima de ráfaga es la tasa de datos una vez que la cabeza está sobre el primer bit de datos. La computadora debe poder manejar datos que llegan con esta rapidez. Sin embargo, la unidad sólo puede mantener esa rapidez durante un sector. Para algunas aplicaciones, como multimedia, lo que importa es la tasa media sostenida durante un periodo de segundos, y esto debe tener en cuenta también las búsquedas y retrasos rotacionales.

Cuando los discos giran de 60 a 120 revoluciones por segundo se calientan y expanden, lo que altera su geometría física. Algunas unidades necesitan recalibrar sus mecanismos de ubicación periódicamente para compensar esta expansión. Esto lo hacen obligando a las cabezas a salir o meterse totalmente. Tales recalibraciones pueden dar al traste con aplicaciones multimedia, que esperan un flujo más o menos constante de bits a la tasa sostenida máxima. Para manejar aplicaciones multimedia, algunas compañías fabrican **unidades de disco audiovisual** que no requieren estas recalibraciones térmicas.

Si pensamos un poco y usamos la vieja fórmula de primaria para obtener la circunferencia de un círculo, $c = 2\pi r$, veremos que las pistas exteriores comprenden una mayor distancia lineal que las interiores. Puesto que todos los discos magnéticos giran con una velocidad angular constante, sin importar dónde estén las cabezas, este hecho crea un problema. En las unidades viejas, los fabricantes utilizaban la densidad lineal máxima posible en la pista más interior, y densidades lineales de bits cada vez más bajas en las pistas más exteriores. Si un disco tenía 18 sectores por pista, por ejemplo, cada una ocupaba 20 grados de arco, sin importar en qué cilindro estuviera.

Hoy día se sigue una estrategia distinta. Los cilindros se dividen en zonas (por lo regular de 10 a 30 por unidad) y el número de sectores por pista se incrementa en cada zona a medida que ésta se aleja de la pista más interior. Este cambio dificulta seguirle la pista a la información, pero incrementa la capacidad de la unidad, lo cual se considera más importante. Todos los sectores tienen el mismo tamaño. Por fortuna, algunas cosas de la vida nunca cambian.

Cada unidad tiene asociado un **controlador de disco**, un chip que controla la unidad. Algunos controladores contienen una CPU dedicado. Las tareas del controlador incluyen aceptar comandos del software, como READ, WRITE y FORMAT (escribir todos los preámbulos), controlar el movimiento del brazo, detectar y corregir errores, y convertir los bytes de 8 bits que se leen de la memoria en un flujo de bits en serie y viceversa. Algunos controladores también colocan en buffers sectores enteros, colocan en caché sectores leídos que podrían usarse en el futuro, y mantienen el mapa de sectores defectuosos. Esta última función es necesaria cuando hay sectores que tienen puntos magnetizados de forma permanente. Cuando el controlador descubre un sector defectuoso, lo sustituye por uno de los sectores de repuesto reservados para este fin dentro de cada cilindro o zona.

2.3.3 Discos flexibles

Con la llegada de la computadora personal, se requirió una forma de distribuir software. La solución fue el **disquete** o **disco flexible**, un medio pequeño y removible que recibió ese nombre porque los primeros eran físicamente flexibles. En realidad, IBM inventó el disco flexible para grabar información de mantenimiento acerca de sus mainframes para el personal de servicio, pero los fabricantes de computadoras personales pronto lo adoptaron como una forma cómoda de distribuir y vender software.

Las características generales son las mismas que las de los discos que acabamos de describir, excepto que, a diferencia de los discos duros, en los que las cabezas flotan sobre la superficie en un colchón de aire en rápido movimiento, las cabezas de las unidades de disco sí tocan los disquetes. El resultado es que tanto los medios como las cabezas se desgastan con comparativa rapidez. A fin de reducir el desgaste, las computadoras personales retraen las cabezas y detienen la rotación cuando una unidad no está leyendo o escribiendo. Por consiguiente, cuando se emite la siguiente orden de leer o escribir, hay un retraso de cerca de medio segundo mientras el motor adquiere la velocidad necesaria.

Hay dos tamaños: 5.25 pulgadas y 3.5 pulgadas. Cada uno tiene una versión de baja densidad (LD) y una de alta densidad (HD). Los disquetes de 3.5 pulgadas están protegidos con una cubierta rígida, así que no son realmente "flexibles". Puesto que los disquetes de 3.5" almacenan más datos y están mejor protegidos, prácticamente ya sustituyeron a los de 5.25", más viejos. En la figura 2-21 se muestran los parámetros más importantes de los cuatro tipos.

2.3.4 Discos IDE

Los discos para las computadoras personales modernas evolucionaron a partir del de la IBM PC XT, que era un disco Seagate de 10 MB manejado por un controlador de disco Xebec en una tarjeta insertable. El disco Seagate tenía 4 cabezas, 306 cilindros y 17 sectores/pista. El controlador podía manejar dos unidades. El sistema operativo leía de un disco y escribía en él colocando parámetros en registros de la CPU e invocando después al **BIOS (sistema básico de entrada y salida**, *Basic Input Output System*) situado en la memoria sólo de lectura integrada a la PC. El BIOS emitía las instrucciones de máquina necesarias para cargar los registros del controlador de disco que iniciaban las transferencias.

Parámetros	LD 5.25″	HD 5.25″	LD 3.5″	HD 3.5″
Tamaño (pulgadas)	5.25	5.25	3.5	3.5
Capacidad (bytes)	360K	1.2M	720K	1.44M
Pistas	40	80	80	80
Sectores/pista	9	15	9	18
Cabezas	2	2	2	2
Rotaciones/min	300	360	300	300
Tasa de datos (kbps)	250	500	250	500
Tipo	Flexible	Flexible	Rígido	Rígido

Figura 2-21. Características de los cuatro tipos de discos flexibles.

La tecnología evolucionó rápidamente, de tener el controlador en una tarjeta aparte a integrarlo íntimamente a las unidades de disco, comenzando por las unidades **IDE (circuitos integrados a la unidad**, *Integrated Drive Electronics*) a mediados de los ochenta. Sin embargo, las convenciones de invocación del BIOS no se modificaron por razones de compatibilidad con modelos anteriores. Estas convenciones direccionaban sectores dando sus números de cabeza, cilindro y sector. Las cabezas y cilindros se numeraban a partir del 0, y los sectores, a partir del 1. Esta decisión probablemente se debió a un error del programador original del BIOS, quien escribió su obra maestra en el ensamblador del 8088. Con 4 bits para la cabeza, 6 bits para el sector y 10 bits para el cilindro, la unidad de disco más grande podía tener 16 cabezas, 63 sectores y 1024 cilindros, para un total de 1,032,192 sectores. Una unidad así tiene una capacidad de 528 MB, que probablemente parecía casi infinita en ese entonces pero que ahora podría considerarse pequeña. (¿Criticaría hoy una máquina nueva que no pudiera manejar unidades de más de un terabyte?)

En relativamente poco tiempo aparecieron unidades de más de 528 MB, pero con la geometría equivocada (por ejemplo, 4 cabezas, 32 sectores, 2000 cilindros). El sistema operativo no tenía forma de direccionarlos a causa de las convenciones de invocación de BIOS estáticas. El resultado es que los controladores de disco comenzaron a mentir, fingiendo que la geometría estaba dentro de los límites de BIOS cuando en realidad estaba estableciendo una nueva correspondencia entre la geometría virtual y la geometría real. Aunque esta estrategia funcionó, hizo estragos con los sistemas operativos que colocaban cuidadosamente los datos a modo de minimizar los tiempos de búsqueda.

Con el paso del tiempo, las unidades IDE se convirtieron en unidades **EIDE (IDE extendido, Extended IDE)**, que también reconocen un segundo esquema de direccionamiento llamado **LBA (direccionamiento por bloque lógico**, *Logical Block Addressing*), que simplemente numera los sectores a partir del 0 y hasta un máximo de $2^{24} - 1$. Este esquema requiere que el controlador convierta las direcciones LBA en direcciones de cabeza, sector y

cilindro, pero al menos logra superar el límite de los 528 MB. Las unidades y controladores EIDE tienen también otras mejoras, como la capacidad de controlar cuatro unidades en lugar de dos, una tasa de transferencia más alta y la capacidad para controlar unidades CD-ROM.

Los discos IDE y EIDE eran originalmente sólo para sistemas basados en Intel, ya que la interfaz es una copia exacta del bus de la IBM PC. Sin embargo, hoy día unas cuantas computadoras más los usan por su bajo precio.

2.3.5 Discos SCSI

Los discos SCSI no son diferentes de los IDE en cuanto a la organización de sus cilindros, pistas y sectores, pero tienen una interfaz diferente y tasas de transferencia mucho más altas. La historia de SCSI se remonta a Howard Shugart, el inventor del disco flexible, cuya compañía introdujo el disco SASI (Shugart Associates System Interface) en 1979. Después de algunas modificaciones y muchos alegatos, el ANSI lo estandarizó en 1986 y cambió su nombre a **SCSI (interfaz de sistema de cómputo pequeño**, *Small Computer System Interface*). SCSI se pronuncia "escosi". En 1994, el ANSI emitió un estándar actualizado, SCSI-2, que en buena medida ha sustituido al SCSI-1 original. Todavía se está trabajando en el SCSI-3, aunque partes de la norma ya están terminadas. Algunos fabricantes han implementado estas partes y las usan para decir que cumplen con SCSI-3.

Puesto que los discos SCSI tienen tasas de transferencia altas, son el disco estándar en casi todas las estaciones de trabajo UNIX de Sun, HP, SGI y otros fabricantes; también son el disco estándar de las Macintosh y son comunes en las PC Intel actuales, sobre todo los servidores de red. En la figura 2-22 se dan los parámetros de algunas de las configuraciones más comunes.

Nombre	Bits de datos	Frecuencia en MHz	MB/s
SCSI-1	8	5	5
SCSI-2	8	5	5
SCSI-2 rápido	8	10	10
SCSI-2 rápido y ancho	16	10	20
Ultra SCSI	16	20	40

Figura 2-22. Algunos posibles parámetros de SCSI.

SCSI es algo más que una interfaz de disco duro; es un bus al que pueden conectarse un controlador SCSI y hasta siete dispositivos. Éstos pueden incluir uno o más discos duros SCSI, CD-ROM, grabadoras de CD, escáneres, unidades de cinta y otros periféricos SCSI. Cada dispositivo SCSI tiene un identificador único, de 0 a 7 (15 en el caso de SCSI ancho). Cada dispositivo tiene dos conectores: uno para entrada y otro para salida. La salida de un dispositivo se conecta a la entrada de otro con cables, en serie, como en una serie barata de focos navideños. El último dispositivo en la serie debe terminarse para evitar que las reflexiones de los extremos del bus SCSI interfieran con otros datos que van sobre el bus. Por lo regular el controlador está en una tarjeta insertable que es el principio de la cadena de cables, aunque la norma no exige estrictamente esta configuración.

El cable más común para SCSI de 8 bits tiene 50 hilos, 25 de los cuales son tierras apareadas uno a uno con los otros 25 hilos para proporcionar la excelente inmunidad al ruido que se necesita para el funcionamiento a alta velocidad. De los 25 hilos, ocho son para datos, uno es para paridad, nueve son para control y los demás son para la potencia o se reservan para usos futuros. Los dispositivos de 16 bits (y 32 bits) necesitan un segundo cable para las señales adicionales. Los cables pueden tener varios metros de longitud, lo que permite tener unidades de disco externas, escáneres, etcétera.

Los controladores y periféricos SCSI pueden operar sea como transmisores o bien como receptores. Por lo regular el controlador, actuando como transmisor, emite comandos a discos y otros periféricos que actúan como receptores. Estos comandos son bloques de hasta 16 bytes que le dicen al receptor qué debe hacer. Los comandos y respuestas se dan en fases, utilizando diversas señales de control para delimitar las fases y arbitrar el acceso al bus cuando varios dispositivos están tratando de usar el bus al mismo tiempo. Este arbitraje es importante porque SCSI permite a todos los dispositivos operar al mismo tiempo, lo que podría mejorar considerablemente el desempeño en un entorno en el que varios procesos están activos simultáneamente (por ejemplo, UNIX o Windows NT). IDE y EIDE sólo permiten un dispositivo activo a la vez.

2.3.6 RAID

El desempeño de las CPU se ha aumentado exponencialmente durante la década pasada, duplicándose aproximadamente cada 18 meses. No ha sucedido lo mismo con el desempeño de los discos. En los años setenta los tiempos de búsqueda promedio en discos para minicomputadora eran de 50 a 100 ms. Hoy esos tiempos son de 10 ms. En casi todas las industrias técnicas (digamos la de automóviles o de aviación) un mejoramiento en el desempeño por un factor de 5 a 10 en dos décadas sería una noticia excelente, pero en la industria de las computadoras es vergonzoso. Así, la brecha entre el desempeño de la CPU y el del disco ha aumentado con el tiempo.

Como hemos visto, el procesamiento en paralelo se está utilizando cada vez más para acelerar el funcionamiento de las CPU. A varias personas se les ha ocurrido que la E/S paralela también podría ser una buena idea. En su artículo de 1988, Patterson *et al.* sugirieron seis organizaciones de disco específicas que podrían servir para mejorar el desempeño de los discos, su confiabilidad o ambas cosas (Patterson *et al.*, 1988). La industria adoptó rápidamente estas ideas y han dado pie a una nueva clase de dispositivo de E/S llamado **RAID**. Patterson *et al.* definieron a RAID como una **formación redundante de discos de bajo costo** (*Redundant Array of Inexpensive Disks*), pero la industria redefinió la "I" de modo que significara "independientes" en lugar de "de bajo costo" (¿tal vez así podrían usar discos caros?). Puesto que también se necesitaba un villano (como en RISC *versus* CISC, también debido a Patterson), el malo aquí fue el **SLED (disco único, grande y costoso**, *Single Large Expensive Disk*).

La idea en que se basa RAID es instalar una caja llena de discos junto a la computadora, que por lo regular es un servidor grande, sustituir la tarjeta del controlador de disco por un controlador de RAID, copiar los datos en el RAID, y continuar con las operaciones normales. En otras palabras, el sistema operativo veía el RAID como si fuera un SLED, sólo que el desempeño y la confiabilidad mejoraban. Puesto que los discos SCSI tienen buen desempeño, precio bajo y la capacidad para conectar hasta siete unidades a un solo controlador (15 en el caso de SCSI ancho), es natural que la mayor parte de los RAID conste de un controlador RAID SCSI más una caja de discos SCSI que el sistema operativo ve como un solo disco grande. De este modo, no se requieren cambios al software para usar el RAID, un punto importante para convencer a los administradores de sistemas de comprarlo.

Además de aparentar ser un solo disco ante el software, todos los RAID tienen la propiedad de que los datos se distribuyen entre todas las unidades, a fin de permitir la operación en paralelo. Patterson y otros autores definieron varios esquemas para hacerlo y ahora se conocen como RAID nivel 0 hasta RAID nivel 5. Además, existen algunos niveles menores que no se discutirán. El término "nivel" es un tanto engañoso porque no existe una jerarquía; simplemente hay seis posibles organizaciones distintas.

RAID nivel 0 se ilustra en la figura 2-23(a), y consiste en ver el disco único virtual simulado por el RAID como si estuviera dividido en tiras (*strips*) de k sectores cada una, con los sectores 0 a $k-1$ en la tira 0, los sectores k a $2k-1$ en la tira 1, etc. Para $k = 1$, cada tira es un sector; para $k = 2$, una tira es dos sectores, etc. La organización RAID nivel 0 escribe tiras consecutivas en las unidades por turno circular, como se muestra en la figura 2-23(a) para un RAID con cuatro unidades de disco. La distribución de datos entre varias unidades de esta forma se llama ***striping (multiplexaje de datos)***. Por ejemplo, si el software emite un comando para leer un bloque de datos que consiste en cuatro tiras consecutivas que comienzan en una frontera de tira, el controlador de RAID desglosará este comando en cuatro comandos individuales, uno para cada uno de los cuatro discos, y los hará operar en paralelo. Así tenemos E/S en paralelo sin que el software se entere de ello.

El RAID nivel 0 funciona mejor cuando la solicitud de datos es grande, mientras más grande mejor. Si una solicitud es mayor que el número de unidades multiplicado por el tamaño de una tira, algunas unidades recibirán varias solicitudes, de modo que cuando terminan de atender la primera solicitud comienzan a atender la segunda. Corresponde al controlador dividir en partes la solicitud y alimentar los comandos apropiados a los discos correctos en la secuencia debida, y luego ensamblar los resultados correctamente en la memoria. El desempeño es excelente y la implementación es sencilla.

El peor funcionamiento de RAID nivel 0 se obtiene con sistemas operativos que acostumbran solicitar datos sector por sector. Los resultados son correctos, pero no hay paralelismo y por tanto no mejora el desempeño. Otra desventaja de esta organización es que la confiabilidad podría ser peor que si se tuviera un SLED. Si un RAID consta de cuatro discos, cada uno con un tiempo medio antes de fallo de 20,000 horas, una unidad fallará aproximadamente una vez cada 5000 horas y se perderán totalmente los datos. Un SLED con un tiempo medio antes de fallo de 20,000 horas sería cuatro veces más confiable. Dado que este diseño no incluye una verdadera redundancia, no es realmente un RAID.

La siguiente opción, RAID nivel 1, que se muestra en la figura 2-23(b), es un verdadero RAID. Se duplican todos los discos, de modo que hay cuatro discos primarios y cuatro discos de respaldo. En una escritura, cada tira se escribe dos veces. En una lectura, puede utilizarse cualquiera de las dos copias, lo que permite distribuir la carga entre más unidades. Por consiguiente, el desempeño de escritura no es mejor que con una sola unidad, pero el desempeño de lectura puede ser hasta dos veces mejor. La tolerancia de fallos es excelente: si una unidad se descompone, simplemente se usa la copia. La recuperación consiste en instalar una nueva unidad y copiar en ella el contenido de la unidad de respaldo.

A diferencia de los niveles 0 y 1, que funcionan con tiras de sectores, RAID nivel 2 funciona con base en palabras, posiblemente incluso con base en bytes. Imagine dividir cada byte del disco virtual único en un par de "nibbles" de cuatro bits, y luego agregar un código de Hamming a cada uno para formar una palabra de siete bits, de la cual los bits 1, 2 y 4 son bits de paridad. Imagine además que las siete unidades de la figura 2-23(c) se sincronizan en términos de posición del brazo y posición rotacional. Entonces sería posible escribir la palabra de siete bits con código de Hamming distribuida en las siete unidades, un bit por unidad.

La computadora CM-2 de Thinking Machines usaba este esquema, tomando palabras de datos de 32 bits y añadiendo seis bits de paridad para formar una palabra de Hamming de 38 bits, más un bit extra para la paridad de palabras, distribuyendo cada palabra entre 39 unidades de disco. El rendimiento total era inmenso, porque en un tiempo de sector se podían escribir 32 sectores de datos. Además, perder una unidad no causaba problemas, porque equivalía a perder un bit de cada palabra de 39 bits leída, algo que el código Hamming podía manejar sobre la marcha.

En el lado negativo, este esquema requiere que todas las unidades estén sincronizadas rotacionalmente, y sólo tiene sentido si el número de unidades es considerable (aun con 32 unidades de datos y 6 unidades de paridad, el gasto extra es de 19%). También se exige mucho al controlador, ya que debe efectuar una suma de verificación de Hamming cada tiempo de bit.

RAID nivel 3 es una versión simplificada de RAID nivel 2, y se ilustra en la figura 2-23(d). Aquí se calcula un solo bit de paridad para cada palabra de datos y se escribe en una unidad de disco de paridad. Al igual que en el RAID nivel 2, las unidades deben estar sincronizadas con exactitud, puesto que las palabras de datos individuales se distribuyen entre varias unidades.

A primera vista, podría parecer que un solo bit de paridad permite detectar errores, pero no corregirlos. En el caso de errores aleatorios no detectados esta observación es correcta, pero en el caso de una unidad que se descompone ofrece plena corrección de errores de un bit, ya que se sabe la posición del bit erróneo. Si una unidad falla, el controlador simplemente supone que todos sus bits son 0. Si una palabra tiene error de paridad, el bit de la unidad inactiva tiene que haber sido un 1, así que se corrige. Aunque ambos niveles de RAID, 2 y 3, ofrecen tasas de datos muy altas, el número de solicitudes de E/S distintas que pueden manejar por segundo no es mayor que con una sola unidad de disco.

Los niveles RAID 4 y 5 trabajan otra vez con tiras, no palabras individuales con paridad, y no requieren unidades sincronizadas. RAID nivel 4 [vea la figura 2-23(e)] se parece a RAID nivel 0, sólo que se escribe una paridad tira por tira en una unidad extra. Por ejemplo, si cada tira tiene k bytes de longitud, se calcula el OR EXCLUSIVO de todas las tiras para obtener una tira de paridad de k bytes. Si una unidad falla, los bytes perdidos se pueden recalcular a partir de la unidad de paridad.

Este diseño protege contra la pérdida de una unidad de disco pero tiene un desempeño pobre en el caso de actualizaciones pequeñas. Si un sector se modifica, es necesario leer todas las unidades para recalcular la paridad, que luego debe reescribirse. Como alternativa, se pueden leer los datos del usuario y de paridad antiguos y recalcular la nueva paridad a partir de ellos. Incluso con esta optimación, una actualización pequeña requiere dos lecturas y dos escrituras.

Lo pesado de la carga sobre la unidad de disco de paridad podría hacer que ésta se convirtiera en un cuello de botella. Esto se resolvería en el RAID nivel 5 distribuyendo los

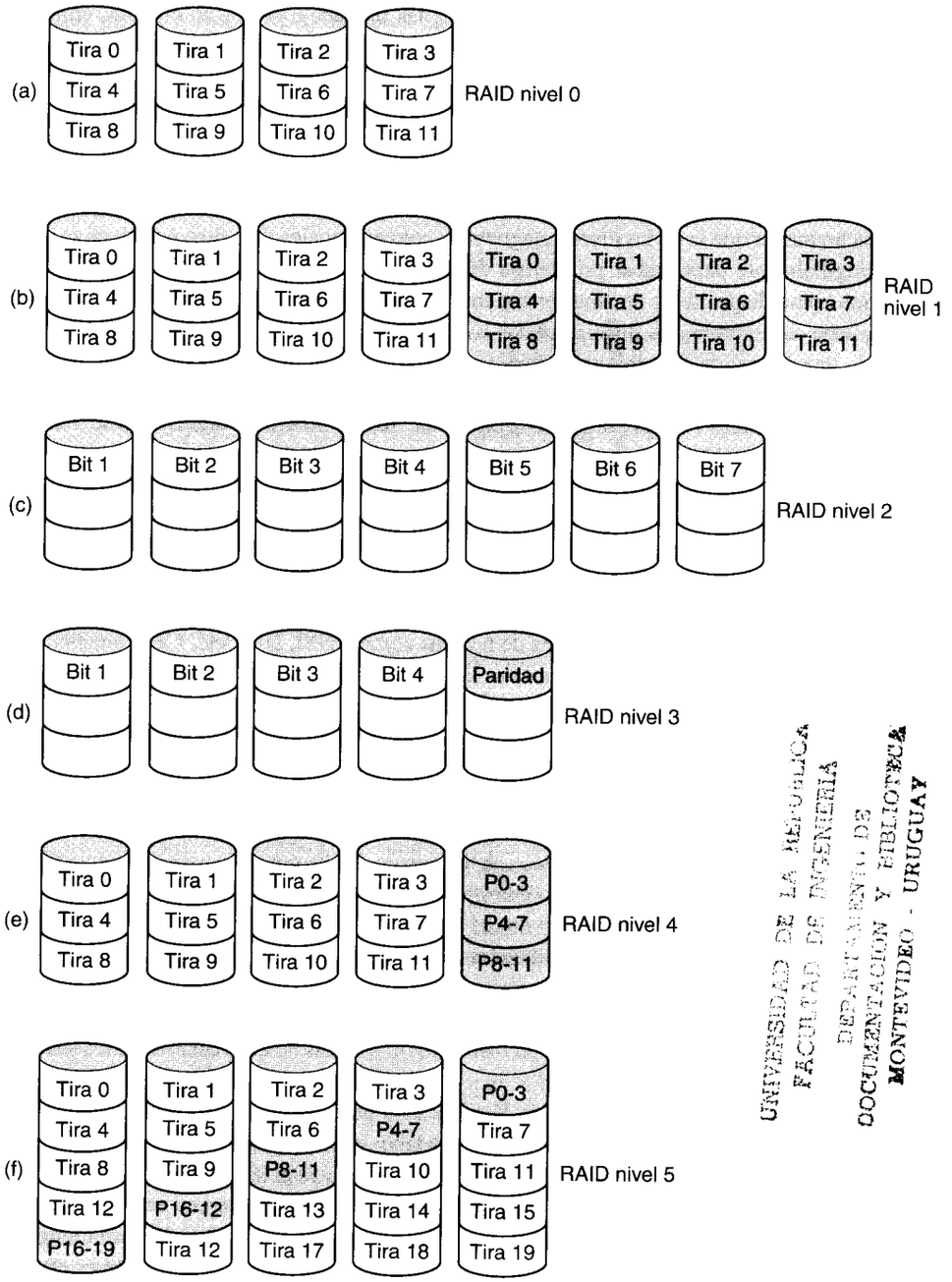


Figura 2-23. RAID niveles 0 a 5. Las unidades de respaldo y paridad se muestran sombreadas.

bits de paridad de manera uniforme entre todas las unidades, por turno circular, como se muestra en la figura 2-23(f). Sin embargo, si una unidad fallara, la reconstrucción de su contenido sería un proceso complejo.

2.3.7 CD-ROM

En años recientes han aparecido discos ópticos (no magnéticos) cuya densidad de grabación es mucho mayor que la de los discos magnéticos convencionales. Los discos ópticos se crearon originalmente para grabar programas de televisión, pero se les puede dar un uso más estético como dispositivos de almacenamiento para computadora. A causa de su capacidad, que puede ser enorme, los discos ópticos han sido tema de muchas investigaciones y han sufrido una evolución increíblemente rápida.

Los discos ópticos de primera generación fueron inventados por el conglomerado de electrónica holandés Philips para grabar películas. Tenían 30 cm de diámetro y se vendían con el nombre LaserVision, pero no se popularizaron fuera de Japón.

En 1980 Philips, junto con Sony, desarrolló el CD (disco compacto, *Compact Disc*), que rápidamente sustituyó al disco de vinilo de 33 1/3 rpm para música grabada. Los detalles técnicos precisos del CD se publicaron en una Norma Internacional oficial (IS 10149) conocido popularmente como el **Libro Rojo**, por el color de su portada. (Las Normas Internacionales son emitidas por la Organización Internacional de Estandarización, que es la contraparte internacional de los grupos nacionales de normas como ANSI, DIN, etc. Cada una tiene un número IS.) Lo que se buscaba al publicar las especificaciones del disco y la unidad como Norma Internacional fue permitir que un CD de diferentes editores de música y reproductores de diferentes compañías electrónicas funcionaran juntos. Todos los CD tienen 120 mm de diámetro y 1.2 mm de espesor, con un orificio central de 15 mm. El CD de audio fue el primer medio de almacenamiento digital que tuvo éxito en el mercado masivo. Se supone que los CD duran 100 años. No olvide preguntar en 2080 qué tan bueno salió el primer lote.

Un CD se prepara utilizando un láser de infrarrojo de alta potencia para quemar orificios de 0.8 micras de diámetro en un disco maestro de vidrio recubierto. Este disco maestro sirve para hacer un molde, con salientes en donde estarán los orificios del láser. En este molde se inyecta resina de policarbonato fundida para formar un CD con el mismo patrón de orificios que el máster de vidrio. Luego se deposita una capa muy delgada de aluminio reflejante sobre el policarbonato, seguida de una laca protectora y finalmente una etiqueta. Las depresiones en el sustrato de policarbonato se llaman **fosos** (*pits*); las áreas no quemadas entre los fosos se llaman **lands**.

Durante la reproducción, un diodo láser de baja potencia dirige luz infrarroja con una longitud de onda de 0.78 micras hacia los fosos y lands que van pasando. El láser incide en el lado del policarbonato, por lo que los fosos se proyectan hacia el láser como salientes de una superficie por lo demás plana. Dado que los fosos tienen una altura de una cuarta parte de la longitud de onda de la luz láser, la luz que se refleja de un foso está defasada media longitud de onda respecto a la luz que se refleja de la superficie circundante. El resultado es que las dos partes se interfieren destructivamente y devuelven menos luz al fotodetector del reproductor que la luz que se refleja de un land. Es así como el reproductor distingue un foso de un land.

Aunque podría parecer que lo más fácil es usar un foso para registrar un 0 y un land para registrar un 1, es más confiable utilizar una transición foso/land o land/foso para registrar un 1, y su ausencia para registrar un 0, y éste es el esquema que se usa.

Los fosos y lands se escriben en una sola espiral continua que comienza cerca del agujero y llega hasta una distancia de 32 mm del borde. La espiral describe 22,188 revoluciones alrededor del disco (unas 600 por mm). Si se desenrollara, tendría una longitud de 5.6 km. La espiral se ilustra en la figura 2-24.

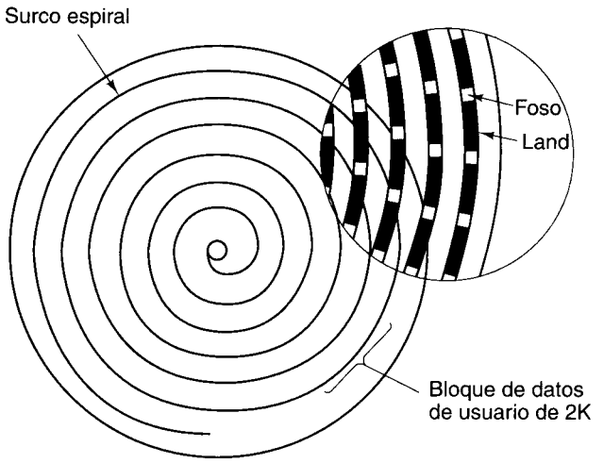


Figura 2-24. Estructura de grabación de un disco compacto o CD-ROM.

Para poder reproducir la música con una rapidez uniforme es preciso que los fosos y lands pasen bajo el láser con una velocidad lineal constante. Por consiguiente, es preciso reducir continuamente la rapidez de rotación del CD a medida que la cabeza de lectura se mueve del interior del CD hacia el exterior. En el interior, la rapidez de rotación es de 530 rpm para lograr la velocidad lineal deseada de 120 cm/s; en el exterior tiene que bajar a 200 rpm para dar la misma velocidad lineal en la cabeza. Una unidad de disco con velocidad lineal constante es muy diferente de una unidad de disco magnético, que opera a velocidad angular constante, independiente de la posición de la cabeza. Además, 530 rpm es muy diferente de las 3600 a 7200 rpm con que gira la mayor parte de los discos magnéticos.

En 1984, Philips y Sony se percataron del potencial que tenían los CD para almacenar datos de computadora, así que publicaron el **Libro Amarillo** que define una norma precisa para lo que ahora llamamos **CD-ROM (disco compacto-memoria sólo de lectura**, *Compact Disc - Read Only Memory*). A fin de aprovechar el para entonces floreciente mercado de los CD de audio, los CD-ROM tendrían el mismo tamaño físico que los CD de audio, serían mecánica y ópticamente compatibles con ellos, y se producirían empleando las mismas máquinas de moldeo por inyección de policarbonato. Las consecuencias de esta decisión fueron

que se requerirían motores lentos de velocidad variable, pero también que el costo de fabricación de un CD-ROM sería muy inferior a un dólar en volúmenes moderados.

Lo que el Libro Amarillo definió fue el formato de los datos para computadora, y también mejoró las capacidades de corrección de errores del sistema. Este paso era indispensable porque, si bien a los aficionados a la música no les importa perder un bit de vez en cuando, los aficionados a la computación suelen ser muy melindrosos al respecto. El formato básico de un CD-ROM consiste en codificar cada byte en un símbolo de 14 bits. Como vimos antes, 14 bits bastan para codificar por Hamming un byte de 8 bites y hasta sobran dos bits. De hecho, se utiliza un sistema de codificación más potente. La transformación de 14 a 8 durante la lectura se efectúa en hardware mediante tablas.

En el siguiente nivel superior, un grupo de 42 símbolos consecutivos constituye un **cuadro** de 588 bits. Cada cuadro contiene 192 bits de datos (24 bytes). Los 396 bits restantes sirven para corrección de errores y control. Hasta aquí, el esquema es idéntico para los CD de audio y los CD-ROM.

Lo que el Libro Amarillo añade es el agrupamiento de 98 cuadros en un **sector de CD-ROM**, como se muestra en la figura 2-25. Cada sector de CD-ROM comienza con un preámbulo de 16 bytes, de los cuales los primeros 12 son 00FFFFFFFFFFFFFFFFFFFF00 (hexadecimal), para que el reproductor pueda reconocer el inicio de un sector de CD-ROM. Los siguientes tres bytes contienen el número de sector, que es necesario porque buscar en un CD-ROM con su espiral de datos única es mucho más difícil que en un disco magnético con sus pistas concéntricas uniformes. Para buscar, el software de la unidad calcula aproximadamente a dónde debe mover la cabeza, coloca la cabeza ahí, y comienza a buscar un preámbulo para ver cuán cerca está de donde debería estar. El último byte del preámbulo es el modo.

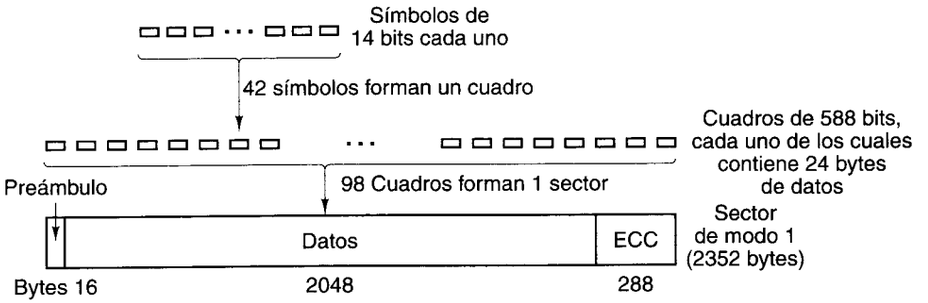


Figura 2-25. Organización lógica de datos en un CD-ROM.

El Libro Amarillo define dos modos. El modo 1 utiliza la organización de la figura 2-25, con un preámbulo de 16 bytes, 2048 bytes de datos y un código de corrección de errores de 288 bytes (un código Reed-Solomon con intercalación cruzada). El modo 2 combina los campos de datos y de ECC en un campo de datos de 2336 bytes para las aplicaciones que no necesitan (o que no tienen tiempo para efectuar) corrección de errores, como audio y video. Cabe señalar que, a fin de lograr una confiabiliad excelente, se emplean tres esquemas de corrección de errores distintos: dentro de un símbolo, dentro de un cuadro y dentro de un

sector de CD-ROM. Los errores de un solo bit se corrigen en el nivel más bajo, los errores de ráfaga corta se corrigen en el nivel de cuadro y cualesquier errores residuales se atrapan en el nivel de sector. El precio que se paga por esta confiabilidad es que se requieren 98 cuadros de 588 bits (7203 bytes) para llevar una carga útil de 2048 bytes, una eficiencia de sólo 28%.

Las unidades de CD-ROM de velocidad sencilla operan a 75 sectores/s, lo que da una tasa de datos de 153,600 bytes/s en el modo 1 y 175,200 bytes/s en el modo 2. Las unidades de doble velocidad son dos veces más rápidas, y así hasta la velocidad más alta. Un CD de audio estándar tiene capacidad para 74 minutos de música que, si se usara para datos en modo 1 sería de 681,984,000 bytes. Esta cifra suele informarse como 650 MB porque 1 MB es 2^{20} bytes (1,048,576 bytes), no 1,000,000 bytes.

Cabe señalar que ni siquiera una unidad CD-ROM 32x (4,915,200 bytes/s) es rival para una unidad de disco magnético SCSI-2 rápida que opera a 10 MB/s, aunque muchas unidades CD-ROM utilizan la interfaz SCSI (también existen unidades CD-ROM IDE). Si nos damos cuenta de que el tiempo de búsqueda suele ser de varios cientos de milisegundos, es evidente que las unidades de CD-ROM no están en la misma categoría de desempeño que las de disco magnético, a pesar de su gran capacidad.

En 1986, Philips dio un nuevo golpe con el **Libro Verde**, que añadía gráficos y la capacidad para intercalar audio, video y datos en el mismo sector, función indispensable en los CD-ROM de multimedia.

La última pieza del rompecabezas de los CD-ROM es el sistema de archivos. Para poder usar el mismo CD-ROM en diferentes computadoras, era necesario llegar a un acuerdo respecto a los sistemas de archivos de los CD-ROM. Para lograr este acuerdo, representantes de muchas compañías de computación se reunieron en Lake Tahoe, en la frontera entre California y Nevada, e idearon un sistema de archivos que llamaron **High Sierra** (por el nombre que recibe la región) y que evolucionó hasta convertirse en una Norma Internacional (IS 9660). La norma tiene tres niveles. En el nivel 1 se usan nombres de archivo de hasta ocho caracteres seguidos opcionalmente de hasta tres caracteres (la convención para nombres de archivo en MS-DOS). Los nombres de archivo sólo pueden contener letras mayúsculas, dígitos y el carácter de subrayado. Los directorios pueden anidarse hasta una profundidad de ocho, pero los nombres de directorio no pueden contener extensiones. El nivel 1 requiere que todos los archivos sean contiguos, lo cual no es un problema en un medio que se escribe sólo una vez. Cualquier CD-ROM que se ajuste a la IS 9660 nivel 1 se puede leer con MS-DOS, una computadora Apple, una computadora UNIX o casi cualquier otra computadora. Los productores de CD-ROM consideran esta propiedad como una gran ventaja.

IS 9660 nivel 2 permite nombres de hasta 32 caracteres, y el nivel 3 permite archivos no contiguos. Las extensiones Rock Ridge (nombre que se les dio caprichosamente para recordar el pueblo que sale en la película *Blazing Saddles* de Gene Wilder) permiten nombres muy largos (para UNIX), UID, GID y vínculos simbólicos, pero los CD-ROM que no se ajustan al nivel 1 no pueden leerse en todas las computadoras.

Los CD-ROM se han vuelto extremadamente populares para publicar juegos, películas, enciclopedias, atlas y obras de referencia de todo tipo. Casi todo el software comercial se vende ahora en CD-ROM. Su combinación de alta capacidad y bajo costo de fabricación los hace ideales para un sinnúmero de aplicaciones.

2.3.8 CD grabables

Inicialmente, el equipo requerido para producir un máster de CD-ROM (o de un CD de audio, para el caso) era extremadamente costoso. Pero como suele suceder en la industria de la computación, todos los precios bajan tarde o temprano. Para mediados de los noventa las grabadoras de CD no más grandes que un reproductor de CD, eran un periférico común que podía adquirirse en casi cualquier tienda de computación. Estos dispositivos seguían siendo diferentes de los discos magnéticos porque, una vez escritos, los CD-ROM no podían borrarse. No obstante, encontraron rápidamente un nicho como medio de respaldo para discos duros grandes y también permitieron a individuos o compañías incipientes fabricar sus propios CD-ROM en cantidades pequeñas o producir másters para entregar a plantas de duplicación de CD comerciales de alto volumen. Estas unidades se denominan **CD-R** (**CD grabables**, *CD-Recordables*).

Físicamente, los CD-R parten de discos de policarbonato en blanco de 120 mm que son parecidos a los CD-ROM, sólo que contienen un surco de 0.6 mm de ancho para guiar al láser durante la escritura. El surco tiene una excursión senoidal de 0.3 mm a una frecuencia de exactamente 22.05 kHz, a fin de proporcionar una retroalimentación continua que permita vigilar con exactitud la velocidad de rotación, y ajustarla si es necesario. Los CD-R tienen un aspecto parecido al de los CD-ROM normales, excepto que son dorados por la parte de arriba, en lugar de plateados. El color dorado proviene del uso de oro en lugar de aluminio para la capa reflejante. A diferencia de los CD plateados, que tienen depresiones físicas, en los CD-R la diferencia en reflectividad entre los fosos y los lands tiene que simularse. Esto se hace añadiendo una capa de colorante entre el policarbonato y la capa de oro reflejante, como se muestra en la figura 2-26. Se utilizan dos tipos de colorante: cianina, que es verde, y ftalocianina, que es de color anaranjado amarillento. Los químicos pueden discutir horas y horas acerca de cuál es mejor. Dichos colorantes son similares a los que se utilizan en fotografía, lo que explica por qué Kodak y Fuji son fabricantes importantes de discos CD-R.

En su estado inicial, la capa de colorante es transparente y permite a la luz láser pasar a través y reflejarse en la capa de oro. Para escribir, el láser del CD-R se alimenta con alta potencia (8-16 mW). Cuando el haz incide sobre un punto con colorante, lo calienta y rompe uno de sus enlaces químicos. Este cambio en la estructura molecular crea un punto oscuro. Durante la reproducción (a 0.5 mW), el fotodetector percibe una diferencia entre los puntos oscuros donde el colorante fue quemado y las áreas transparentes donde está intacto. Esta diferencia se interpreta como la diferencia entre fosos y lands, cuando se reproduce con un lector de CD-ROM normal o con un reproductor de CD de audio.

Un tipo nuevo de CD sentiría una vergüenza enorme si no tuviera su propio libro de color, así que el CD-R tiene el **Libro Anaranjado**, que se publicó en 1989. Este documento define el CD-R y también un nuevo formato, **CD-ROM XA**, que permite escribir los CD-R de forma incremental: unos cuantos sectores hoy, otros pocos mañana, y unos más el mes próximo. Un grupo de sectores consecutivos que se escriben en una sola operación se llama **pista de CD-ROM**.

Uno de los primeros usos del CD-R fue el PhotoCD de Kodak. Con este sistema, el cliente lleva al procesador fotográfico un rollo de película expuesta y su antiguo PhotoCD, y luego recoge el mismo PhotoCD al que se han añadido las nuevas imágenes después de las

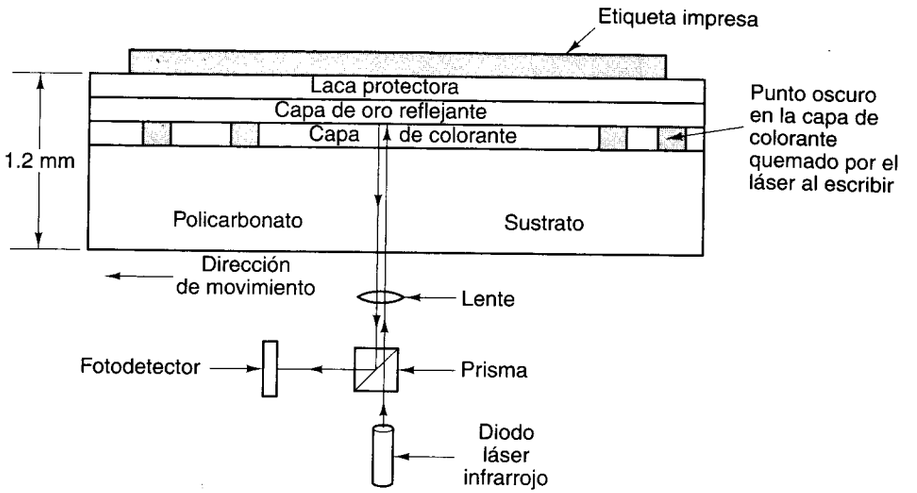


Figura 2-26. Sección transversal de un disco CD-R y láser (no está a escala). Un CD-ROM plateado tiene una estructura similar, excepto que no tiene capa de colorante y tiene una capa de aluminio con fosos en lugar de una capa de oro.

antiguas. El nuevo lote, que se crea digitalizando los negativos, se escribe en el PhotoCD como pista de CD-ROM aparte. Se requiere una escritura incremental porque los discos de CD-R en blanco son demasiado costosos como para gastar uno en cada rollo de película.

Sin embargo, la escritura incremental crea un problema nuevo. Antes del Libro Anaranjado, todos los CD-ROM tenían una sola **VTOC** (**tabla de contenido del volumen**, *Volume Table of Contents*) al principio. Ese esquema no funciona con las escrituras incrementales (es decir, de múltiples pistas). La solución del Libro Anaranjado es dar a cada pista de CD-ROM su propia VTOC. Los archivos enumerados en la VTOC pueden incluir algunos de los archivos de pistas anteriores, o todos. Una vez que el CD-R se inserta en la unidad, el sistema operativo examina todas las pistas de CD-ROM hasta encontrar la VTOC más reciente, que contiene la situación actual del disco. Al incluir algunos de los archivos de pistas anteriores, pero no todos, en la VTOC vigente, es posible crear la ilusión de que se han borrado archivos. Las pistas pueden agruparse en **sesiones**, lo que da pie a CD-ROM **multisesión**. Los reproductores de CD de audio estándar no pueden manejar los CD multisesión poque esperan una sola VTOC al principio.

Cada pista tiene que escribirse en una sola operación continua sin pausas. Por consiguiente, el disco duro del cual provienen los datos tiene que ser lo bastante rápido como para suministrarlos a tiempo. Si los archivos que se van a copiar están dispersos por todo el disco duro, los tiempos de búsqueda podrían hacer que el flujo de datos al CD-R se interrumpa y propicie una insuficiencia de buffer. El resultado es un bonito y brillante (pero un tanto costoso) posavasos para sus bebidas, o un frisbee (disco volador) dorado de 120 mm. El software de CD-R generalmente ofrece la opción de reunir todos los archivos de entrada en una sola imagen de CD-ROM contigua de 650 MB antes de quemar el CD-R, pero este proceso nor-

malmente duplica el tiempo de escritura efectivo, requiere 650 MB de espacio libre en disco y aun así no protege contra las recalibraciones térmicas repentinas que efectúan los discos duros cuando se calientan demasiado.

El CD-R permite a individuos y compañías copiar fácilmente los CD-ROM (y los CD de audio), generalmente con violación de los derechos de autor del productor. Se han ideado varios esquemas para obstaculizar tal piratería y dificultar la lectura de un CD-ROM utilizando otra cosa que no sea el software del productor. Uno de ellos implica registrar la longitud de los archivos del CD-ROM como de varios gigabytes, lo que frustra cualquier intento por copiar los archivos en el disco duro empleando software de copiado estándar. Las verdaderas longitudes están incorporadas en el software del productor u ocultas (tal vez en forma cifrada) en el CD-ROM en un lugar inesperado. Otro esquema utiliza ECC intencionalmente incorrectos en sectores selectos, con la esperanza de que el software de copiado de CD "corrija" los errores. El software de aplicación verifica los ECC por sí mismo, y se niega a funcionar si son correctos. Otras posibilidades son el uso de separaciones no estándar entre las pistas y otros "defectos" físicos.

2.3.9 CD reescribibles

Aunque estamos acostumbrados a varios medios de escritura única, como el papel y la película fotográfica, existe demanda por un CD-ROM reescribible. Una tecnología que ya está disponible es **CD-RW** (**CD reescribible**, *CD-ReWritable*), que utiliza medios del mismo tamaño que los CD-R, sólo que en lugar de un colorante como cianina o ftalocianina, el CR-RW emplea una aleación de plata, indio, antimonio y telurio para la capa de grabación. Esta aleación tiene dos estados estables: cristalina y amorfa, con diferente reflectividad.

Las unidades de CD-RW emplean láseres con tres potencias distintas. A la potencia alta, el láser funde la aleación, transformándola del estado cristalino de alta reflectividad, al estado amorfo de baja reflectividad. Esto representa un foso. A la potencia media, la aleación se funde y se recristaliza en su estado natural para convertirse otra vez en un land. A la potencia baja, el estado del material se detecta (para lectura), pero no ocurre ninguna transición de fase.

La razón por la que el CD-RW no ha suplantado al CD-R es que los discos CD-RW en blanco son mucho más costosos que los de CD-R. Además, cuando el disco se usa para crear copias de seguridad de discos duros, el hecho de que una vez escrito un CD-R no puede borrarse accidentalmente es una gran ventaja.

2.3.10 DVD

El formato de CD/CD-ROM básico se estableció desde 1980. Desde entonces la tecnología ha mejorado y ha hecho factibles económicamente discos ópticos de mayor capacidad, para los cuales hay gran demanda. A Hollywood le encantaría sustituir las videocintas analógicas por discos digitales, ya que éstos tienen más alta calidad, su fabricación cuesta menos, duran más, ocupan menos espacio en los anaqueles de las tiendas de video y no tienen que rebobinarse. Las compañías de electrónica para el consumidor están buscando un nuevo producto que se

venda mucho, y un gran número de compañías de computación quieren añadir características multimedia a su software.

Esta combinación de tecnología y demanda por parte de tres industrias inmensamente ricas y poderosas ha dado origen al **DVD**, que originalmente era un acrónimo de **videodisco digital** (*Digital Video Disc*) pero ahora significa oficialmente **disco digital versátil** (*Digital Versatile Disc*). Los DVD tienen el mismo diseño general que los CD, con discos de policarbonato de 120 mm moldeados por inyección que contienen fosos y lands y que se iluminan con un diodo láser para ser leídos por un fotodetector. Lo que es nuevo es el uso de

1. Fosos más pequeños (0.4 micras en lugar de 0.8 micras para los CD).
2. Una espiral más cerrada (0.74 micras entre pistas en lugar de 1.6 micras para los CD).
3. Un láser rojo (de 0.65 micras en vez de 0.78 micras para los CD).

Juntas, estas mejoras multiplican la capacidad siete veces, a 4.7 GB. Una unidad de DVD 1x opera a 1.4 MB/s (compárese con 150 KB/s para los CD). Desafortunadamente, el cambio a los láseres rojos empleados en los supermercados implica que los reproductores de DVD requieren un segundo láser o una óptica de conversión precisa para poder leer los CD y CD-ROM existentes, algo que tal vez no hagan algunos de ellos. Además, quizá podrían no leerse los CD-R y los CD-RW en una unidad DVD.

¿Es suficiente 4.7 GB? Tal vez. Utilizando compresión MPEG-2 (estandarizada en IS 13346), un DVD de 4.7 GB puede contener 133 minutos de video de pantalla completa con pleno movimiento y alta definición (720×480), así como pistas sonoras en hasta ocho idiomas y subtítulos en 32 más. Cerca de 92% de las películas que ha hecho Hollywood duran menos de 133 minutos. No obstante, algunas aplicaciones como los juegos de multimedia o las obras de referencia podrían necesitar más, y a Hollywood le gustaría grabar varias películas en el mismo disco. Por ello, se han definido cuatro formatos:

1. Un solo lado, una sola capa (4.7 GB).
2. Un solo lado, capa dual (8.5 GB).
3. Dos lados, una sola capa (9.4 GB).
4. Dos lados, capa dual (17 GB).

¿Por qué tantos formatos? En una palabra: política. Philips y Sony querían discos de un solo lado y capa dual para la versión de alta capacidad, pero Toshiba y Time Warner querían discos de dos lados y una sola capa. Philips y Sony no creían que la gente estaría dispuesta a darle vuelta al disco, y Time Warner no creía que fuera factible colocar dos capas en un mismo lado. La concesión: todas las combinaciones, pero el mercado decidirá cuáles han de sobrevivir.

La tecnología de capa dual tiene una capa reflejante abajo, cubierta por una capa semirreflejante. Dependiendo de la distancia focal del láser, el haz se refleja en una capa o en la otra. La capa inferior necesita fosos y lands un poco mayores para que su lectura sea confiable, por lo que su capacidad es un poco menor que la de la capa superior.

Los discos de dos lados se fabrican tomando dos discos de un solo lado, de 0.6 mm, y pegándolos espalda con espalda. Para que el espesor de todas las versiones sea el mismo, un disco de un solo lado consiste en un disco de 0.6 mm pegado a un sustrato en blanco (o tal vez, en el futuro, uno que contenga 133 minutos de anuncios, en la esperanza de que la gente sentirá curiosidad por ver qué hay ahí abajo). La estructura del disco de dos lados y capa dual se ilustra en la figura 2-27.

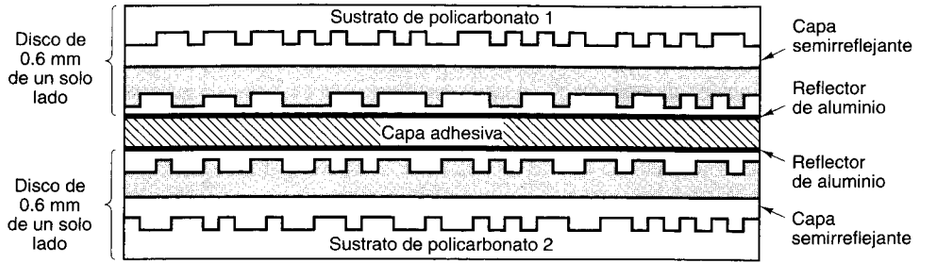


Figura 2-27. Disco DVD de dos lados y capa dual.

El DVD fue inventado por un consorcio de 10 compañías de electrónica para el consumidor, siete de ellas japonesas, en estrecha cooperación con los principales estudios de Hollywood (algunos de los cuales son propiedad de las compañías japonesas de electrónica del consorcio). Las industrias de las computadoras y telecomunicaciones no fueron invitadas al picnic, y el enfoque radicó en el uso del DVD para la renta de películas y exposiciones de ventas. Por ejemplo, entre las funciones estándar están la omisión en tiempo real de escenas fuertes (para que los padres puedan convertir una película para adultos en una que toda la familia pueda ver), sonido de seis canales, y apoyo de Pan-and-Scan. Esta última función permite al reproductor de DVD decidir dinámicamente cómo debe recortar los bordes izquierdo y derecho de las películas (cuya relación anchura:altura es de 3:2) de modo que quepan en las pantallas de televisión actuales (cuya relación es 4:3).

Otro aspecto sobre el que tal vez no se había hablado en la industria de las computadoras es una incompatibilidad intencional entre los discos hechos para distribuirse en Estados Unidos y aquéllos para Europa, además de otros estándares para otros continentes. Hollywood exigió esta "función" porque las películas nuevas siempre se exhiben primero en Estados Unidos y luego se envían a Europa cuando los videos salen a la venta en Estados Unidos. Lo que se buscaba era evitar que las tiendas de video europeas compraran videos en Estados Unidos antes de tiempo, lo cual reduciría los ingresos por la exhibición de esas películas en los cines europeos. Si Hollywood controlara la industria de las computadoras, se usarían disquetes de 3.5" en Estados Unidos y de 9cm en Europa.

Si el DVD tiene mucho éxito, en poco tiempo se producirán en masa los DVD-R (grabables) y DVD reescribibles. Sin embargo, el éxito de los DVD no está garantizado, ya que las compañías de cable tienen un plan muy diferente para distribuir películas —video por demanda de cable— y la batalla apenas comienza.

2.4 ENTRADA/SALIDA

Como mencionamos al principio del capítulo, un sistema de computación tiene tres componentes principales: la CPU, las memorias (primaria y secundaria) y el equipo de **E/S (entrada/salida**, *Input/Output*) que incluye impresoras, escáneres y módems. Ya hemos hablado de la CPU y las memorias. Ha llegado el momento de examinar el equipo de E/S y ver cómo se conecta con el resto del sistema.

2.4.1 Buses

Físicamente, la mayor parte de las computadoras personales y estaciones de trabajo tiene una estructura similar a la que se muestra en la figura 2-28. La disposición usual es una caja metálica con una tarjeta grande de circuitos impresos en su base, llamada **tarjeta madre** o *motherboard* (o tarjeta familia o *parentboard*, para quienes quieren ser diplomáticamente correctos). La tarjeta madre contiene el chip de CPU, algunas ranuras en las que pueden insertarse módulos DIMM, y diversos chips de apoyo. Además, contiene un bus grabado a todo su largo, y zócalos en los que pueden insertarse los conectores de arista de tarjetas de E/S. A veces hay dos buses, uno de alta velocidad (para las tarjetas de E/S modernas) y uno de baja velocidad (para las tarjetas de E/S más viejas).

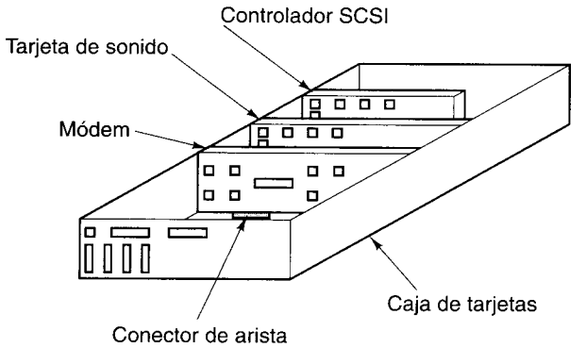


Figura 2-28. Estructura física de una computadora personal.

En la figura 2-29 se muestra la estructura lógica de una sencilla computadora personal. Ésta tiene un solo bus que sirve para conectar la CPU, la memoria y los dispositivos de E/S; casi todos los sistemas tienen dos o más buses. Cada dispositivo de E/S consta de dos partes: una que contiene casi todos los circuitos electrónicos, llamada **controlador**, y una que contiene el dispositivo de E/S propiamente dicho, como una unidad de disco. El controlador suele estar contenido en una tarjeta que se inserta en una ranura desocupada, con excepción de los controladores que no son opcionales (como el del teclado), los cuales a veces se encuentran en la tarjeta madre. Aunque la pantalla (monitor) no es opcional, el controlador de video a veces se encuentra en una tarjeta insertable para que el usuario pueda escoger entre tarjetas con o sin aceleradores de gráficos, memoria adicional, etc. El controlador se conecta a su dispositivo con un cable unido a un conector en la parte de trasera del gabinete.

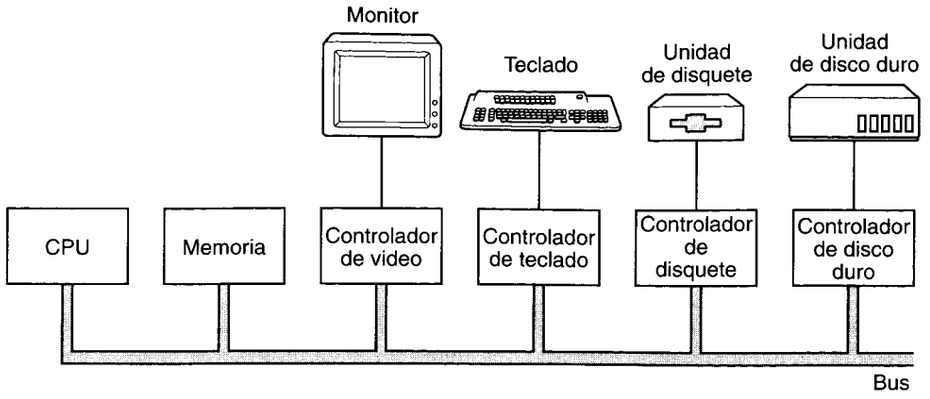


Figura 2-29. Estructura lógica de una computadora personal sencilla.

La tarea de un controlador es dominar su dispositivo de E/S y manejar su acceso al bus. Por ejemplo, cuando un programa quiere datos del disco, envía un comando al controlador del disco, que a su vez emite comandos de búsqueda y de otro tipo a la unidad de disco. Una vez que se localizan la pista y el sector apropiados, la unidad de disco comienza a enviar los datos como un flujo de bits en serie al controlador. Corresponde a éste dividir el flujo de bits en unidades y escribir cada unidad en la memoria conforme se ensambla. Las unidades constan normalmente de una o más palabras. Un controlador que lee datos de la memoria o los escribe en ella sin intervención de la CPU está efectuando **acceso directo a la memoria**, mejor conocido por sus iniciales en inglés, **DMA** (*Direct Memory Access*). Una vez completada la transferencia, el controlador normalmente genera una **interrupción** que obliga a la CPU a dejar de ejecutar su programa actual y comenzar a ejecutar un procedimiento especial, llamado **manejador de interrupciones**, para verificar la presencia de errores, efectuar cualquier acción especial que se requiera, e informar al sistema operativo sobre la finalización de la E/S. Cuando el manejador de interrupciones concluye su intervención, la CPU continúa con el programa que se suspendió cuando ocurrió la interrupción.

El bus no sólo es utilizado por los controladores de E/S, sino también por la CPU para obtener instrucciones y datos. ¿Qué sucede si la CPU y un controlador de E/S quieren usar el bus al mismo tiempo? La respuesta es que un chip llamado **árbitro de bus** decide quién tendrá el acceso. En general, los dispositivos de E/S tienen preferencia sobre la CPU, porque los discos y otros dispositivos móviles no pueden detenerse, y obligarlos a esperar podría causar la pérdida de datos. Si no se está efectuando E/S, la CPU puede aprovechar todos los ciclos de bus para hacer referencia a la memoria. Pero si también está funcionando un dispositivo de E/S, éste solicitará el bus cuando lo necesite, y se le otorgará. Este proceso se llama **robo de ciclos** y hace más lenta a la computadora.

Este diseño funcionaba bien en las primeras computadoras personales, porque todos los componentes estaban más o menos balanceados. Pero a medida que aumentó la rapidez de las CPU, memorias y dispositivos de E/S, surgió un problema: el bus ya no podía manejar la carga que se le presentaba. En un sistema cerrado, como una estación de trabajo de ingeniería,

la solución era diseñar un bus nuevo más rápido para el siguiente modelo. Puesto que a nadie se le ocurría pasar dispositivos de E/S de un modelo viejo a uno nuevo, la estrategia funcionaba bien.

En cambio, en el mundo de las PC era común que la gente modernizara su CPU pero quisiera seguir usando su impresora, su escáner y su módem anteriores con el nuevo sistema. Además, había surgido una enorme industria dedicada a proveer una inmensa gama de dispositivos de E/S para el bus de la IBM PC, y esta industria no tenía ningún interés en tirar por la borda toda su inversión y comenzar otra vez desde el principio. IBM aprendió esta lección por las malas cuando sacó la sucesora de la IBM PC, la serie PS/2. La PS/2 tenía un bus nuevo, más rápido, pero casi todos los fabricantes de clones siguieron usando el viejo bus de la PC, ahora llamado bus **ISA** (**arquitectura estándar de la industria**, *Industry Standard Architecture*). También, la mayoría de los fabricantes de discos y dispositivos de E/S siguieron produciendo controladores para ese bus, e IBM se encontró en la peculiar situación de ser el único fabricante de PC que no era compatible con IBM. Finalmente, IBM se vio obligada a apoyar otra vez el bus ISA. Por cierto, cabe señalar que ISA significa arquitectura del conjunto de instrucciones (*Instruction Set Architecture*) en el contexto de niveles de máquina, pero quiere decir arquitectura estándar de la industria en el ámbito de los buses.

No obstante, a pesar de la presión del mercado contra los cambios, el viejo bus realmente era demasiado lento, por lo que había que hacer algo. Esta situación hizo que otras compañías crearan máquinas con múltiples buses, uno de los cuales era el viejo bus ISA, o su suscesor compatible con lo existente, el EISA (**ISA extendido**). El más popular de éstos es actualmente el bus **PCI** (**interconexión de componentes periféricos**, *Peripheral Component Interconnect*). Intel diseñó este bus, pero decidió colocar todas las patentes en el dominio público para animar a toda la industria (incluidos sus competidores) a que lo adoptaran.

El bus PCI se puede usar en muchas configuraciones, pero la que se ilustra en la figura 2-30 es típica. Aquí la CPU se comunica con un controlador de memoria por una conexión dedicada de alta velocidad. El controlador se comunica con la memoria y con el bus PCI directamente, de modo que el tráfico entre la CPU y la memoria no usa el bus PCI. En cambio, los periféricos con gran ancho de banda (es decir, tasa de datos elevada), como los discos SCSI, se pueden conectar directamente al bus PCI. Además, el bus PCI tiene un puente con el bus ISA, lo que permite seguir usando controladores ISA y sus dispositivos. Una máquina con este diseño incluiría por lo regular tres o cuatro ranuras PCI vacías y otras tres o cuatro ranuras ISA vacías, para que los clientes pudieran insertar tarjetas de E/S ISA viejas (generalmente para dispositivos lentos) y también tarjetas de E/S PCI nuevas (generalmente para dispositivos rápidos).

Actualmente se venden muchas clases de dispositivos de E/S. A continuación describiremos algunos de los más comunes.

2.4.2 Terminales

Las terminales de computadora constan de dos partes: un teclado y un monitor. En el mundo de las mainframes, estas partes a menudo se integran en un solo dispositivo y se conectan a la computadora principal con una línea en serie o una línea telefónica. En las industrias de las reservaciones de vuelos, bancos y otras que están orientadas hacia mainframes, estos dispositivos se siguen usando ampliamente. En el mundo de las computadoras personales el teclado y el monitor son dispositivos independientes. En ambos casos, la tecnología de las dos partes es la misma.

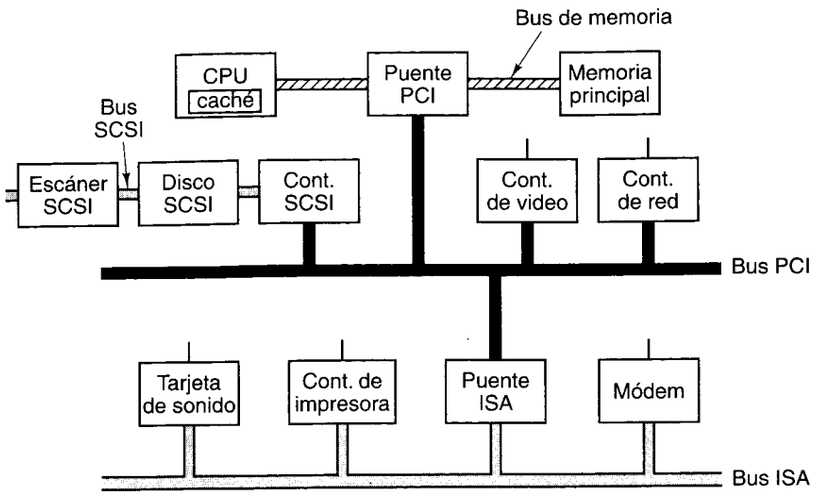


Figura 2-30. PC moderna representativa con un bus PCI y un bus ISA. El módem y la tarjeta de sonido son dispositivos ISA; el controlador SCSI es un dispositivo PCI.

Teclados

Los teclados son de varios tipos. La IBM PC original se vendía con un teclado que tenía un interruptor de acción brusca bajo cada tecla que proporcionaba retroalimentación táctil y emitía un chasquido cuando la tecla se oprimía lo suficiente. Hoy en día los teclados más baratos tienen teclas que se limitan a hacer contacto mecánico cuando se oprimen. Los mejores tienen una lámina de material elastomérico (una especie de caucho) entre las teclas y la tarjeta de circuitos impresos subyacente. Bajo cada tecla hay un pequeño domo que se aplasta cuando se le oprime con la fuerza suficiente. Un pequeño punto de material conductor dentro del domo cierra el circuito. Algunos teclados tienen un imán debajo de cada tecla que pasa por el interior de una bobina cuando la tecla se oprime e induce una corriente que puede detectarse. Además de éstos, se usan otros métodos, tanto mecánicos como electromagnéticos.

En las computadoras personales, cuando se oprime una tecla, se genera una interrupción que hace que se inicie el manejador de interrupciones del teclado (un programa que forma parte del sistema operativo). El manejador de interrupciones lee un registro de hardware dentro del controlador del teclado para obtener el número de la tecla (1 a 102) que recién se ha oprimido. Cuando se suelta una tecla, se causa una segunda interrupción. De este modo, cuando un usuario oprime la tecla SHIFT (mayúsculas), luego oprime y suelta la tecla M, y luego suelta la tecla SHIFT, el sistema operativo interpreta que el usuario quiere una "M" mayúscula y no una "m" minúscula. El manejo de secuencias de múltiples teclas en las que intervienen SHIFT, CTRL y ALT se realiza totalmente en software (incluida la secuencia CTRL-ALT-DEL de triste fama que sirve para reiniciar todas las IBM PC y sus clones).

Monitores de CRT

Un monitor es una caja que contiene un **CRT** (**tubo de rayos catódicos**, *Cathode Ray Tube*) y sus fuentes de potencia. El CRT contiene un cañón que puede disparar un haz de electrones contra una pantalla fosforescente cerca del frente del tubo, como se muestra en la figura 2-31(a). (Los monitores a color tienen tres cañones de electrones, uno para cada uno de los colores rojo, verde y azul.) Durante el barrido horizontal, el haz cruza la pantalla en unos 50 μs, describiendo una línea casi horizontal en ella. Luego ejecuta un retrazado horizontal para regresar al borde izquierdo e iniciar el siguiente barrido. Un dispositivo como éste que produce una imagen línea por línea se denomina dispositivo de **barrido por cuadro**.

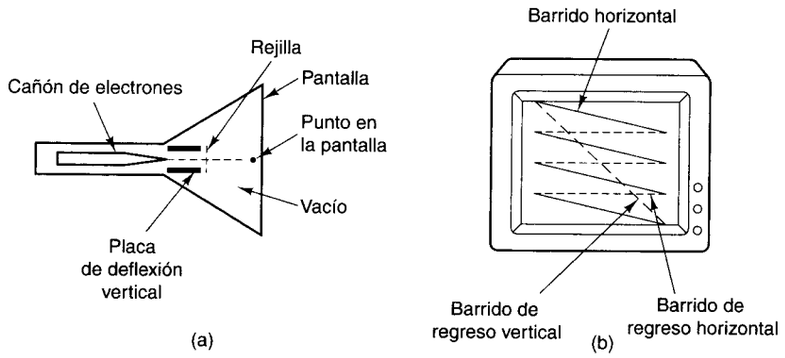


Figura 2-31. (a) Sección transversal de un CRT. (b) Patrón de barrido de un CRT.

El barrido horizontal se controla con un voltaje que aumenta linealmente y se aplica a las placas de desviación horizontal colocadas a la izquierda y la derecha del cañón de electrones. El movimiento vertical se controla con un voltaje cuyo incremento lineal es mucho más lento y se aplica a las placas de desviación vertical colocadas arriba y abajo del cañón. Después de unos 400 a 1000 barridos, los voltajes de las placas de desviación horizontal y vertical se invierten juntos rápidamente para colocar el haz otra vez en la esquina superior izquierda. Una imagen de pantalla completa normalmente se redibuja entre 30 y 60 veces cada segundo. Los movimientos del haz se muestran en la figura 2-31(b). Aunque según nuestra descripción se usan campos eléctricos para guiar el haz en su barrido de la pantalla, muchos modelos usan campos magnéticos en lugar de eléctricos, sobre todo los monitores más caros.

Para producir un patrón de puntos en la pantalla, hay una rejilla dentro del CRT. Cuando se aplica un voltaje positivo a la rejilla, los electrones se aceleran y hacen que el haz choque con la pantalla y la haga brillar efímeramente. Cuando se usa un voltaje negativo, los electrones se repelen, de modo que no pasan por la rejilla y la pantalla no brilla. Así, el voltaje aplicado a la rejilla hace que el patrón de bits correspondiente aparezca en la pantalla. Este mecanismo permite convertir una señal eléctrica binaria en una imagen formada por puntos brillantes y oscuros.

Pantallas planas

Los CRT son demasiado voluminosos y pesados para usarse en computadoras portátiles, y es por ello que se emplea una tecnología totalmente distinta en sus pantallas. La más común es la tecnología **LCD (presentación de cristal líquido**, *Liquid Crystal Display*). Esta tecnología es muy compleja, tiene muchas variaciones y está cambiando rápidamente, así que por fuerza la presente descripción será breve y muy simplificada.

Los cristales líquidos son moléculas orgánicas viscosas que fluyen como un líquido pero también tienen una estructura espacial, como un cristal. Un botánico austriaco, Rheinitzer, los descubrió en 1888, y se utilizaron en pantallas (de calculadoras y relojes de pulsera, por ejemplo) por primera vez en la década de los sesenta. Cuando todas las moléculas están alineadas en la misma dirección, las propiedades ópticas del cristal dependen de la dirección y la polarización de la luz incidente. Con la ayuda de un campo eléctrico es posible modificar la alineación molecular y por ende las propiedades ópticas del cristal. En particular, si se coloca una lámpara detrás de un cristal líquido, es posible controlar eléctricamente la intensidad de la luz que se percibe a través del cristal. Esta propiedad puede aprovecharse en la construcción de pantallas planas.

Una pantalla LCD consiste en dos placas de vidrio paralelas entre las que hay un volumen sellado que contiene un cristal líquido. Cada placa tiene conectados electrodos transparentes. Una luz detrás de la placa trasera (natural o artificial) ilumina la pantalla desde atrás. Los electrodos transparentes unidos a cada placa sirven para crear campos eléctricos en el cristal líquido. Diferentes partes de la pantalla reciben diferentes voltajes, y con esto se controla la imagen que se exhibe. Pegados a la parte de adelante y de atrás de la pantalla hay filtros polarizantes porque la tecnología de la pantalla requiere luz polarizada. La configuración general se muestra en la figura 2-32(a).

Aunque se usan muchos tipos de pantallas LCD, ahora consideraremos un tipo específico, la pantalla **TN (nemática torcida**, *Twisted Nematic*) como ejemplo. En esta pantalla, la placa trasera contiene diminutos surcos horizontales, y la delantera, verticales, como se muestra en la figura 2-32(b). En ausencia de un campo eléctrico, las moléculas de cristal líquido tienden a alinearse en la dirección de los surcos. Puesto que las alineaciones delantera y trasera difieren en 90 grados, las moléculas (y por tanto la estructura del cristal) se tuercen de atrás hacia adelante.

En la parte de atrás de la pantalla hay un filtro polarizante horizontal que únicamente deja pasar luz polarizada horizontalmente. En la parte delantera hay un filtro polarizante vertical que sólo permite pasar luz polarizada verticalmente. Si no hubiera líquido entre las placas, la luz polarizada horizontalmente que atraviesa el filtro polarizante trasero sería bloqueada por el filtro polarizante delantero, y la pantalla sería de color negro uniforme.

Sin embargo, la estructura cristalina torcida de las moléculas de cristal líquido guía la luz a medida que pasa y hace girar su polarización, de modo que sale polarizada horizontalmente. Así, en ausencia de un campo eléctrico, la pantalla LCD es uniformemente brillante. Si se aplica un voltaje a partes selectas de la placa, la estructura torcida se destruye, y la luz no atraviesa esas partes.

Se utilizan dos sistemas para aplicar el voltaje. En una **pantalla de matriz pasiva** (de bajo costo), ambos electrodos contienen alambres paralelos. En una pantalla de 640 × 480,

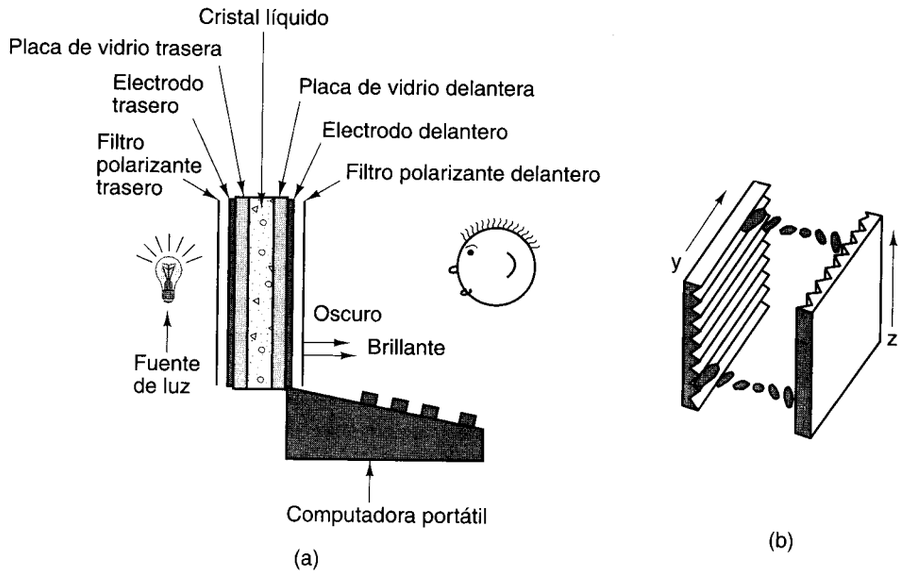


Figura 2-32. (a) Construcción de una pantalla LCD. (b) Los surcos de las placas delantera y trasera son mutuamente perpendiculares.

por ejemplo, el electrodo trasero podría tener 640 alambres verticales; y el delantero, 480 alambres horizontales. Si se aplica un voltaje a uno de los alambres verticales y luego se hace pulsar uno de los horizontales, se puede alterar el voltaje en una posición definida de un pixel, haciendo que se oscurezca por un momento. Si se repite este pulso con el siguiente pixel y luego con el siguiente, se puede pintar una línea de barrido oscura, de forma análoga a como se hace en un CRT. Normalmente, toda la pantalla se pinta 60 veces por segundo para engañar el ojo y hacerlo creer que hay una imagen constante ahí, como se hace también en un CRT.

El otro esquema de uso muy difundido es la **pantalla de matriz activa**. Su costo es considerablemente mayor, pero da una mejor imagen y por ello está ganando terreno. En lugar de tener sólo dos juegos de alambres perpendiculares, tiene un diminuto elemento conmutador en cada posición de pixel en uno de los electrodos. Al encender y apagar estos elementos, puede crearse un patrón de voltaje arbitrario en toda la pantalla, y así dibujar un patrón de bits arbitrario.

Hasta ahora hemos descrito cómo funciona una pantalla monocromática. Baste decir que las pantallas a color se basan en los mismos principios generales que las monocromáticas, pero que los detalles son mucho más complicados. Se utilizan filtros ópticos para separar la luz blanca en componentes rojo, verde y azul en cada posición de pixel, de modo que puedan exhibirse de forma independiente. Cualquier color puede crearse con una superposición lineal de estos tres colores primarios.

Terminales de mapa de caracteres

Se usan comúnmente tres tipos de terminales: de mapa de caracteres, de mapa de bits y RS-232-C. Todas ellas pueden usar cualquier tipo de teclado, pero difieren en la forma en que la computadora se comunica con ellas y en el manejo de las salidas. A continuación describiremos brevemente cada clase.

En una computadora personal hay dos formas de organizar las salidas que se envían a la pantalla: un mapa de caracteres y un mapa de bits. La figura 2-33 muestra cómo se usa un mapa de caracteres para exhibir salidas en el monitor. (El teclado se trata como un dispositivo totalmente independiente.) En la tarjeta de comunicaciones en serie hay una porción de memoria, llamada **memoria de video**, además de algunos circuitos para acceder al bus y generar señales de video.

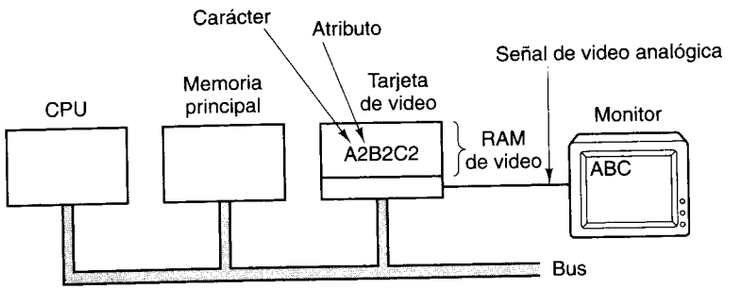


Figura 2-33. Salida en una terminal de una computadora personal.

Para exhibir caracteres, la CPU los copia en la memoria de video en bytes alternados. Cada carácter tiene asociado un **byte de atributo** que describe la forma en que debe exhibirse el carácter. Los atributos pueden incluir su color, intensidad, si está parpadeando o no, etc. Así, una imagen de pantalla de 25 × 80 caracteres requiere 4000 bytes de memoria de video, 2000 para los caracteres y 2000 para los atributos. Casi todas las tarjetas tienen más memoria, a fin de mantener varias imágenes de pantalla.

La tarea de la tarjeta de video consiste en traer una y otra vez caracteres de la RAM de video y generar la señal necesaria que se alimenta al monitor. Se trae toda una línea de caracteres a la vez para poder calcular las líneas de barrido individuales. Esta señal es del tipo analógica de alta frecuencia que controla el barrido del haz de electrones que pinta los caracteres en la pantalla. Puesto que la tarjeta genera una señal de video, el monitor debe estar a unos cuantos metros de la computadora a fin de evitar distorsiones.

Terminales de mapa de bits

Una variación de esta idea es ver la pantalla no como una matriz de caracteres 25 × 80, sino como una formación de elementos de imagen llamados **pixeles**. Cada pixel está encendido o

apagado, y representa un bit de información. En las computadoras personales la pantalla podría contener 640 × 480 pixeles, pero es más común encontrar 800 × 600 o más. En las estaciones de trabajo de ingeniería la pantalla normalmente tiene 1280 × 960 pixeles o más. Las terminales que usan un mapa de bits en lugar de un mapa de caracteres se llaman **terminales de mapa de bits**. Todas las tarjetas de video modernas pueden operar como terminales de mapa de caracteres o de mapa de bits, bajo el control de software.

Se utiliza la misma idea general que en la figura 2-33, sólo que la RAM de video se ve como un gran arreglo de bits. El software puede crear ahí cualquier patrón que desee, y se exhibirá instantáneamente. Para dibujar caracteres, el software podría decidir asignar, por ejemplo, un rectángulo de 9 por 14 a cada carácter y "encender" los bits necesarios para que aparezca el carácter. Esta técnica permite al software crear diversos tipos de letra y entremezclarlos a voluntad. Lo único que hace el hardware es exhibir el arreglo de bits. En el caso de pantallas a color, cada pixel requiere 8, 16 o 24 bits.

Las terminales de mapa de bits suelen usarse para presentar imágenes que contienen varias **ventanas**. Una ventana es un área de la pantalla utilizada por un programa. Con múltiples ventanas es posible tener varios programas ejecutándose al mismo tiempo, y que cada uno exhiba sus resultados con independencia de los demás.

Aunque las terminales de mapa de bits son muy flexibles, tienen dos desventajas importantes. Primera, requieren una cantidad considerable de RAM de video. Los tamaños más comunes en estos días son 640 × 480 (VGA), 800 × 600 (SVGA), 1024 × 768 (XVGA) y 1280 × 960. Cabe señalar que todas éstas tienen una relación de aspecto (anchura:altura) de 4:3, a fin de ajustarse a la relación vigente en los televisores. Para obtener color fiel, se requieren 8 bits para cada uno de los colores primarios, o sea 3 bytes/pixel. Así, una pantalla de 1024 × 768 requiere 2.3 MB de RAM de video.

En vista del tamaño de la memoria necesaria, algunas computadoras hacen concesiones y utilizan un número de 8 bits para indicar el color deseado. Este número se maneja como índice de una tabla en hardware, llamada **paleta de colores**, que contiene 256 entradas, cada una con un valor RGB de 24 bits. Un diseño así, llamado **color indizado**, reduce las necesidades de memoria RAM de video en 2/3 partes, pero sólo permite 256 colores en la pantalla a la vez. Por lo regular, cada ventana de la pantalla tiene su propio mapeo, pero al haber sólo una paleta de colores en hardware es común que si hay varias ventanas en la pantalla sólo la ventana activa aparezca con sus colores correctos.

La segunda desventaja de una pantalla de mapa de bits es su desempeño. Una vez que los programadores de aplicaciones se enteran de que pueden controlar cada pixel tanto en el espacio como en el tiempo, quieren hacerlo. Aunque es posible copiar datos de la RAM de video al monitor sin pasar por el bus de sistema principal, la transferencia de datos a la RAM de video sí emplea el bus de sistema. Para exhibir multimedia de pantalla completa a todo color en una pantalla de 1024 × 768 es preciso copiar 2.3 MB de datos en la RAM de video para cada cuadro. En el caso de video con pleno movimiento, se requiere una tasa de por lo menos 25 cuadros/s, para una tasa de datos total de 57.6 MB/s. Esta carga es mucho más que lo que el bus (E)ISA puede manejar, de modo que las tarjetas de video de alto desempeño para IBM PC necesitan ser tarjetas PCI, y aun así se requieren importantes concesiones.

Un problema de desempeño relacionado es cómo desplazar la imagen en la pantalla (*scroll*). Un método sería copiar todos los bits en software, pero esto impone una carga gigantesca a la CPU. Es natural que muchas tarjetas de video estén equipadas con hardware especial para desplazar partes de la pantalla modificando registros base en vez de por copiado.

Terminales RS-232-C

Docenas de compañías fabrican computadoras y muchas más fabrican terminales (sobre todo para mainframes). Con el fin de lograr que (casi) cualquier terminal pueda usarse con (casi) cualquier computadora, la Electronics Industries Association (EIA) creó una interfaz estándar computadora-terminal llamada **RS-232-C**. Cualquier terminal que reconozca la interfaz RS-232-C se puede conectar a cualquier computadora que también reconozca esa interfaz.

Las terminales RS-232-C tienen un conector estandarizado de 25 terminales. La norma RS-232-C define el tamaño y la forma físicos del conector, los niveles de voltaje y el significado de cada una de las señales de las agujas.

Cuando la computadora y la terminal están muy separadas, en muchos casos la única forma práctica de conectarlos es a través del sistema telefónico. Desafortunadamente, el sistema telefónico no puede transmitir las señales requeridas por la norma RS-232-C, y es necesario insertar un dispositivo llamado **módem** (**mo**dulador-**dem**odulador) entre la computadora y el teléfono y también entre la terminal y el teléfono, a fin de efectuar la conversión de las señales. Estudiaremos los módems en breve.

La figura 2-34 muestra la colocación de la computadora, los módems y la terminal cuando se usa una línea telefónica. Si la terminal está lo bastante cerca de la computadora como para usar cableado directo, no se usan módems, pero sí los mismos conectores y cables RS-232-C, aunque no se necesiten las terminales relacionadas con el control del módem.

Para comunicarse, la computadora y la terminal contienen un chip llamado **UART** (**transmisor-receptor universal asincrónico**, *Universal Asynchronous Receiver Transmitter*), además de lógica para acceder al bus. Para exhibir un carácter, la computadora lo trae de su memoria principal y lo presenta al UART, que entonces lo mete en el cable RS-232-C bit por bit. De hecho, el UART es un convertidor de paralelo a serie, ya que recibe un carácter completo (1 byte) en una sola operación y produce los bits uno por uno con cierta rapidez. Además, el UART añade un bit de inicio y un bit de paro a cada carácter para delimitar el principio y el final del carácter (a 110 bps, se usan dos bits de paro).

En la terminal, otro UART recibe los bits y reconstruye el carácter, que luego se exhibe en la pantalla. Las entradas del teclado de la terminal se someten a una conversión paralelo-serie en la terminal y el UART las reensambla en la computadora.

La norma RS-232-C define casi 25 señales, pero en la práctica sólo se usan unas cuantas (y casi todas esas pueden omitirse cuando la terminal se conecta directamente a la computadora sin módems). Las terminales 2 y 3 son para transmitir y recibir datos, respectivamente. Cada terminal maneja un flujo de bits en un sentido. Cuando se enciende la terminal o computadora, habilita (es decir, pone en 1) la señal de (Lista Terminal de Datos) para informar al módem que está encendida. Por su parte, el módem habilita la señal de Data Set Ready para

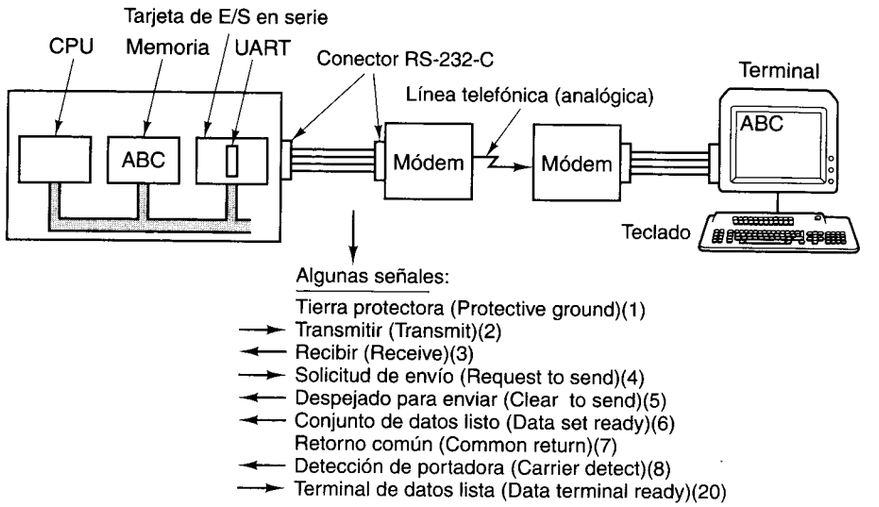


Figura 2-34. Conexión de una terminal RS-232-C con una computadora. Los números entre paréntesis en la lista de señales son los números de terminal.

indicar su presencia. Cuando la terminal o computadora quiere enviar datos, habilita la señal Request to Send para pedir permiso. Si el módem está dispuesto a conceder ese permiso, habilita la señal Clear to Send como respuesta. Otras terminales tienen diversas funciones de estado, prueba y temporización.

2.4.3 Ratones

Conforme pasa el tiempo, personas con cada vez menos conocimientos acerca de cómo funcionan las computadoras están usando estas máquinas. Las computadoras de la generación ENIAC sólo eran utilizadas por quienes las habían construido. En los cincuenta, sólo programadores profesionales con alto nivel de capacitación usaban computadoras. Hoy día las computadoras son utilizadas ampliamente por personas que necesitan realizar alguna tarea y no saben (ni quieren saber) mucho acerca de cómo funcionan o cómo se programan las computadoras.

En el pasado, casi todas las computadoras tenían interfaces de línea de comandos, en las que los usuarios tecleaban comandos. Puesto que muchas personas no eran especialistas en computación percibían las interfaces de línea de comandos como poco amables hacia el usuario, por no decir hostiles, algunos fabricantes de computadoras crearon interfaces de apuntar y hacer clic, como las de Macintosh y Windows. Para usar este modelo se requiere una forma de apuntar a la pantalla, y la forma más común de hacerlo es con un ratón.

Un **ratón** (*mouse*) es una pequeña caja de plástico que descansa sobre la mesa junto al teclado. Cuando el ratón se mueve sobre la mesa, también se mueve un pequeño puntero en la

pantalla, y ello permite a los usuarios apuntar a elementos de la pantalla. El ratón tiene uno, dos o tres botones en la parte superior, que permiten a los usuarios seleccionar opciones de menús. Se ha derramado mucha sangre como resultado de discusiones acerca de cuántos botones debe tener un ratón. Los usuarios poco sofisticados prefieren uno (es difícil oprimir el botón equivocado si sólo hay uno), pero a los sofisticados les gusta la potencia de múltiples botones para hacer infinidad de cosas.

Se han producido tres tipos de ratones: mecánicos, ópticos y optomecánicos. Los primeros ratones tenían dos ruedas de caucho que sobresalían por abajo y tenían sus ejes perpendiculares entre sí. Cuando el ratón se movía paralelo a su eje principal, una rueda giraba. Cuando se movía perpendicularmente a su eje principal, giraba la otra rueda. Cada rueda impulsaba un resistor variable (potenciómetro). Si se medían los cambios en la resistencia, era posible ver qué tanto había girado cada rueda y así calcular qué tanto se había movido el ratón en cada dirección. En años recientes este diseño ha sido reemplazado en buena medida por uno en el que se usa una esfera que sobresale ligeramente de la base, en lugar de ruedas. Éste se muestra en la figura 2-35.

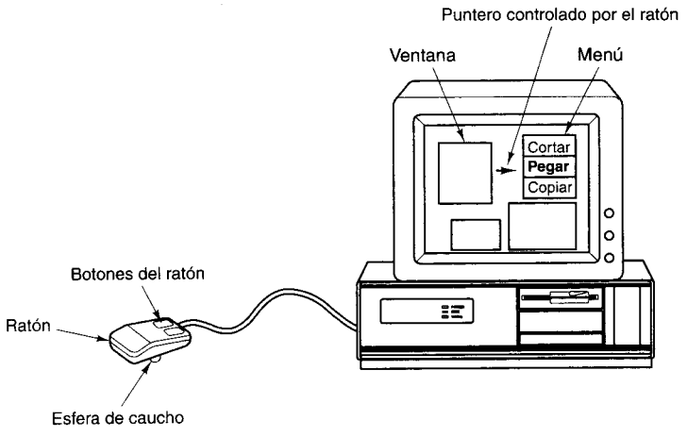


Figura 2-35. Uso del ratón para apuntar a opciones de un menú.

El segundo tipo de ratón es el óptico. Este tipo no tiene ruedas ni esfera. En vez de ello, tiene un **LED (diodo emisor de luz**, *Light Emitting Diode*) y un fotodetector en la base. El ratón óptico se utiliza sobre una base de plástico especial que contiene una cuadrícula de líneas muy juntas. A medida que el ratón se mueve sobre la cuadrícula, el fotodetector percibe el paso de las líneas por los cambios en la cantidad de luz del LED que se refleja. Unos circuitos internos del ratón cuentan el número de líneas de cuadrícula que se cruzan en cada dirección.

El tercer tipo de ratón es optomecánico. Al igual que el ratón mecánico más nuevo, tiene una esfera rodante que hace girar dos ejes perpendiculares entre sí. Los ejes están conectados a codificadores provistos de ranuras a través de las cuales puede pasar la luz. A medida que el ratón se mueve, los ejes giran y pulsos de luz inciden sobre los detectores cada vez que una ranura pasa entre un LED y su detector. El número de pulsos detectados es proporcional a la cantidad de movimiento.

Aunque los ratones se pueden configurar de varias maneras, un esquema común es hacer que el ratón envíe una secuencia de 3 bytes a la computadora cada vez que se mueva cierta distancia mínima (por ejemplo, 0.01 de pulgada), llamado en ocasiones **mickey**. Por lo regular, dichos caracteres llegan por una línea serial, bit por bit. El primer byte contiene un entero con signo que indica cuántas unidades se movió el ratón en la dirección x en los últimos 100 ms. El segundo byte da la misma información para la dirección y. El tercer byte contiene la situación actual de los botones del ratón. A veces se usan dos bytes para cada coordenada.

Software de bajo nivel de la computadora acepta esta información conforme llega y convierte los movimientos relativos enviados por el ratón en una posición absoluta. Luego, el software exhibe una flecha en la pantalla en la posición correspondiente al ratón. Cuando la flecha apunta al elemento correcto, el usuario hace clic con un botón del ratón y la computadora puede determinar cuál elemento se seleccionó, porque sabe en qué parte de la pantalla está la flecha.

2.4.4 Impresoras

Después de preparar un documento o de bajar una página de la World Wide Web, es común que los usuarios quieran imprimirlos, así que todas las computadoras se pueden equipar con una impresora. En esta sección describiremos algunos de los tipos más comunes de impresoras monocromáticas (es decir, blanco y negro) y a color.

Impresoras monocromáticas

El tipo de impresora más económico es la **impresora de matriz**, en la que una cabeza de impresión que contiene entre 7 y 24 agujas activables electromagnéticamente se mueve a lo largo de cada línea de impresión. Las impresoras de bajo costo tienen siete agujas para imprimir, digamos, 80 caracteres en una matriz de 5×7 a lo largo de la línea. De hecho, la línea de impresión consiste en siete líneas horizontales, cada una de las cuales consta de $5 \times 80 = 400$ puntos. Cada punto puede imprimirse o no, dependiendo de los caracteres que han sido seleccionados para tal efecto. En la figura 2-36(a) se ilustra la letra "A" impresa en una matriz de 5×7.

La calidad de impresión se puede incrementar empleando dos técnicas: usar más agujas, y hacer que los puntos se traslapen. En la figura 2-36(b) se muestra una "A" impresa con 24 agujas que producen puntos que se traslapan. Por lo regular se requieren varias pasadas sobre cada línea de barrido para producir puntos traslapados, por lo que un aumento en la calidad generalmente implica una tasa de impresión más lenta. Casi todas las impresoras de matriz pueden operar de varios modos, que ofrecen diferentes equilibrios entre calidad de impresión y rapidez.

Las impresoras de matriz son económicas (sobre todo en términos de consumibles) y muy confiables, pero son lentas, ruidosas y malas para imprimir gráficos. Estas impresoras tienen tres usos principales en los sistemas modernos. Primero, se usan mucho para imprimir

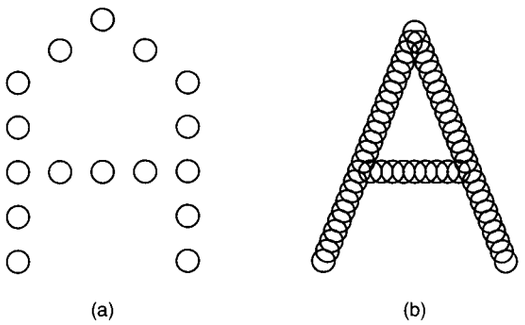


Figura 2-36. (a) La letra "A" en una matriz de 5 × 7. (b) La letra "A" impresa con 24 agujas que se traslapan.

en formatos grandes (> 30 cm) preimpresos. Segundo, son buenas para imprimir en pequeños trozos de papel, como los recibos de caja registradora, boletas de transacción de cajero automático o tarjeta de crédito, pases de abordar de líneas aéreas, etc. Tercero, generalmente son la tecnología más económica para imprimir en formas continuas de varias copias que tienen papel carbón intercalado.

Para la impresión casera de bajo costo, las **impresoras de inyección de tinta** son las favoritas. La cabeza de impresión móvil, que lleva un cartucho de tinta, se mueve horizontalmente a lo ancho del papel mientras rocía tinta con sus diminutas boquillas. Dentro de cada boquilla, una pequeña gota de tinta se calienta eléctricamente más allá de su punto de ebullición, hasta que hace explosión. La única dirección en que la tinta puede moverse es hacia la salida de la boquilla para chocar contra el papel. Luego la boquilla se enfría y el vacío que se produce succiona otra gotita de tinta. La rapidez de la impresora está limitada por la rapidez con que puede repetirse el ciclo de ebullición/enfriamiento. Las impresoras de inyección de tinta suelen tener definiciones de 300 **dpi** (**puntos por pulgada**, *dots per inch*) a 720 dpi, aunque también las hay de 1440 dpi. Estas impresoras son económicas, silenciosas y tienen buena calidad, pero también son lentas, usan cartuchos de tinta caros y producen impresiones saturadas de tinta.

Tal vez el avance más interesante en cuanto a impresión desde que Johann Gutenberg inventó los tipos móviles en el siglo xv, es la **impresora láser**. Este dispositivo combina una imagen de alta calidad, excelente flexibilidad, buena velocidad y costo moderado en un mismo periférico. Las impresoras láser emplean casi la misma tecnología que las fotocopiadoras; de hecho, muchas compañías producen aparatos que combinan el copiado y la impresión (y a veces también un faccímil).

La tecnología básica se ilustra en la figura 2-37. El corazón de la impresora es un cilindro de precisión giratorio (o, en algunos sistemas del extremo superior, una banda). Al principio de cada ciclo de página, el cilindro se carga hasta cerca de 1000 volts y se recubre con un material fotosensible. Luego la luz de un láser se mueve a lo largo del cilindro de forma muy similar a como un haz de electrones se mueve en un CRT, sólo que en lugar de efectuar la desviación horizontal con la aplicación de un voltaje se usa un espejo octogonal giratorio

para barrer el cilindro a lo largo. El haz de luz se modula para producir un patrón de puntos claros y oscuros. Los puntos en los que el haz incide pierden su carga eléctrica.

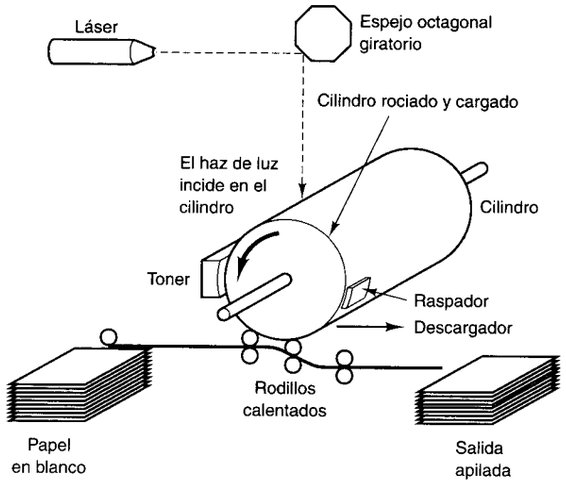


Figura 2-37. Funcionamiento de una impresora láser.

Una vez que se ha pintado una línea de puntos, el cilindro gira una fracción de grado para que pueda pintarse la siguiente línea. En algún momento, la primera línea de puntos llega al tóner, un depósito de polvo negro sensible a las cargas electrostáticas. El tóner es atraído hacia los puntos que todavía conservan su carga, y así forman una imagen visual de esa línea. Un poco más adelante en el trayecto de transporte, el cilindro recubierto con tóner se oprime contra el papel y transfiere a éste el polvo negro. Luego el papel pasa entre rodillos calientes que fusionan el tóner con el papel de forma permanente, lo que fija la imagen. Al continuar su rotación, el cilindro pierde su carga y un raspador elimina cualquier residuo de tóner, antes de que el cilindro se vuelva a cargar y recubrir para la siguiente página.

Sobra decir que este proceso es una combinación en extremo compleja de ciencia física, ciencia química, ingeniería mecánica e ingeniería óptica. No obstante, varios proveedores producen ensambles completos, llamados **máquinas de impresión**, que los fabricantes de impresoras láser combinan con sus propios circuitos electrónicos y software para crear una impresora completa. Los circuitos constan de una CPU y varios megabytes de memoria para contener el mapa de bits de una página completa y numerosos tipos de letra, algunos de los cuales son proporcionados con la impresora, mientras que otros se bajan de una fuente externa, como la computadora. Casi todas las impresoras aceptan comandos que describen las páginas a imprimir (en lugar de aceptar simplemente mapas de bits preparados por la CPU principal). Estos comandos se dan en lenguajes como PCL de HP y PostScript de Adobe.

Las impresoras láser de 600 dpi o mejores pueden imprimir fotografías en blanco y negro con una calidad razonable, pero la tecnología tiene sus bemoles. Considere una fotografía digitalizada a 600 dpi que se va a imprimir en una impresora de 600 dpi. La imagen digitalizada contiene 600 × 600 pixeles/pulg^2, cada uno de los cuales consiste en un valor de gris que va de 0

(blanco) a 255 (negro). La impresora también puede imprimir 600 dpi, pero cada pixel impreso es negro (tóner presente) o bien blanco (sin tóner). No es posible imprimir tonos de gris.

La solución usual para imprimir imágenes que tienen tonos de gris es utilizar **medios tonos**, como se hace en los carteles que se imprimen comercialmente. La imagen se divide en celdas de medio tono, cada una de las cuales contiene por lo regular 6×6 pixeles. Cada celda puede contener entre 0 y 36 pixeles negros. El ojo percibe una celda con muchos pixeles como más oscura que una que tiene menos pixeles. Los valores de gris dentro del intervalo de 0 a 255 se representan dividiendo este intervalo en 37 zonas. Los valores del 0 al 6 están en la zona 0, los valores del 7 al 13 están en la zona 1, y así (la zona 36 es un poco más pequeña que las demás porque 256 no es divisible exactamente entre 37). Cada vez que se encuentra un valor de gris en la zona 0, su celda de medio tono en el papel se deja en blanco, como se ilustra en la figura 2-38(a). Un valor de zona 1 se imprime como un pixel negro. Un valor de zona 2 se imprime como dos pixeles negros, como se muestra en la figura 2-38(b). En la figura 2-38(c)-(f) se muestran otros valores de zona. Desde luego, si tomamos una fotografía digitalizada a 600 dpi y la convertimos en medios tonos con esta técnica, reduciremos la definición efectiva a 100 celdas/pulgada. Ésta se denomina **frecuencia de pantalla de medio tono** y por convención se mide en **lpi** (**líneas por pulgada**).

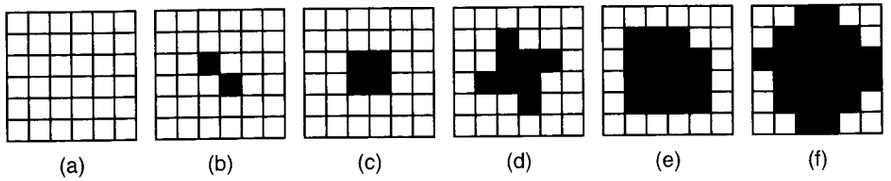


Figura 2-38. Puntos de medio tono para diversos intervalos de escala de grises. (a) 0-6. (b) 14-20. (c) 28-34. (d) 56-62. (e) 105-111. (f) 161-167.

Impresoras a color

Las imágenes a color pueden verse de dos maneras: como luz transmitida o como luz reflejada. Las imágenes de luz transmitida, como las que se producen en los monitores de CRT, se forman a partir de la superposición lineal de los tres colores primarios aditivos, rojo, verde y azul. Las imágenes de luz reflejada, como las fotografías a color y las ilustraciones de las revistas, absorben ciertas longitudes de onda de la luz y reflejan el resto. Estas imágenes se forman por la superposición lineal de los tres colores primarios sustractivos, turquesa o cian (se absorbe todo el rojo), amarillo (se absorbe todo el azul) y magenta (se absorbe todo el verde). En teoría, podemos producir cualquier color mezclando tinta turquesa, amarilla y magenta, pero en la práctica es difícil obtener tintas lo bastante puras como para que absorban toda la luz y produzcan un verdadero negro. Por esta razón, casi todos los sistemas de impresión a color emplean cuatro tintas: turquesa, amarilla, magenta y negra. Estos sistemas se llaman **impresoras CYMK** por las iniciales de los colores en inglés: *Cyan, Yellow, Magenta y blacK* (se usa la *K* y no la *B* para no confundir el blacK (negro) con *Blue*, azul). Los monitores, en cambio, usan luz transmitida y el sistema RGB para producir colores.

El conjunto total de colores que una pantalla o impresora puede producir es su **gama**. Ningún dispositivo tiene una gama que coincida con el mundo real, puesto que en el mejor de los casos cada color viene en 256 intensidades, lo que da sólo 16,777,216 colores discretos. Imperfecciones de la tecnología reducen aun más el total, y los colores restantes no siempre están distribuidos de manera uniforme en el espectro de color. Además, la percepción del color tiene mucho que ver con el funcionamiento de los bastones y los conos de la retina humana, no sólo con los aspectos físicos de la luz.

Una consecuencia de las observaciones anteriores es que convertir una imagen a color que se ve bien en la pantalla, en una imagen impresa idéntica, no es nada trivial. Algunos de los problemas son:

1. Los monitores a color usan luz transmitida; las impresoras a color usan luz reflejada.
2. Los CRT producen 256 intensidades por color; las impresoras a color deben emplear medios tonos.
3. Los monitores tienen un fondo oscuro; el papel tiene un fondo claro.
4. Las gamas RGB y CYMK son diferentes.

Lograr que las imágenes a color impresas coincidan con el mundo real (o siquiera con las imágenes en pantalla) requiere calibración del dispositivo, software avanzado y considerable experiencia por parte del usuario.

Existen cinco tecnologías de uso común para la impresión a color, todas ellas basadas en el sistema CYMK. En el extremo inferior están las impresoras de inyección de tinta a color. Éstas funcionan igual que las monocromáticas, pero con cuatro cartuchos de tinta (para C, Y, M y K) en lugar de uno. Estas impresoras producen buenos resultados con gráficos a color y resultados aceptables con fotografías, a un costo moderado (las impresoras no cuestan mucho, pero los cartuchos de tinta sí).

Para obtener resultados óptimos hay que usar tinta y papel especiales. Existen dos tipos de tinta. Las **tintas basadas en colorantes** que consisten en sustancias de coloración disueltas en un fluido portador. Éstas producen colores brillantes y fluyen fácilmente. Su desventaja principal es que se destiñen cuando se exponen a luz ultravioleta, como la contenida en la luz solar. Las **tintas basadas en pigmentos** contienen partículas sólidas de pigmento suspendidas en un fluido portador que se evapora del papel y deja atrás el pigmento. Éstas no se destiñen con el tiempo pero no son tan brillantes como las tintas basadas en colorantes y las partículas de pigmento tienden a taponar las boquillas, que requieren una limpieza periódica. Se necesita papel recubierto o satinado para imprimir fotografías. Estos tipos de papel están diseñados especialmente para retener las pequeñas gotas de tinta e impedir que se extiendan.

En el siguiente escalón arriba de las impresoras de inyección de tinta están las **impresoras de tinta sólida**. Éstas aceptan cuatro bloques sólidos de una tinta cerosa especial que se funden para introducirlos en depósitos de tinta caliente. Los tiempos de preparación de estas impresoras pueden llegar a los 10 minutos, mientras se funden los bloques de tinta. La tinta caliente se rocía sobre el papel, donde se solidifica y fusiona con el papel cuando éste se comprime entre dos rodillos duros.

El tercer tipo de impresora a color es la impresora láser a color. Ésta funciona como su pariente monocromático, excepto que se depositan imágenes C, Y, M y K individuales que se

transfieren a un rodillo utilizando cuatro tóners distintos. Puesto que generalmente se produce con anticipación el mapa de bits completo, una imagen de 1200 × 1200 dpi para una página que contiene 80 pulgadas cuadradas requiere 115 millones de pixeles. Con 4 bits/pixel, la impresora necesita 55 MB sólo para el mapa de bits, sin contar la memoria para los procesadores internos, tipos de letra, etc. Este requisito hace que las impresoras láser a color sean costosas, pero la impresión es rápida, la calidad es alta y las imágenes son duraderas.

La cuarta clase de impresora a color es la **impresora de cera**. Ésta tiene una cinta ancha de cera de cuatro colores segmentada en bandas del tamaño de una página. Miles de elementos calefactores funden la cera a medida que el papel pasa debajo de ella. La cera se fusiona con el papel en forma de pixeles empleando el sistema CYMK. Las impresoras de cera solían ser la principal tecnología de impresión a color, pero están siendo sustituidas por las otras clases, cuyos consumibles son más económicos.

El quinto tipo de impresora a color es la **impresora de sublimación de colorante**. Aunque la palabra sublimación tiene connotaciones freudianas, es el término científico que describe la conversión de un sólido en gas sin pasar por el estado líquido. El hielo seco (dióxido de carbono congelado) es un material conocido que se sublima. En una impresora de sublimación de colorante, un portador que contiene los colorantes CYMK pasa por una cabeza de impresión térmica que contiene cientos de elementos calefactores programables. Los colorantes se vaporizan instantáneamente y son absorbidos por un papel especial que está cerca. Cada elemento calefactor puede producir 256 temperaturas distintas. Cuanto más alta es la temperatura, más colorante se deposita y más intenso es el color. A diferencia de las demás impresoras a color, es posible obtener colores casi continuos para cada pixel, de modo que no se requieren medios tonos. Casi todas las impresoras de fotografías instantáneas emplean el proceso de sublimación de colorantes para producir imágenes fotográficas altamente realistas en papel especial (y caro).

2.4.5 Módems

Con el crecimiento del uso de las computadoras en los últimos años, es común que una computadora necesite comunicarse con otra. Por ejemplo, mucha gente tiene una computadora personal en casa que usa para comunicarse con su computadora en el trabajo, con un proveedor de servicio de Internet o con un sistema de "banco en su casa". Todas estas aplicaciones utilizan el teléfono para efectuar la comunicación.

Sin embargo, una línea telefónica ordinaria no es apropiada para transmitir señales de computadora, que generalmente representan un 0 como 0 volts y un 1 como 3 a 5 volts, como se muestra en la figura 2-39(a). Las señales de dos niveles sufren una distorsión considerable cuando se transmiten por una línea telefónica de grado de voz, y esto da pie a errores de transmisión. En cambio, una señal de onda senoidal pura con una frecuencia de 1000 a 2000 Hz, llamada **portadora**, se puede transmitir con relativamente poca distorsión, y este hecho se aprovecha en casi todos los sistemas de telecomunicaciones.

Puesto que las pulsaciones de una onda senoidal son totalmente predecibles, una onda senoidal pura no transmite información. Sin embargo, si se varía la amplitud, la frecuencia o la fase, se puede transmitir una secuencia de unos y ceros, como se muestra en la figura 2-39.

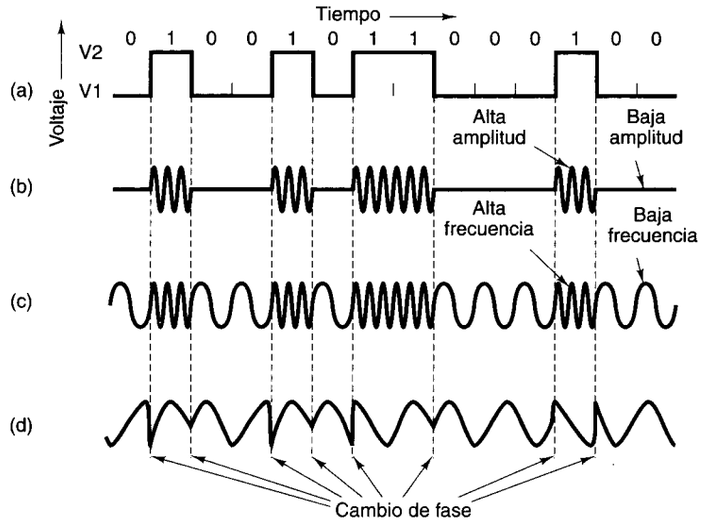


Figura 2-39. Transmisión del número binario 01001011000100 por una línea telefónica bit por bit. (a) Señal de dos niveles. (b) Modulación de amplitud. (c) Modulación de frecuencia. (d) Modulación de fase.

Este proceso se llama **modulación**. En la **modulación de amplitud** [vea la figura 2-39(b)] se usan dos niveles de voltaje distintos para el 0 y para el 1. Una persona que escuchara datos digitales transmitidos con una tasa de datos muy baja escucharía un ruido fuerte cuando se transmite un 1 y silencio cuando se transmite un 0.

En la **modulación de frecuencia** [vea la figura 2-39(c)] el nivel de voltaje es constante pero la frecuencia de la portadora es diferente para el 1 y para el 0. Una persona que escuchara datos digitales de frecuencia modulada escucharía dos tonos, uno para el 0 y otro para el 1. Esta técnica también se conoce como **modulación por desplazamiento de frecuencia** (FSK, *Frequency Shift Keying*).

En la **modulación de fase** simple [vea la figura 2-39(d)], la amplitud y la frecuencia no cambian, pero la fase de la portadora se invierte 180 grados cuando los datos cambian de 0 a 1 o de 1 a 0. En los sistemas de modulación de fase más avanzados, al principio de cada intervalo de tiempo indivisible la fase de la portadora se desplaza abruptamente 45, 135, 225 o 315 grados, lo que permite transmitir dos bits en cada intervalo de tiempo. Esto se conoce como codificación por fase **dibit**. Por ejemplo, un cambio de fase de 45 grados podría representar 00 y uno de 135 grados, 01. También existen otros esquemas para transmitir tres o más bits por intervalo de tiempo. El número de intervalos (es decir, el número de posibles cambios de señal por segundo) se llama tasa de **bauds** (*baud rate*). Con dos o más bits por intervalo, la tasa de bits excederá la tasa de bauds. Muchas personas confunden estos dos términos.

Si los datos por transmitir consisten en una serie de caracteres de ocho bits, sería deseable tener una conexión capaz de transmitir ocho bits simultáneamente, es decir, ocho pares de hilos. Puesto que las líneas telefónicas de grado de voz sólo ofrecen un canal, los bits se

deben enviar en serie, uno tras otro (o en grupos de dos si se está empleando codificación dibit). El dispositivo que acepta caracteres de una computadora en forma de señales de dos niveles, bit por bit, y transmite los bits en grupos de uno o dos, en forma de amplitud modulada, frecuencia modulada o fase modulada, es el módem. Para marcar el principio y el final de cada carácter, un carácter de ocho bits normalmente se envía precedido por un bit de inicio y seguido de un bit de paro, para un total de 10 bits.

El módem transmisor envía los bits individuales de un carácter a intervalos de tiempo equidistantes. Por ejemplo, 9600 bauds implica un cambio de señal cada 104 μs. Un segundo módem en el extremo receptor convierte una portadora modulada en un número binario. Puesto que los bits llegan al receptor a intervalos regulares, una vez que el módem receptor ha determinado el inicio del carácter su reloj le dice cuándo debe muestrear la línea para leer los bits que llegan.

Los módems modernos operan con tasas de datos entre 28,800 bits/s (bps) y 57,600 bits/s, y normalmente a tasas de bauds mucho más bajas. Emplean una combinación de técnicas para enviar varios bits por baud, modulando la amplitud, la frecuencia y la fase. Casi todos ellos son **dúplex (*full duplez*)**, lo que implica que pueden transmitir en ambas direcciones al mismo tiempo (empleando diferentes frecuencias). Los módems o líneas de transmisión que sólo pueden transmitir en una dirección a la vez (como una vía férrea sencilla que puede manejar trenes que van al norte o trenes que van al sur, pero no al mismo tiempo) se llaman **semidúplex (*half duplex*)**. Las líneas que sólo pueden transmitir en una dirección son **simplex**.

ISDN

A principios de la década de los ochenta las PTT europeas (administraciones de correo, teléfono y telégrafo) crearon un estándar para telefonía digital llamado **ISDN (red digital de servicios integrados**, *Integrated Services Digital Network*). La intención original era que las personas de edad avanzada pudieran tener alarmas en sus casas conectadas a oficinas de monitoreo centrales, y muchas otras aplicaciones extrañas que nunca se hicieron realidad. Ellas promovieron intensamente la idea, pero sin mucho éxito. Luego, la World Wide Web apareció de la noche a la mañana y la gente estaba exigiendo a gritos acceso digital con gran ancho de banda a Internet. Así, de repente, ISDN había descubierto una superaplicación (aunque no había sido contemplada por sus diseñadores). La ISDN también se popularizó en Estados Unidos y otros países.

Cuando un cliente de una compañía telefónica se suscribe a ISDN, se sustituye la antigua línea analógica por una digital. (De hecho, no se cambia la línea misma, sólo el equipo en sus extremos.) La nueva línea comprende dos canales digitales independientes de 64,000 bits/s cada uno, más un canal de señalización de 16,000 bits/s. Existen equipos que combinan los tres canales en un solo canal digital de 144,000 bps. Para las empresas está disponible una línea ISDN de 30 canales.

ISDN no sólo es más rápida que un canal analógico, sino que también permite establecer conexiones en un tiempo que normalmente no es mayor que un segundo, ya no requiere un módem analógico, y es mucho más confiable (menos errores) que una línea analógica. Además, cuenta con varias funciones y opciones adicionales que las líneas analógicas no siempre ofrecen.

En la figura 2-40 se muestra la estructura de una conexión ISDN. Lo que la compañía portadora proporciona es un conducto de bits digital que se limita a transportar bits. El significado de los bits es cuestión del transmisor y del receptor. La interfaz entre el equipo del cliente y el equipo de la portadora es el dispositivo NT1, con la interfaz T en un lado y la interfaz U en el otro. En Estados Unidos los clientes deben comprar su propio dispositivo NT1. En muchos países europeos deben rentarlo a la compañía portadora.

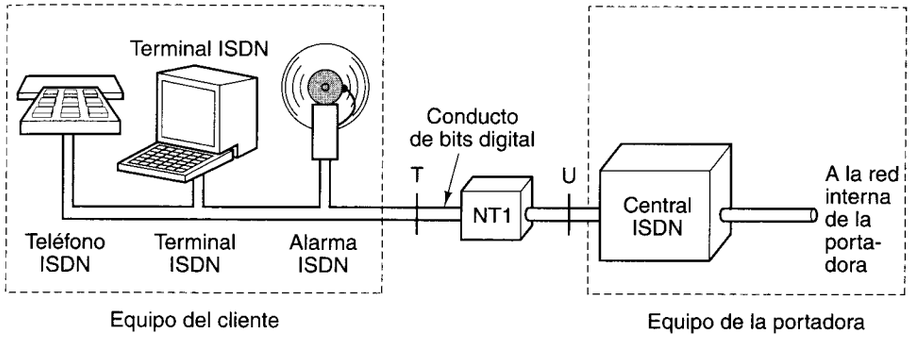


Figura 2-40. ISDN para uso casero.

2.4.6 Códigos de caracteres

Cada computadora usa un conjunto de caracteres. Como mínimo, este conjunto incluye las 26 letras mayúsculas del inglés, las 26 letras minúsculas, los dígitos del 0 al 9 y un conjunto de símbolos especiales, como espacio, punto, signo menos, coma y retorno de carro.

Para transferir estos caracteres a la computadora, se asigna un número a cada uno: por ejemplo, a = 1, b = 2, ..., z = 26, + = 27, – = 28. La correspondencia entre caracteres y enteros se llama **código de caracteres**. Es indispensable que dos computadoras que se comunican empleen el mismo código, pues de lo contrario no podrán entenderse. Para ello se han creado normas, y a continuación examinaremos dos de las más importantes.

ASCII

Un código de amplio uso se llama **ASCII (Código Estándar Americano para Intercambio de Información**, *American Standard Code for Information Interchange*). Cada carácter ASCII tiene 7 bits, lo que permite representar un total de 128 caracteres. La figura 2-41 muestra el código ASCII. Los códigos 0 a 1F (hexadecimal) son caracteres de control y no se imprimen.

Muchos de los caracteres de control ASCII desempeñan funciones durante la transmisión de datos. Por ejemplo, un mensaje podría consistir en un carácter SOH (*Start of Header*, inicio de cabecera), una cabecera, un carácter STX (*Start of Text*, inicio de texto), el texto

Hex	Nombre	Significado	Hex	Nombre	Significado
0	NUL	Nulo	10	DLE	Escape de enlace de datos
1	SON	Inicio de cabecera	11	DC1	Control de dispositivo 1
2	STX	Inicio de texto	12	DC2	Control de dispositivo 2
3	ETX	Fin de texto	13	DC3	Control de dispositivo 3
4	EOT	Fin de transmisión	14	DC4	Control de dispositivo 4
5	ENQ	Pregunta	15	NAK	Reconocimiento negativo
6	ACK	Reconocimiento	16	SYN	Tiempo muerto sincrónico
7	BEL	Alarma	17	ETB	Fin de bloqueo de transmisión
8	BS	Retroceso	18	CAN	Cancelar
9	HT	Tabulación horizontal	19	EM	Fin de medio
A	LF	Salto de linea	1A	SUB	Sustituto
B	VT	Tabulación vertical	1B	ESC	Escape
C	FF	Salto de página	1C	FS	Separador de archivos
D	CR	Retorno de carro	1D	GS	Separador de grupos
E	SO	Desactivar cambio	1E	RS	Separador de registros
F	SI	Activar cambio	1F	US	Separador de unidades

Hex	Car	Hex	Car	Hex	Car	Hex	Car	Hex	Car	Hex	Car
20	(Espacio)	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figura 2-41. El conjunto de caracteres ASCII.

mismo, un carácter ETX (*End of Text*, fin de texto) y luego un carácter EOT (*End of Transmission*, fin de transmisión). En la práctica los mensajes que se envían por líneas telefónicas y redes tienen un formato muy diferente, y es por ello que ya casi no se usan los códigos de control de transmisión ASCII.

Los caracteres imprimibles ASCII no tienen mayor ciencia. Incluyen las letras mayúsculas y minúsculas, los dígitos, signos de puntuación y unos cuantos símbolos matemáticos.

UNICODE

La industria de las computadoras creció principalmente en Estados Unidos, y esto dio lugar al conjunto de caracteres ASCII. El ASCII está bien para el inglés pero no tanto para otros idiomas. El francés necesita acentos (por ejemplo, système); el alemán necesita signos diacríticos (por ejemplo, für), etc. Algunos lenguajes europeos tienen unas cuantas letras que no están incluidas en ASCII, como la ß alemana y la ø danesa. Algunos idiomas tienen alfabetos totalmente distintos (como el ruso y el árabe), y unos cuantos idiomas ni siquiera tienen un alfabeto (como el chino). A medida que las computadoras se propagaron a todos los rincones del mundo y los proveedores de software comenzaron a querer vender productos en países en los que la mayoría de los usuarios no habla inglés, se hizo necesario un conjunto de caracteres distinto.

El primer intento de extender ASCII fue IS 646, que añadió otros 128 caracteres a ASCII para convertirlo en un código de ocho bits llamado **Latin-1**. Los caracteres adicionales eran en su mayor parte letras latinas con acentos y signos diacríticos. El siguiente intento fue IS 8859, que introdujo el concepto de **página de código**, un conjunto de 256 caracteres para un idioma o grupo de idiomas específico. IS 8859-1 es Latin-1. IS 8859-2 maneja los idiomas eslavos basados en el latín (como el checo, el polaco y el húngaro). IS 8859-3 contiene los caracteres necesarios para el turco, maltés, esperanto y gallego, etc. El problema con el enfoque de página de código es que el software tiene que saber en qué página está, es imposible combinar idiomas de diferentes páginas, y el esquema no cubre el chino ni el japonés.

Un grupo de compañías de computación decidió resolver este problema formando un consorcio para crear un sistema nuevo, llamado **UNICODE**, y hacer que se proclamara como Norma Internacional (IS 10646). Actualmente varios lenguajes de programación (como Java), algunos sistemas operativos (como Windows NT) y muchas aplicaciones reconocen UNICODE. Es probable que la aceptación de este sistema aumente cada vez más a medida que la industria de la computación se vuelve global.

La idea básica en que se basa UNICODE es asignar a cada carácter y símbolo un valor único y permanente de 16 bits, llamado **punto de código**. No se usan caracteres de múltiples bytes ni secuencias de escape. Al hacer que cada símbolo tenga 16 bits la escritura de software se simplifica mucho.

Con símbolos de 16 bits, UNICODE tiene 65,356 puntos de código. Puesto que los idiomas del mundo emplean colectivamente unos 200,000 símbolos, los puntos de código son un recurso escaso que debe asignarse con mucho cuidado. Ya se ha asignado cerca de la mitad de los puntos de código, y el consorcio UNICODE examina continuamente propuestas para asignar el resto. A fin de acelerar la aceptación de UNICODE, el consorcio utilizó sagazmente Latin-1 para los puntos de código del 0 al 255, lo que facilita la conversión entre ASCII y UNICODE.

Para evitar el desperdicio de puntos de código, cada signo diacrítico tiene su propio punto de código, y corresponde al software combinar los signos diacríticos con sus vecinas para formar nuevos caracteres.

El espacio de puntos de código se divide en bloques, cada uno de los cuales es un múltiplo de 16 puntos de código. Cada alfabeto principal en UNICODE tiene una secuencia de zonas consecutivas. Algunos ejemplos (y el número de códigos de punto asignados) son latín (336), griego (144), cirílico (256), armenio (96), hebreo (112), devanagari (128), gurmukhi (128), oriya (128), telugu (128) y kannada (128). Cabe señalar que a cada uno de estos idiomas se les ha asignado más puntos de códigos que las letras que tienen. Esta decisión se tomó en parte porque muchos idiomas tienen varias formas para cada letra. Por ejemplo, cada letra del español tiene dos formas, minúscula y MAYÚSCULA. Algunos idiomas tienen tres o más formas, posiblemente dependiendo de si la letra está al principio, en medio o al final de una palabra.

Además de estos alfabetos, se han asignado puntos de código a signos diacríticos (112) y de puntuación (112), subíndices y superíndices (48), símbolos de divisas (48) y matemáticos (256), formas geométricas (96) y ornamentos tipográficos (192).

Después de éstos vienen los símbolos requeridos para el chino, japonés y coreano. Primero hay 1024 símbolos fonéticos (como katakana y bopomofo) y luego los ideogramas Han unificados (20,992) empleados en chino y japonés, y las sílabas Hangul coreanas (11,156).

A fin de que los usuarios puedan inventar caracteres especiales para propósitos específicos, se han asignado 6400 puntos de código para uso local.

Si bien UNICODE resuelve muchos problemas relativos a la internacionalización, no resuelve todos los problemas del mundo (ni intenta hacerlo). Por ejemplo, si bien el alfabeto latino está en orden, los ideogramas Han no están en el orden del diccionario; por consiguiente, un programa en español puede examinar "gato" y "perro" y ordenarlos alfabéticamente con sólo comparar el valor UNICODE de su primer carácter. Un programa japonés necesita tablas externas para determinar cuál de dos símbolos precede al otro en el diccionario.

Otro problema es que continuamente aparecen palabras nuevas. Hace 50 años nadie hablaba de applets, ciberespacio, gigabytes, láseres, módems o videocintas. La adición de palabras nuevas al español no requiere nuevos puntos de código, pero añadirlas al japonés sí. Además de nuevas palabras técnicas, existe demanda para añadir al menos 20,000 nombres personales y de lugares nuevos (casi todos chinos). Los ciegos piensan que el Braille debe estar incluido, y todo tipo de grupos con intereses especiales quieren los puntos de código que, según ellos, por derecho les corresponden. El consorcio UNICODE estudia todas las nuevas propuestas y emite un dictamen.

UNICODE emplea el mismo punto de código para caracteres que son casi idénticos pero tienen diferente significado, o que chinos y japoneses escriben de forma ligeramente distinta (como si un procesador de textos en español siempre escribiera "caza" como "casa", porque suenan igual). Algunas personas ven esto como una optimación para ahorrar valiosos puntos de código; otras creen que es una manifestación del imperialismo cultural anglosajón (y usted que creía que asignar valores de 16 bits a los caracteres nada tenía que ver con la política). Para empeorar las cosas, un diccionario japonés completo tiene 50,000 kanji (sin incluir nombres), así que con sólo 20,992 puntos de código disponibles para los ideogramas Han, fue preciso tomar decisiones. No todos los japoneses creen que un consorcio de compañías de computación, aunque algunas de ellas sean japonesas, sea el foro ideal para tomar esas decisiones.

2.5 RESUMEN

Los sistemas de computación se construyen con tres tipos de componentes: procesadores, memorias y dispositivos de E/S. La tarea de un procesador es buscar instrucciones una por una en una memoria, decodificarlas y ejecutarlas. El ciclo de búsqueda-decodificación-ejecución siempre puede describirse como un algoritmo y, de hecho, a veces es ejecutado por un intérprete en software que se ejecuta en un nivel más bajo. A fin de mejorar la velocidad, muchas computadoras actuales tienen uno o más filas de procesamiento, o tienen un diseño superescalar con varias unidades funcionales que operan en paralelo.

Cada vez son más comunes los sistemas con varios procesadores. Las computadoras paralelas incluyen los procesadores de arreglos, en los que la misma operación se ejecuta con varios conjuntos de datos al mismo tiempo, los multiprocesadores, en los que varias CPU comparten una misma memoria, y las multicomputadoras, en las que varias computadoras, cada una con su propia memoria, se comunican mediante el envío de mensajes.

Las memorias pueden clasificarse como primarias o secundarias. La memoria primaria se usa para contener el programa que se está ejecutando. Su tiempo de acceso es corto –unas cuantas decenas de nanosegundos como máximo– e independiente de la dirección a la que se está accediendo. Las cachés reducen este tiempo de acceso aún más. Algunas memorias están equipadas con códigos de corrección de errores para mejorar la confiabilidad.

Las memorias secundarias, en contraste, tienen tiempos de acceso mucho más largos (milisegundos o más) y que dependen de la ubicación de los datos que se están leyendo o escribiendo. Las cintas, discos magnéticos y discos ópticos son las memorias secundarias más comunes. Hay muchas variedades de discos magnéticos: discos flexibles, discos Winchester, discos IDE, discos SCSI y RAID. Los discos ópticos incluyen los CD-ROM, los CD-R y los DVD.

Los dispositivos de E/S sirven para transferir información de y hacia la computadora, y están conectados al procesador y a la memoria con uno o más buses. Como ejemplos podemos citar las terminales, los ratones, las impresoras y los módems. Casi todos los dispositivos de E/S utilizan el código de caracteres ASCII, aunque UNICODE está adquiriendo cada vez más aceptación a medida que la industria de las computadoras se vuelve global.

PROBLEMAS

1. Considere el funcionamiento de una máquina con el camino de datos de la figura 2-2. Suponga que cargar los registros de entrada de la ALU toma 5 ns, para ejecutar la operación la ALU toma 10 ns y almacenar el resultado en un registro toma 5 ns. ¿Qué número máximo de MIPS podrá ejecutar esta máquina si no hay fila de procesamiento?
2. ¿Para qué sirve el paso 2 de la lista de la sección 2.1.2? ¿Qué sucedería si se omitiera este paso?
3. En la computadora 1, todas las instrucciones tardan 10 ns en ejecutarse. En la computadora 2, todas tardan 5 ns. ¿Puede decir con certeza que la computadora 2 es más rápida? Explique.

4. Suponga que está diseñando una computadora de un solo chip que se usará en sistemas incorporados. El chip tendrá toda su memoria en el chip y ésta operará a la misma velocidad que la CPU sin retraso por acceso. Examine cada uno de los principios vistos en la sección 2.1.4 y diga si siguen siendo tan importantes (suponiendo que se desea un buen desempeño).

5. ¿Sería posible añadir cachés a los procesadores de la figura 2-8(b)? Si es posible, ¿qué problema habría que resolver primero?

6. Cierto cálculo es altamente secuencial; es decir, cada paso depende del que lo precede. ¿Qué sería más apropiado para este cálculo, un arreglo de procesadores o un procesador con fila de procesamiento? Explique.

7. Para competir con el reciente invento de la imprenta, cierto monasterio medieval decidió producir en masa libros de bolsillo escritos a mano juntando un gran número de escribanos en un gran salón. El monje director gritaba la primera palabra del libro a producir y todos los escribanos la copiaban. Luego el monje director gritaba la segunda palabra y todos los escribanos la copiaban. Este proceso se repetía hasta que se había leído en voz alta y copiado todo el libro. ¿A cuál de los sistemas de procesador paralelo que vimos en la sección 2.1.6 se parece más este sistema?

8. Conforme bajamos por la jerarquía de memoria de cinco niveles que vimos en el texto, el tiempo de acceso aumenta. Haga una conjetura razonable de la relación entre el tiempo de acceso de un disco óptico y el de la memoria de registros. Suponga que el disco ya está en línea.

9. Calcule la tasa de datos del ojo humano con base en la información siguiente: El campo visual consiste en unos 10^6 elementos (pixeles). Cada pixel puede reducirse a una superposición de los tres colores primarios, cada uno de los cuales tiene 64 intensidades. La definición temporal es de 100 ms.

10. La información genética de todos los seres vivos se codifica en moléculas de DNA. Una molécula de DNA es una secuencia lineal de los cuatro nucleótidos básicos: *A*, *C*, *G* y *T*. El genoma humano contiene aproximadamente 3×10^9 nucleótidos en forma de unos 100,000 genes. ¿Qué capacidad de información total (en bits) tiene el genoma humano? ¿Qué capacidad de información (en bits) tiene un gen típico?

11. ¿Cuáles de las siguientes memorias son posibles? ¿Cuáles son razonables? explique.

    a. Dirección de 10 bits, 1024 celdas, tamaño de celda de 8 bits

    b. Dirección de 10 bits, 1024 celdas, tamaño de celda de 12 bits

    c. Dirección de 9 bits, 1024 celdas, tamaño de celda de 10 bits

    d. Dirección de 11 bits, 1024 celdas, tamaño de celda de 10 bits

    e. Dirección de 10 bits, 10 celdas, tamaño de celda de 1024 bits

    f. Dirección de 1024 bits, 10 celdas, tamaño de celda de 10 bits

12. Los sociólogos pueden obtener tres posibles respuestas a una pregunta de encuesta típica como "¿Cree usted en el Ratoncito Pérez?": sí, no, y no tengo opinión al respecto. Con esto en mente, la Compañía de Computadoras Sociomagnéticas decidió construir una computadora para procesar datos de encuestas. Esta computadora tiene una memoria trinaria; es decir, cada byte (¿tryte?) consiste en 8 trits, y un trit contiene un 0, un 1 o un 2. ¿Cuántos trits se necesitan para almacenar un número de 6 bits? Dé una expresión para el número de trits necesarios para almacenar *n* bits.

13. Cierta computadora puede equiparse con 268,435,456 bytes de memoria. ¿Por qué habría de escoger un fabricante un número tan extraño, en lugar de un número fácil de recordar como 250,000,000?

14. Derive un código Hamming de paridad par para los dígitos del 0 al 9.

15. Derive un código para los dígitos 0 al 9 cuya distancia de Hamming sea 2.

16. En un código de Hamming, algunos bits se "desperdician" en el sentido de que se usan para verificación y no para información. ¿Cuál es el porcentaje de bits desperdiciados en mensajes cuya longitud total (datos + bits de verificación) es $2^n - 1$? Evalúe esta expresión numéricamente para valores de *n* de 3 a 10.

17. Los errores de transmisión en las líneas telefónicas a menudo ocurren en ráfagas (muchos bits consecutivos con error) en lugar de individualmente. Puesto que el código de Hamming básico sólo puede corregir errores individuales dentro de un carácter, no sirve de nada si una ráfaga de ruido hace que *n* bits consecutivos sean erróneos. Diseñe un método para transmitir texto ASCII por una línea en la que las ráfagas de ruido pudieran hacer que hasta 100 bits consecutivos fueran erróneos. Suponga que el intervalo mínimo entre dos ráfagas de ruido es de miles de caracteres. *Sugerencia:* piense cuidadosamente en el orden de transmisión de los bits.

18. ¿Cuánto tiempo toma leer un disco que tiene 800 cilindros, cada uno de los cuales contiene cinco pistas de 32 sectores? Primero deben leerse todos los sectores de la pista 0, comenzando con el sector 0, luego todos los sectores de la pista 1 comenzando con el sector 0, y así sucesivamente. El tiempo de rotación es de 20 ms, y una búsqueda tarda 10 ms entre cilindros adyacentes y 50 ms en el peor de los casos. La conmutación entre las pistas de un cilindro puede ser instantánea.

19. El disco que se ilustra en la figura 2-19 tiene 64 sectores/pista y una tasa de rotación de 7200 rpm. Calcule la tasa de transferencia sostenida del disco a lo largo de una pista.

20. Una computadora tiene un bus con un tiempo de ciclo de 25 ns, durante el cual puede leer o escribir una palabra de 32 bits de la memoria. La computadora tiene un disco Ultra-SCSI que utiliza el bus y opera a 40 MB/s. La CPU normalmente busca y ejecuta una instrucción de 32 bits cada 25 ns. ¿Qué tanto retrasa el disco a la CPU?

21. Imagine que está escribiendo la parte de administración de disco de un sistema operativo. Lógicamente, usted representa el disco como una secuencia de bloques, del 0 en la parte interior hasta algún valor máximo en la exterior. A medida que se crean archivos, usted tiene que asignar sectores libres, y podría hacerlo desde el interior hacia afuera o desde el exterior hacia adentro. ¿Importa cuál estrategia escoja? Explique su respuesta.

22. El direccionamiento LBA emplea 24 bits para direccionar un sector. ¿Cuál es el disco más grande que puede manejar?

23. RAID nivel 3 puede corregir errores de un solo bit utilizando sólo una unidad de disco de paridad. ¿De qué sirve RAID nivel 2? Después de todo, sólo puede corregir un error también, y necesita más unidades de disco para hacerlo.

24. ¿Qué capacidad exacta (en bytes) tiene un CD-ROM modo 2 que contiene los 74 minutos de datos estándar?

25. Para quemar un CD-R, el láser debe emitir pulsos a alta velocidad. Al operar a velocidad 4x en el modo 1, ¿qué duración tiene un pulso en nanosegundos?

26. Para almacenar 133 minutos de video en un DVD de un solo lado y una sola capa se requiere una cantidad apreciable de compresión. Calcule el factor de compresión requerido. Suponga que se

dispone de 3.5 GB de espacio para la pista de video, que la definición de imagen es de 720×480 pixeles con color de 24 bits, y que las imágenes se exhiben a razón de 30 cuadros/s.

27. La tasa de transferencia entre una CPU y su memoria es varios órdenes de magnitud mayor que la tasa de transferencia de E/S mecánica. ¿Cómo puede esta diferencia causar ineficiencias? ¿Cómo puede aliviarse?

28. Una terminal de mapa de bits tiene una pantalla de 1024×768. La pantalla se redibuja 75 veces cada segundo. ¿Qué duración tiene el pulso correspondiente a un pixel?

29. Un fabricante anuncia que su terminal a color de mapa de bits puede exhibir 2^{24} colores distintos. Sin embargo, el hardware sólo tiene un byte para cada pixel. ¿Cómo puede hacerse esto?

30. Con cierto tipo de letra, una impresora láser monocromática puede imprimir 50 líneas de 80 caracteres por página. Un carácter típico ocupa un cuadro de 2 mm × 2 mm, y cerca del 25% de esta área es tóner. El resto está en blanco. La capa de tóner tiene un espesor de 25 micras. El cartucho de tóner de la impresora mide $25 \times 8 \times 2$ cm. ¿Cuántas páginas pueden imprimirse con un cartucho de tóner?

31. Cuando texto ASCII con paridad par se transmite asincrónicamente a razón de 2880 caracteres/s por un módem de 28,800 bps, ¿qué porcentaje de los bits recibidos contienen realmente datos (no gasto extra)?

32. La Hi-Fi Modem Company acaba de diseñar un nuevo módem con modulación de frecuencia que utiliza 16 frecuencias en lugar de sólo 2. Cada segundo se divide en n intervalos de tiempo iguales, cada uno de los cuales contiene uno de los 16 tonos posibles. ¿Cuántos bits por segundo puede transmitir este módem si emplea transmisión sincrónica?

33. Estime cuántos caracteres, contando los espacios, contiene un libro de texto de computación típico. ¿Cuántos bits se necesitan para codificar un libro en ASCII con paridad? ¿Cuántos CD-ROM se necesitan para almacenar una biblioteca de ciencias de la computación que tiene 10,000 libros? ¿Cuántos DVD de dos lados y capa dual se necesitan para la misma biblioteca?

34. Decodifique el siguiente texto binario en ASCII: 1001001 0100000 1001100 1001111 1010110 1000101 0100000 1011001 1001111 1010101 0101110.

35. Escriba un procedimiento *hamming*(*ascii*, *codificado*) que convierta los siete bits de orden bajo de *ascii* en una palabra de código entera de 11 bits almacenada en *codificado*.

36. Escriba una función *distancia*(*código*, *n*, *k*) que acepte como entrada un arreglo *código* de *n* caracteres de *k* bits cada uno y devuelva como salida la distancia del conjunto de caracteres.

3

EL NIVEL DE LÓGICA DIGITAL

En la base de la jerarquía de la figura 1-2 encontramos el nivel de lógica digital, el hardware real de la computadora. En este capítulo examinaremos muchos aspectos de la lógica digital como cimientos para el estudio de niveles más altos en capítulos subsecuentes. Este tema está en la frontera entre las ciencias de la computación y la ingeniería eléctrica, pero el material está explicado de manera clara, por lo que no se necesita experiencia previa en hardware ni en ingeniería para entenderlo.

Los elementos básicos con que se construyen todas las computadoras digitales son asombrosamente sencillos. Iniciaremos nuestro estudio examinando estos elementos básicos y también el álgebra especial de dos valores (álgebra booleana) que se usa para analizarlos. Luego examinaremos algunos circuitos fundamentales que pueden construirse a base de compuertas en combinaciones sencillas, incluidos circuitos que realizan operaciones aritméticas. El siguiente tema es la forma de combinar compuertas para almacenar información, es decir, cómo se organizan las memorias. Después pasaremos al tema de las CPU, y en especial a la forma en que se comunican con la memoria y los dispositivos periféricos. En una sección posterior del capítulo se discutirán muchos ejemplos de aplicación en la industria.

3.1 COMPUERTAS Y ÁLGEBRA BOOLEANA

Podemos construir circuitos digitales a partir de un número reducido de elementos primitivos combinándolos de un sinnúmero de maneras. En las secciones que siguen describiremos esos elementos primitivos, mostraremos cómo pueden combinarse, y presentaremos una potente técnica matemática que puede servir para analizar su comportamiento.

3.1.1 Compuertas

Un circuito digital es en el que sólo están presentes dos valores lógicos. Por lo regular, una señal entre 0 y 1 volt representa un valor (digamos, el 0 binario) y una señal entre 2 y 5 volts representa el otro valor (digamos, el 1 binario). No se permiten voltajes fuera de estos dos intervalos. Diminutos dispositivos electrónicos, llamados **compuertas**, pueden calcular diversas funciones con estas señales de dos valores. Las compuertas constituyen la base de hardware sobre la que se construyen todas las computadoras digitales.

Los detalles del funcionamiento interno de las compuertas rebasan el alcance de este libro, y pertenecen al **nivel de dispositivo**, que está abajo de nuestro nivel 0. No obstante, haremos aquí una muy breve digresión para examinar la idea básica, que no es difícil. Toda la lógica digital moderna se apoya en última instancia en el hecho de que es posible hacer que un transistor se comporte como un conmutador binario muy rápido. En la figura 3-1(a) se muestra un transistor bipolar (el círculo) incorporado a un circuito muy sencillo. Este transistor tiene tres conexiones con el mundo exterior: el **colector**, la **base** y el **emisor**. Cuando el voltaje de entrada, V_{in} es menor que cierto valor crítico, el transistor se apaga y actúa como una resistencia infinita. Esto hace que la salida del circuito, V_{out}, adquiera un valor cercano a V_{CC}, un voltaje regulado externamente, que suele ser +5 volts para este tipo de transistor. Cuando V_{in} excede el valor crítico, el transistor se enciende y actúa como un alambre, lo que hace que V_{out} se baje a tierra (por convención, 0 volts).

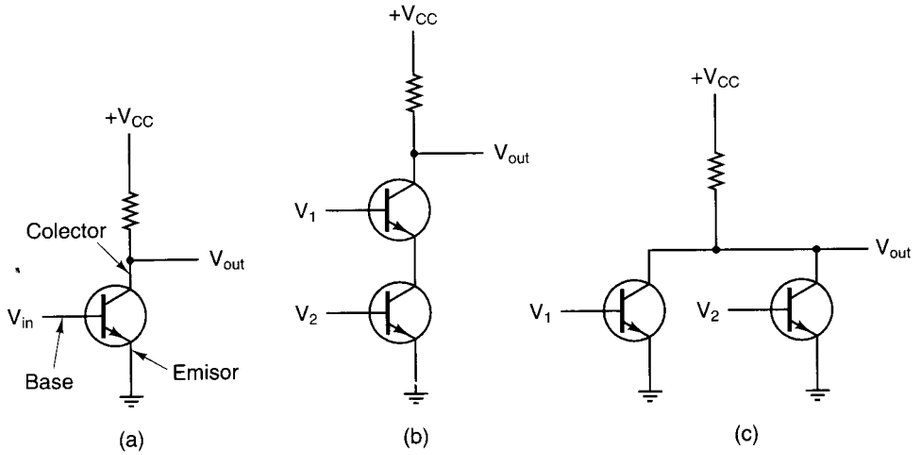


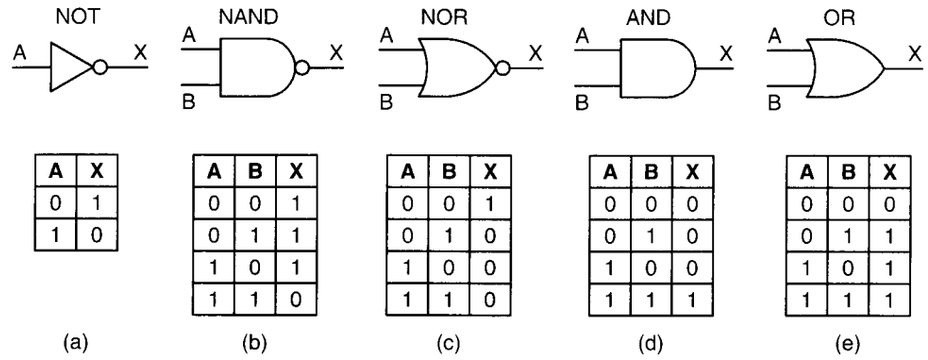
Figura 3-1. (a) Transistor inversor. (b) Compuerta NAND. (c) Compuerta NOR.

El aspecto importante es que cuando V_{in} es bajo, V_{out} es alto, y viceversa. Así, este circuito es un inversor que convierte un 0 lógico en un 1 lógico, y un 1 lógico en un 0 lógico. El resistor (la línea en zigzag) es necesario para limitar la cantidad de corriente que el transistor absorbe, y evitar que se queme. El tiempo necesario para conmutar de un estado al otro suele ser de unos cuantos nanosegundos.

En la figura 3-1(b) dos transistores se han conectado en serie. Si V_1 y V_2 son altos, ambos transistores conducirán y V_{out} bajará. Si cualquiera de las entradas es baja, el transistor correspondiente se apagará y la salida será alta. En otras palabras, V_{out} será bajo si y sólo si tanto V_1 como V_2 son altos.

En la figura 3-1(c) los dos transistores están conectados en paralelo, no en serie. En esta configuración, si cualquiera de las entradas es alta, el transistor correspondiente se encenderá y ocasionará que la salida vaya a tierra. Si ambas entradas son bajas, la salida se mantendrá alta.

Estos tres circuitos, o sus equivalentes, forman las tres compuertas básicas, y se llaman compuertas NOT, NAND y NOR, respectivamente. Las compuertas NOT también se conocen como **inversores**; aquí usaremos los dos términos indistintamente. Si ahora adoptamos la convención de que "alto" (V_{CC} volts) es un 1 lógico, y que "bajo" (tierra) es un 0 lógico, podremos expresar el valor de la salida en función de los valores de las entradas. Los símbolos que se usan para representar estas tres compuertas se muestran en la figura 3-2(a)-(c), junto con el comportamiento funcional de cada circuito. En estas figuras, A y B son entradas y X es la salida. Cada renglón especifica la salida para una combinación distinta de las entradas.



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Figura 3-2. Símbolos y comportamiento funcional de las cinco compuertas básicas.

Si la señal de salida de la figura 3-1(b) se alimenta en un circuito inversor, obtenemos otro circuito que es exactamente opuesto a la compuerta NAND, a saber, un circuito cuya salida es 1 si y sólo si ambas entradas son 1. Un circuito así se llama compuerta AND; su símbolo y descripción funcional se dan en la figura 3-2(d). De forma similar, la compuerta NOR puede conectarse a un inversor para dar un circuito cuya salida es 1 si cualquiera de las entradas es 1, pero 0 si ambas entradas son 0. El símbolo y la descripción funcional de este circuito, llamado compuerta OR, se dan en la figura 3-2(e). Los círculos pequeños que se incluyen en los símbolos para el inversor, la compuerta NAND y la compuerta NOR se llaman **burbujas de inversión (compuertas universales)**, y se usan también en otros contextos para indicar una señal invertida.

Las cinco compuertas de la figura 3-2 son los principales bloques de construcción del nivel de lógica digital. Por la explicación anterior debe quedar claro que las compuertas NAND y NOR requieren dos transistores cada una, mientras que las compuertas AND y OR requieren tres.

Por esta razón, muchas computadoras se basan en las compuertas NAND y NOR en lugar de las más conocidas AND y OR. (En la práctica, todas las compuertas se implementan de forma un tanto distinta, pero las NAND y NOR siguen siendo más sencillas que las AND y OR. Vale la pena señalar que las compuertas pueden tener más de dos entradas. En principio, una compuerta NAND, por ejemplo, puede tener un número arbitrario de entradas, pero en la práctica es inusitado que haya más de ocho entradas.

Aunque el tema de la construcción de las compuertas pertenece al nivel de dispositivos, nos gustaría mencionar las principales familias de tecnología de fabricación porque continuamente se hace referencia a ellas. Las dos tecnologías principales son la **bipolar** y la **MOS** (Metal Óxido Semiconductor). Los principales tipos bipolares son **TTL** (lógica transistor-transistor), que ha sido el caballito de batalla de la electrónica digital durante muchos años, y **ECL** (lógica de emisor acoplado, *Emitter-Coupled Logic*), que se usa cuando se requiere una operación a muy alta velocidad.

Las compuertas MOS son más lentas que las TTL y ECL pero requieren mucho menos potencia y ocupan mucho menos espacio, lo que permite encapsular grandes cantidades de compuertas. Existen muchas variedades de MOS, que incluyen PMOS, NMOS y CMOS. Si bien los transistores MOS tienen una construcción diferente de la de los transistores bipolares, su capacidad para funcionar como interruptores electrónicos es la misma. Casi todas las CPU y las memorias modernas emplean tecnología CMOS, que opera a +3.3 volts. Esto es todo lo que trataremos sobre el nivel de dispositivos. Los lectores interesados en ahondar en el estudio de este nivel pueden consultar las lecturas que se sugieren en el capítulo 9.

3.1.2 Álgebra booleana

Para describir los circuitos que pueden construirse combinando compuertas se requiere un nuevo tipo de álgebra, una en la que las variables y funciones sólo puedan adoptar los valores 0 y 1. Semejante álgebra se denomina **álgebra booleana**, así llamada por su descubridor, el matemático inglés George Boole (1815-1864). En términos estrictos, nos estamos refiriendo realmente a un tipo específico de álgebra booleana, un **álgebra de conmutación**, pero el término "álgebra booleana" se usa tan ampliamente para referirse al "álgebra de conmutación" que no vamos a hacer la distinción aquí.

Así como hay funciones en el álgebra "ordinaria" (la de bachillerato), también hay funciones en el álgebra booleana. Una función booleana tiene una o más variables de entrada y produce un resultado que depende sólo de los valores de dichas variables. Podemos definir una función sencilla, f, diciendo que $f(A)$ es 1 si A es 0 y es 0 si A es 1. Ésta es la función NOT de la figura 3-2(a).

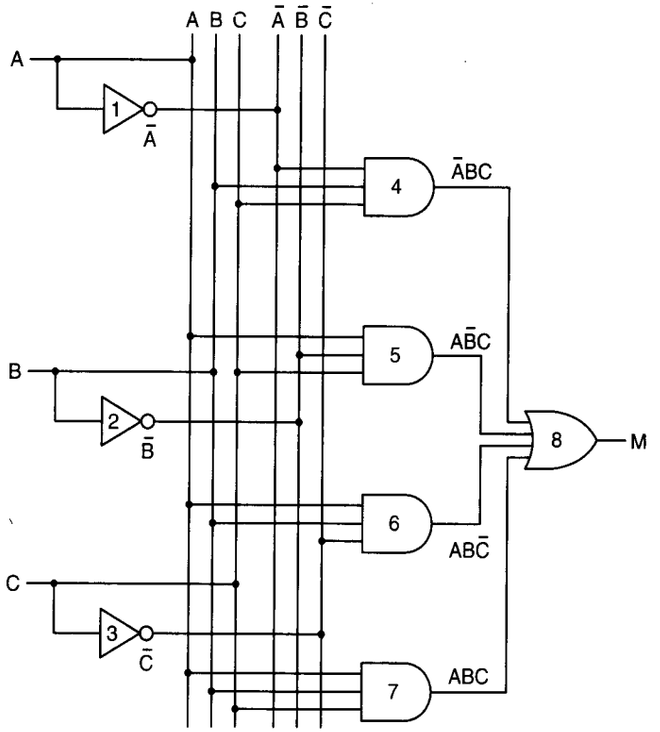
Puesto que una función booleana de n variables sólo tiene 2^n posibles combinaciones de los valores de entrada, la función puede describirse totalmente con una tabla de 2^n renglones, cada uno de las cuales indica el valor de la función para una combinación distinta de los valores de entrada. Semejante tabla se llama **tabla de verdad**. Todas las tablas de la figura 3-2 son ejemplos de tablas de verdad. Si convenimos en enumerar siempre las filas de una tabla de verdad en orden numérico (base 2), es decir, para dos variables en el orden 00, 01, 10 y 11, la función podrá describirse totalmente con el número binario de 2^n bits que se obtiene leyendo

verticalmente la columna del resultado de la tabla de verdad. Así, NAND es 1110, NOR es 1000, AND es 0001 y OR es 0111. Obviamente, sólo existen 16 funciones booleanas de dos variables, que corresponden a las 16 cadenas de resultado de 4 bits posibles. En contraste, el álgebra ordinaria tiene un número infinito de funciones de dos variables, ninguna de las cuales puede describirse dando una tabla de salidas para todas las posibles entradas porque cada variable puede adoptar cualquiera de un número infinito de valores posibles.

En la figura 3-3(a) se muestra la tabla de verdad para una función booleana de tres variables: $M = f(A, B, C)$. Esta función es la función de mayoría lógica, es decir, es 0 si la mayor parte de sus entradas son 0, y 1 si la mayor parte de sus entradas son 1. Aunque cualquier función booleana puede especificarse cabalmente dando su tabla de verdad, a medida que el número de variables aumenta esta notación se vuelve cada vez menos manejable. Por ello, se utiliza otra notación.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Figura 3-3. (a) Tabla de verdad de la función de mayoría de tres variables. (b) Circuito para (a).

Para entender de dónde sale esta notación, observe que cualquier función booleana puede especificarse indicando cuáles combinaciones de variables de entrada dan una salida con valor 1. Para la función de la figura 3-3(a), hay cuatro combinaciones de variables que hacen

que M sea 1. Por convención, colocamos una barra o testa sobre una variable de entrada para indicar que su valor se invierte. La ausencia de la barra implica que la variable no se invierte. Además, utilizaremos la multiplicación implícita o un punto para indicar la función AND booleana y + para denotar la función OR booleana. Así, por ejemplo, $A\overline{B}C$ adopta el valor 1 sólo cuando $A = 1$, $B = 0$ y $C = 1$. También, $A\overline{B} + B\overline{C}$ es 1 sólo cuando ($A = 1$ y $B = 0$) o ($B = 1$ y $C = 0$). Las cuatro filas de la figura 3-3(a) que producen bits 1 en la salida son: $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ y ABC. La función, M, es verdadera (o sea, 1) si es verdadera cualquiera de estas cuatro condiciones; por tanto, podemos escribir

$$M = \overline{A}BC + A\overline{B}C + ABC + AB\overline{C}$$

y ésta será una forma compacta de dar la tabla de verdad. Así, una función de n variables puede describirse dando una "suma" de cuando más 2^n términos "producto" de n variables. Esta formulación tiene gran importancia porque, como veremos en breve, nos lleva directamente a una implementación de la función empleando compuertas estándar.

Es importante tener presente la distinción entre una función booleana abstracta y su implementación con un circuito electrónico. Una función booleana consta de variables, como A, B y C, y operadores booleanos como AND, OR y NOT. Una función booleana se describe dando una tabla de verdad o una función booleana como

$$F = A\overline{B}C + AB\overline{C}$$

Una función booleana puede implementarse con un circuito electrónico (a menudo de muchas formas distintas) utilizando señales que representen las variables de entrada y salida, y compuertas como AND, OR y NOT. Generalmente usaremos la notación AND, OR y NOT para referirnos a los operadores booleanos, y AND, OR y NOT para referirnos a las compuertas, pero en muchos casos esto es ambiguo.

3.1.3 Implementación de funciones booleanas

Como ya lo afirmamos, la formulación de una función booleana como una suma de hasta 2^n términos producto nos lleva directamente a una posible implementación. Utilizando la figura 3-3 como ejemplo, vemos cómo se logra esa implementación. En la figura 3-3(b) las entradas A, B y C aparecen a la izquierda, y la función de salida, M, aparece a la derecha. Puesto que se necesitan complementos (inversos) de las variables de entrada, se generan sacando una derivación de cada entrada y haciendo pasar estas derivaciones por los inversores rotulados 1, 2 y 3. Para que la figura sea más o menos clara, hemos dibujado seis líneas verticales, tres conectadas a las variables de entrada y tres conectadas a sus complementos. Estas líneas son una fuente cómoda de entradas para las compuertas subsecuentes. Por ejemplo, A es una entrada de las compuertas 5, 6 y 7. En un circuito real estas compuertas probablemente se conectarían directamente a A sin usar alambres "verticales" intermedios.

El circuito contiene cuatro compuertas AND, una para cada término de la ecuación de M (es decir, una para cada renglón de la tabla de verdad que tiene 1 bit en la columna de resultado). Cada compuerta AND calcula un renglón de la tabla de verdad, como se indica. Por último, se obtiene el OR de todos los términos producto para obtener el resultado final.

El circuito de la figura 3-3(b) sigue una convención que usaremos una y otra vez en todo el libro: cuando dos líneas se cruzan, no hay una conexión implícita a menos que haya un punto grueso en la intersección. Por ejemplo, la salida de la compuerta 3 cruza las seis líneas verticales pero sólo está conectada a $\overline{C}$. Advertimos al lector que algunos autores emplean otras convenciones.

El ejemplo de la figura 3-3 nos muestra claramente cómo podemos implementar un circuito para cualquier función booleana:

1. Escribir la tabla de verdad para la función.
2. Incluir inversores para generar el complemento de cada entrada.
3. Dibujar una compuerta AND para cada término que tiene un 1 en la columna de resultado.
4. Conectar las compuertas AND a las entradas apropiadas.
5. Alimentar la salida de todas las compuertas AND a una compuerta OR.

Aunque hemos mostrado cómo puede implementarse cualquier función booleana empleando compuertas NOT, AND y OR, a menudo es conveniente implementar los circuitos utilizando un solo tipo de compuerta. Por fortuna, es fácil convertir los circuitos generados por el algoritmo anterior en una forma NAND pura o NOR pura. Para efectuar tal conversión, lo único que necesitamos es una forma de implementar NOT, AND y OR empleando un solo tipo de compuerta. El renglón superior de la figura 3-4 muestra cómo pueden implementarse estas tres compuertas empleando sólo compuertas NAND; el renglón inferior muestra cómo hacerlo usando sólo compuertas NOR. (Estos métodos son sencillos, pero no son los únicos.)

Una forma de implementar una función booleana usando sólo compuertas NAND o sólo compuertas NOR es seguir primero el procedimiento anterior para construirla con NOT, AND y OR. Luego se sustituyen las compuertas de varias entradas con circuitos equivalentes que sólo tengan compuertas de dos entradas. Por ejemplo, $A + B + C + D$ pueden calcularse como $(A + B) + (C + D)$, utilizando tres compuertas OR de dos entradas. Por último, las compuertas NOT, AND y OR se sustituyen por los circuitos de la figura 3-4.

Aunque este procedimiento no nos lleva a circuitos óptimos, en el sentido de tener un número mínimo de compuertas, sí nos demuestra que siempre es posible encontrar una solución. Se dice que las compuertas NAND y NOR son **completas (o universales)** porque cualquier función booleana puede calcularse empleando cualquiera de ellas. Ninguna otra compuerta tiene esta propiedad, y esto es otra razón por las que a menudo se les prefiere como bloques de construcción de circuitos.

3.1.4 Equivalencia de circuitos

Los diseñadores de circuitos a menudo tratan de reducir el número de compuertas en sus productos a fin de reducir el costo de los componentes, el espacio que ocupan en las tarjetas de circuitos impresos, el consumo de electricidad, etc. Para reducir la complejidad de un

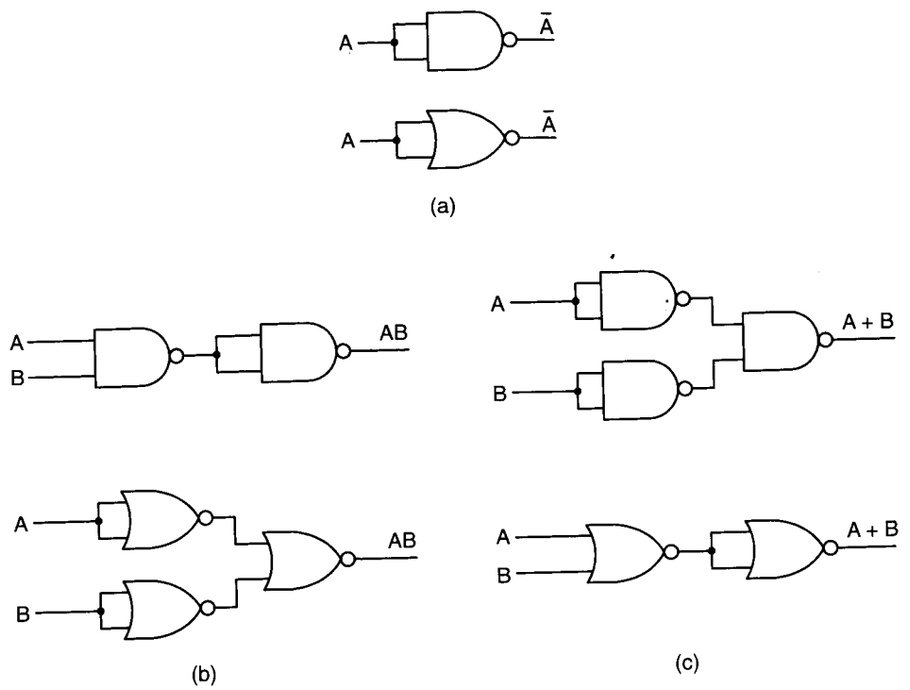


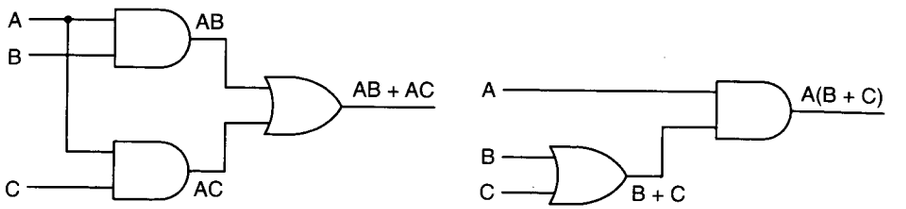
Figura 3-4. Construcción de compuertas (a) NOT, (b) AND y (c) OR utilizando sólo compuertas NAND o sólo compuertas NOR.

circuito, el diseñador debe encontrar otro circuito que calcule la misma función que el original pero lo haga con menos compuertas (o tal vez con compuertas más sencillas, por ejemplo, compuertas de dos entradas en lugar de cuatro). En la búsqueda de circuitos equivalentes, el álgebra booleana puede ser una herramienta valiosa.

Como ejemplo del uso del álgebra booleana, considere el circuito y la tabla de verdad para $AB + AC$ que se muestra en la figura 3-5(a). Aunque no hemos hablado de ello, muchas de las reglas del álgebra ordinaria también se cumplen en el álgebra booleana. En particular, $AB + AC$ puede factorizarse para dar $A(B + C)$ empleando la ley distributiva. La figura 3-5(b) muestra el circuito y la tabla de verdad para $A(B + C)$. Puesto que dos funciones son equivalentes si y sólo si tienen la misma salida para todas las posibles entradas, si examinamos las tablas de verdad de la figura 3-5 es fácil ver que $A(B + C)$ es equivalente a $AB + AC$. A pesar de esta equivalencia, es obvio que el circuito de la figura 3-5(b) es mejor que el de la figura 3-5(a) porque contiene menos compuertas.

En general, un diseñador de circuitos parte de una función booleana y luego le aplica las leyes del álgebra booleana en un intento por encontrar una función equivalente más sencilla. El circuito puede construirse a partir de la función final.

Para utilizar este enfoque, necesitamos algunas identidades del álgebra booleana. La figura 3-6 muestra algunas de las principales. Resulta interesante que cada ley tiene dos



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figura 3-5. Dos funciones equivalentes. (a) $AB + AC$. (b) $A(B + C)$.

formas, una de las cuales es el **dual** de la otra. Si intercambiamos AND y OR y también 0 y 1, cualquier forma puede producirse a partir de la otra. Todas las leyes pueden demostrarse fácilmente construyendo sus tablas de verdad. Con excepción de la ley de DeMorgan, la ley de absorción y la forma AND de la ley distributiva, los resultados son más o menos intuitivos. La ley de DeMorgan puede extenderse a más de dos variables, por ejemplo, $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$

La ley de DeMorgan sugiere una notación alternativa. En la figura 3-7(a) se muestra la forma AND para indicar negación mediante burbujas de inversión, tanto en la entrada como en la salida. Así, una compuerta OR con entradas invertidas equivale a una compuerta NAND. Si examinamos la figura 3-7(b), la forma dual de la ley de DeMorgan, debe ser obvio que una compuerta NOR puede dibujarse como una compuerta AND con entradas invertidas. Si negamos ambas formas de la ley de DeMorgan, llegamos a la figura 3-7(c) y (d), que muestra representaciones equivalentes de las compuertas AND y OR. Existen símbolos análogos para las formas de múltiples variables de la ley de DeMorgan (por ejemplo, una compuerta NAND de n entradas se convierte en una compuerta OR con n entradas invertidas).

Utilizando las identidades de la figura 3-7, y las identidades análogas para compuertas con varias entradas, es fácil convertir la representación de suma de productos de una tabla de verdad a una forma NAND pura o NOR pura. Por ejemplo, consideremos la función OR EXCLUSIVO de la figura 3-8(a). El circuito estándar de suma de productos se muestra en la

Nombre	Función AND	Función OR
Ley de identidad	$1A = A$	$0 + A = A$
Ley nula	$0A = 0$	$1 + A = 1$
Ley idempotencia	$AA = A$	$A + A = A$
Ley inversa	$A\bar{A} = 0$	$A + \bar{A} = 1$
Ley conmutativa	$AB = BA$	$A + B = B + A$
Ley asociativa	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Ley distributiva	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Ley de absorción	$A(A + B) = A$	$A + AB = A$
Ley de DeMorgan	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Figura 3-6. Algunas identidades del álgebra booleana.

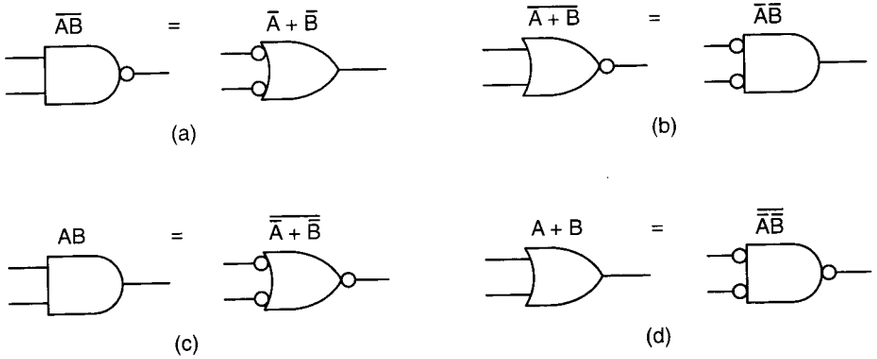


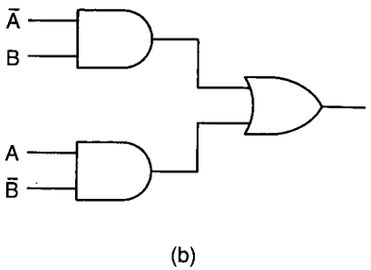
Figura 3-7. Símbolos alternativos para algunas compuertas: (a) NAND. (b) NOR. (c) AND. (d) OR.

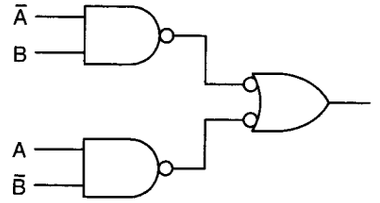
figura 3-8(b). Para convertirlo a una forma NAND, las líneas que conectan la salida de las compuertas AND a la entrada de la compuerta OR se deben redibujar con dos burbujas de inversión, como se muestra en la figura 3-8(c). Por último, utilizando la figura 3-7(a), llegamos a la figura 3-8(d). Las variables $\bar{A}$ y $\bar{B}$ pueden generarse a partir de A y B utilizando compuertas NAND o NOR con sus entradas unidas. Observe que las burbujas de inversión se mueven a lo largo de una línea a voluntad; por ejemplo, de las salidas de las compuertas de entrada de la figura 3-8(d) a las entradas de la compuerta de salida.

Como nota final acerca de la equivalencia de circuitos, demostraremos el sorprendente resultado de que la misma compuerta física puede calcular diferentes funciones, dependiendo de las convenciones empleadas. En la figura 3-9(a) mostramos la salida de cierta compuerta, F, para diferentes combinaciones de entradas. Tanto las entradas como las salidas se

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

(a)





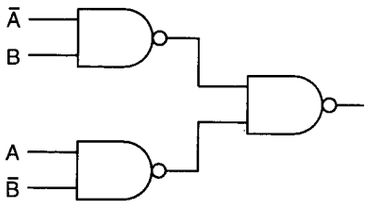


Figura 3-8. (a) Tabla de verdad de la función XOR. (b)-(d) Tres circuitos para calcularla.

indican en volts. Si adoptamos la convención de que 0 volts es un 0 lógico y 3.3 volts o 5 volts es un 1 lógico, llamada **lógica positiva**, obtenemos la tabla de verdad de la figura 3-9(b), la función AND. Pero si adoptamos **lógica negativa**, en la que 0 volts es un 1 lógico y 3.3 o 5 volts es un 0 lógico, obtenemos la tabla de verdad de la figura 3-9(c), la función OR.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Figura 3-9. (a) Características eléctricas de un dispositivo. (b) Lógica positiva. (c) Lógica negativa.

Así pues, la convención que se escoja para transformar voltajes en valores lógicos es crucial. Excepto si se especifica lo contrario, de aquí en adelante usaremos lógica positiva, de modo que los términos 1 lógico, verdadero y alto son sinónimos, lo mismo que los términos 0 lógico, falso y bajo.

3.2 CIRCUITOS LÓGICOS DIGITALES BÁSICOS

En las secciones anteriores vimos cómo implementar tablas de verdad y otros circuitos sencillos utilizando compuertas individuales. En la práctica, ya son pocos los circuitos que se construyen compuerta por compuerta, aunque esto solía ser común. Hoy en día, los bloques de construcción usuales son módulos que contienen varias compuertas. En las secciones que siguen examinaremos más de cerca esos bloques de construcción para ver cómo se usan y cómo pueden construirse a partir de compuertas individuales.

3.2.1 Circuitos integrados

Las compuertas no se fabrican ni se venden individualmente, sino en unidades llamadas **circuitos integrados**, también conocidos como **IC** (*Integrated Circuits*) o **chips**. Un IC es un trozo cuadrado de silicio de unos 5 mm × 5 mm en el que se han depositado algunas compuertas. Los IC pequeños suelen montarse en paquetes de plástico o cerámica rectangulares con una anchura de 5 a 15 mm y una longitud de 20 a 50 mm. En los lados largos hay dos filas paralelas de terminales de unos 5 mm de largo que se pueden insertar en zócalos o soldarse a tarjetas de circuitos impresos. Cada terminal está conectada a la entrada o salida de alguna compuerta del chip, a la alimentación eléctrica, o a tierra. Los paquetes con dos filas de terminales en el exterior e IC en el interior se conocen técnicamente como **paquetes duales en línea** o **DIP** (*Dual Inline Packages*), pero todo mundo los llama chips, lo que hace borrosa la distinción entre el trozo de silicio y su paquete. Los paquetes más comunes tienen 14, 16, 18, 20, 22, 24, 28, 40, 64 o 68 terminales. En el caso de chips grandes es común usar paquetes cuadrados con terminales en los cuatro lados o en la base.

Los chips pueden dividirse a grandes rasgos en clases basadas en el número de compuertas que contienen, como se indica en seguida. Obviamente, este esquema de clasificación es en extremo burdo, pero en ocasiones resulta útil.

Circuito SSI (integrado a pequeña escala, *Small Scale Integrated*): 1 a 10 compuertas.
Circuito MSI (integrado a mediana escala, *Medium Scale Integrated*): 10 a 100 compuertas.
Circuito LSI (integrado a gran escala, *Large Scale Integrated*): 100 a 100,000 compuertas.
Circuito VLSI (integrado a muy grande escala, *Very Large Scale Integrated*): >100,000 compuertas.

Estas clases tienen diferentes propiedades y se usan de distintas formas.

Un chip SSI por lo regular contiene entre dos y seis compuertas independientes, cada una de las cuales se puede usar en forma individual, al estilo de las secciones anteriores. La figura 3-10 ilustra esquemáticamente un chip SSI común que contiene cuatro compuertas NAND. Cada una de éstas tiene dos entradas y una salida, lo que requiere un total de 12 terminales para las cuatro compuertas. Además, el chip necesita alimentación (V_{CC}) y tierra (GND), que todas las compuertas comparten. En general, el paquete tiene una muesca cerca de la terminal 1 para identificar la orientación. Para evitar confusión en los diagramas de circuitos, por convención no se muestran la alimentación, la tierra ni las compuertas que no se utilizan.

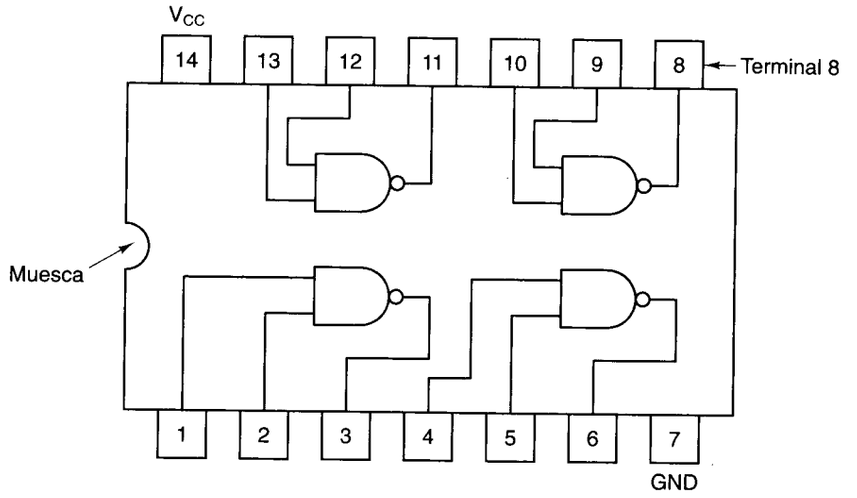


Figura 3-10. Chip SSI que contiene cuatro compuertas.

Es posible obtener muchos otros chips como éste por unos cuantos centavos de dólar cada uno. Cada chip SSI tiene un puñado de compuertas y hasta unas 20 terminales. En los años setenta, las computadoras se construían con grandes cantidades de estas compuertas, pero hoy en día toda una CPU y una cantidad sustancial de memoria (caché) se integran en un solo chip.

Para nuestros fines, todas las compuertas son ideales en el sentido de que la salida aparece tan pronto como se aplica la entrada. En realidad, los chips tienen un **retraso de compuerta** finito que incluye tanto el tiempo de propagación de la señal a través del chip y el tiempo de conmutación. Los retardos típicos son de 1 a 10 ns.

Actualmente es posible colocar casi 10 millones de transistores en un chip. Dado que cualquier circuito se puede construir con compuertas NAND, podríamos pensar que un fabricante podría crear un chip muy general con cinco millones de compuertas NAND. Desafortunadamente, semejante chip necesitaría 15,000,002 terminales. Con el espaciado estándar de 0.1 pulg entre terminales, el chip tendría más de 18 km de largo, lo cual podría tener un efecto negativo sobre las ventas. Obviamente, la única forma de aprovechar la tecnología es diseñar circuitos con una fuerte relación compuertas/terminales. En las secciones que siguen examinaremos circuitos MSI sencillos que combinan varias compuertas internamente para proporcionar una función útil que sólo requiere un número limitado de conexiones externas (terminales metálicas).

3.2.2 Circuitos combinacionales

Muchas aplicaciones de la lógica digital requieren un circuito con varias entradas y varias salidas en el que las salidas estén determinadas de forma única por las entradas vigentes. Un circuito así se denomina **circuito combinacional**. No todos los circuitos tienen esta propiedad. Por ejemplo, un circuito que contiene elementos de memoria bien podría generar salidas que depen-

den de los valores almacenados además de las variables de entrada. Un circuito que implementa una tabla de verdad, como el de la figura 3-3(a), es un ejemplo típico de circuito combinacional. En esta sección examinaremos algunos circuitos combinacionales de uso frecuente.

Multiplexores

En el nivel de lógica digital, un **multiplexor** es un circuito con 2^n entradas de datos, una salida de datos y n entradas de control que seleccionan una de las entradas de datos. La entrada de datos seleccionada se "pasa por compuertas" (se encamina) hacia la salida. La figura 3-11 es un diagrama esquemático de un multiplexor de ocho entradas. Las tres líneas de control, A, B y C, codifican un número de 3 bits que especifica cuál de las ocho líneas de entrada se encamina a la compuerta OR y de ahí a la salida. Sea cual sea el valor que esté en las líneas de control, siete de las compuertas AND siempre producirán 0; la otra podría producir 0 o 1, dependiendo del valor de la línea de entrada seleccionada. Cada compuerta AND se habilita con una combinación distinta de las entradas de control. El circuito del multiplexor se muestra en la figura 3-11. Después de añadir alimentación y tierra, puede meterse en un encapsulado de 14 terminales.

Con el multiplexor podemos implementar la función de mayoría de la figura 3-3(a) como se muestra en la figura 3-12(b). Para cada combinación de A, B y C, se selecciona una de las líneas de entrada de datos. Cada entrada se conecta con V_{CC} (1 lógico) o con tierra (0 lógico). El algoritmo para conectar las entradas es sencillo: la entrada D_i tiene el mismo valor que el renglón i de la tabla de verdad. En la figura 3-3(a) los renglones 0, 1, 2 y 4 son 0, así que las entradas correspondientes se conectan a tierra; los demás renglones son 1, así que se conectan con el 1 lógico. De esta forma, cualquier tabla de verdad de tres variables se puede implementar utilizando el chip de la figura 3-12(a).

Ya vimos cómo puede usarse un chip multiplexor para seleccionar una de varias entradas y cómo puede implementar una tabla de verdad. Otra de sus aplicaciones es como convertidor de datos paralelos a seriales. Si colocamos 8 bits de datos en las líneas de entrada y luego alimentamos las líneas de control sucesivamente con los números binarios 000 a 111, los ocho bits saldrán por la línea de salida en serie. Un uso típico de la conversión paralelo a serial es en un teclado, donde cada digitación define implícitamente un número de 7 u 8 bits que se debe enviar serialmente por una línea telefónica.

Lo contrario de un multiplexor es un **desmultiplexor**, que encamina su única señal de entrada a una de 2^n salidas, dependiendo de los valores de las n líneas de control. Si el valor binario en las líneas de control es k, se selecciona la salida k.

Decodificadores

Como segundo ejemplo, examinaremos un circuito que acepta un número de n bits como entrada y lo utiliza para seleccionar (es decir, poner en 1) una y sólo una de las 2^n líneas de salida. Semejante circuito, que se ilustra para $n = 3$ en la figura 3-13, se llama **decodificador**.

Para ver en qué casos podría ser útil un decodificador, imagine una memoria que consiste en ocho chips, cada uno de los cuales contiene 1 MB. El chip 0 tiene las direcciones de 0 a

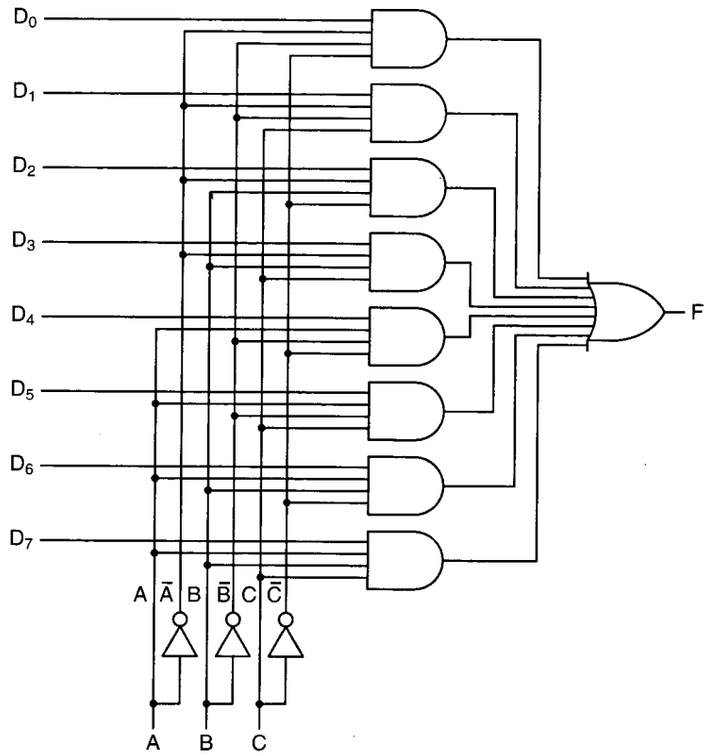


Figura 3-11. Circuito multiplexor de ocho entradas.

1 MB, el chip 1 tiene las direcciones de 1 MB a 2 MB, etc. Cuando se presenta una dirección a la memoria, los 3 bits de orden alto sirven para seleccionar uno de los ocho chips. Utilizando el circuito de la figura 3-13, estos 3 bits son las tres entradas A, B y C. Dependiendo de las entradas, una y sólo una de las ocho líneas de salida, D_0, ..., D_7, es 1; el resto son 0. Cada línea de salida habilita uno de los ocho chips de memoria. Puesto que sólo una línea de salida se pone en 1, sólo se habilita un chip.

El funcionamiento del circuito de la figura 3-13 es sencillo. Cada compuerta AND tiene tres entradas, de las cuales la primera es A o bien $\overline{A}$, la segunda es B o bien $\overline{B}$ y la tercera es C o bien $\overline{C}$. Cada compuerta se habilita con una combinación de entradas distinta: D_0 con $\overline{A}\,\overline{B}\,\overline{C}$, D_1 con $\overline{A}\,\overline{B}\,C$, etcétera.

Comparadores

Otro circuito útil es el **comparador**, que compara dos palabras de entrada. El sencillo comparador de la figura 3-14 acepta dos entradas, A y B, cada una de 4 bits, y produce 1 si son iguales y 0 en caso contrario. El circuito se basa en la compuerta XOR (OR EXCLUSIVA),

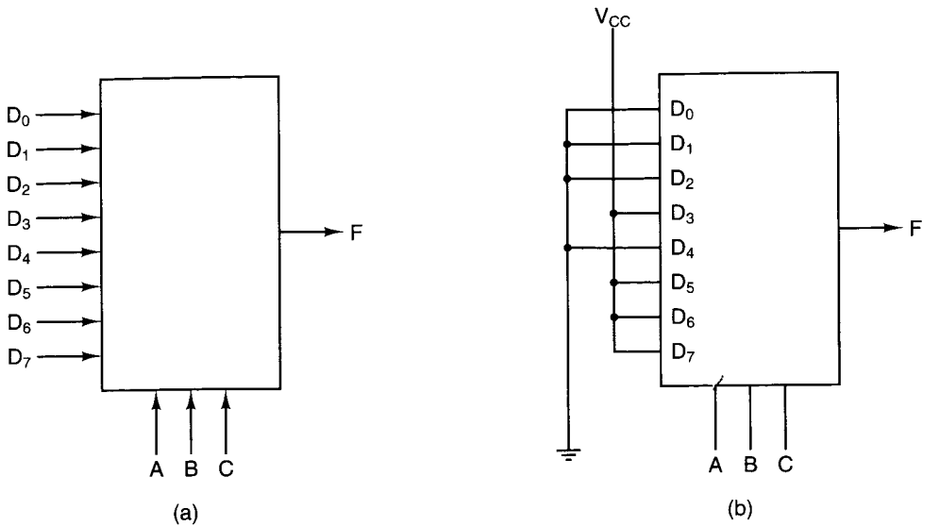


Figura 3-12. (a) Multiplexor MSI. (b) El mismo multiplexor alambrado para calcular la función de mayoría.

que produce un 0 si sus entradas son iguales, y un 1 si no lo son. Si las dos palabras de entrada son iguales, las cuatro compuertas XOR deberán producir 0. Estas cuatro señales se conectan después a una compuerta OR; si el resultado es 0, las palabras de entrada son iguales; si es 1, no lo son. En nuestro ejemplo usamos una compuerta NOR como etapa final para invertir el sentido de la prueba: 1 significa igualdad; 0 significa desigualdad.

Arreglos lógicos programables

Ya vimos que es posible construir funciones arbitrarias (tablas de verdad) calculando términos producto con compuertas AND y haciendo pasar los resultados por una compuerta OR. Un chip muy general para formar sumas de productos es el **arreglo lógico programable** o **PLA** (*Programmable Logic Array*), del cual se muestra un ejemplo pequeño en la figura 3-15. Este chip tiene líneas de entrada para 12 variables. El complemento de cada entrada se genera internamente, para dar un total de 24 señales de entrada. El corazón del circuito es un arreglo de 50 compuertas AND, cada una de las cuales puede tener en potencia cualquier subconjunto de las 24 señales de entrada como su entrada. La determinación de qué señal de entrada va a qué compuerta AND se efectúa con un arreglo de 24×50 bits proporcionado por el usuario. Cada línea de entrada a las 50 compuertas AND contiene un fusible. Cuando el circuito sale de la fábrica, los 1200 fusibles están intactos. Para programar el arreglo, el usuario quema fusibles selectos aplicando un voltaje alto al chip.

La parte de salida del circuito consiste en seis compuertas OR, cada una de las cuales tiene hasta 50 entradas que corresponden a las 50 salidas de las compuertas AND. En este caso también, un arreglo (de 50×6) proporcionado por el usuario indica cuáles de las posibles

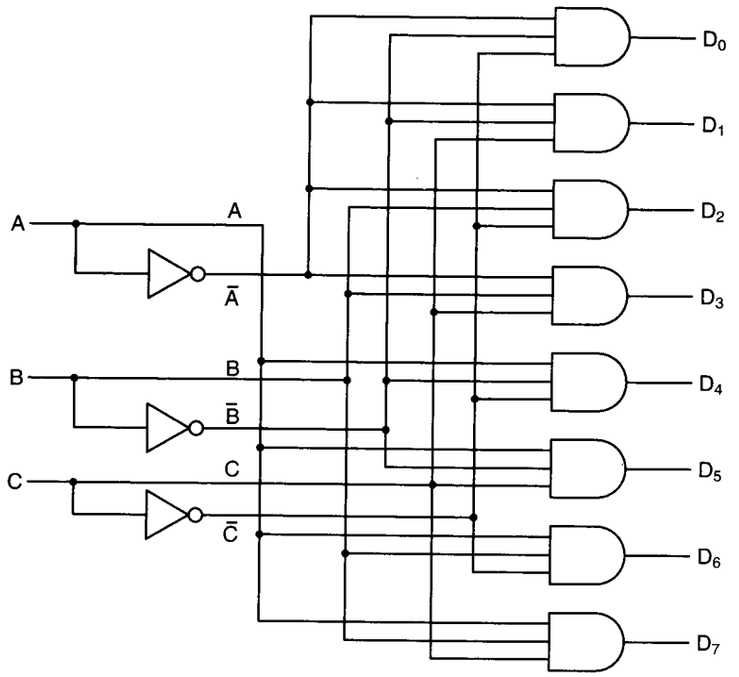


Figura 3-13. Un circuito decodificador de 3 a 8.

conexiones realmente existen. El chip tiene 12 terminales de entrada, 6 terminales de salida, alimentación y tierra, para un total de 20.

Como ejemplo del uso de un PLA, volvamos a considerar el circuito de la figura 3-3(b). Este circuito tiene tres entradas, cuatro compuertas AND, una compuerta OR y tres inversores. Una vez hechas las conexiones internas apropiadas, nuestro PLA puede calcular la misma función utilizando tres de sus 12 entradas, cuatro de sus 50 compuertas AND y una de sus seis compuertas OR. (Las cuatro compuertas AND deberán calcular $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ y ABC, respectivamente; la compuerta OR acepta estos cuatro términos de producto como entradas.) De hecho, el mismo PLA podría configurarse a modo de calcular simultáneamente un total de cuatro funciones de complejidad similar. En el caso de estas funciones sencillas el número de variables de entrada es el factor limitante; si la función es más compleja, el factor limitante podría ser el número de compuertas AND y OR.

Aunque los PLA programables en campo que acabamos de describir se siguen usando, para muchas aplicaciones son preferibles los PLA hechos a la medida. El cliente (al mayoreo) diseña estos PLA y el fabricante los produce según las especificaciones del cliente. Tales PLA cuestan menos que los programables en campo.

Ahora podemos comparar las tres diferentes formas de implementar la tabla de verdad de la figura 3-3(a) que hemos visto. Si utilizamos componentes SSI, necesitamos cuatro chips. Como alternativa, bastaría con un chip multiplexor MSI, como se muestra en la figura

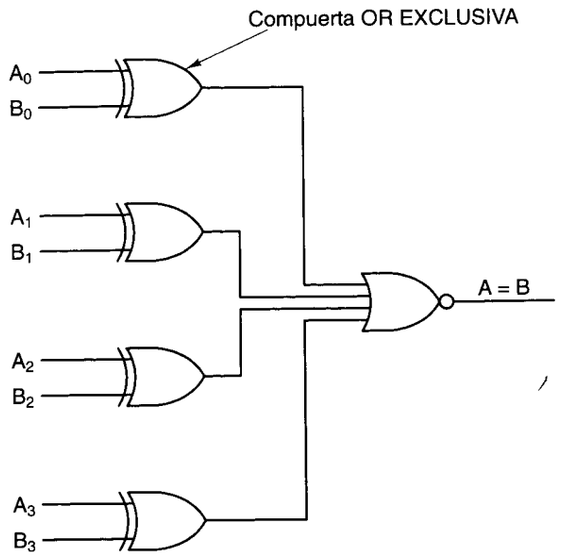


Figura 3-14. Comparador simple de 4 bits.

3-12(b). Por último, podríamos usar una cuarta parte de un chip PLA. Obviamente, si se necesitan muchas funciones, el PLA es más eficiente que los otros dos métodos. En el caso de circuitos sencillos, los chips SSI y MSI, más baratos, podrían ser preferibles.

3.2.3 Circuitos aritméticos

Ha llegado el momento de pasar de los circuitos MSI de propósito general vistos anteriormente a los circuitos combinacionales MSI que sirven para efectuar operaciones aritméticas. Comenzaremos con un sencillo desplazador de 8 bits, y luego veremos cómo se construyen los sumadores. Por último, examinaremos las unidades aritmética y lógica, que desempeñan un papel central en cualquier computadora.

Desplazadores

Nuestro primer circuito MSI de aritmética es un desplazador de ocho entradas y ocho salidas (vea la figura 3-16). Se presentan ocho bits de entrada en las líneas $D_0, ..., D_7$. La salida, que no es más que la entrada desplazada un bit, se obtiene en las líneas $S_0, ..., S_7$. La línea de control, C, determina la dirección del desplazamiento, 0 hacia la izquierda y 1 hacia la derecha.

Para ver cómo funciona el circuito, observe los pares de compuertas AND para todos los bits excepto las compuertas del final. Cuando $C = 1$, el miembro derecho de cada par se enciende, y pasa el bit de entrada correspondiente a la salida. Puesto que la compuerta AND derecha está conectada a la salida de la compuerta OR que está a su derecha, se efectúa un

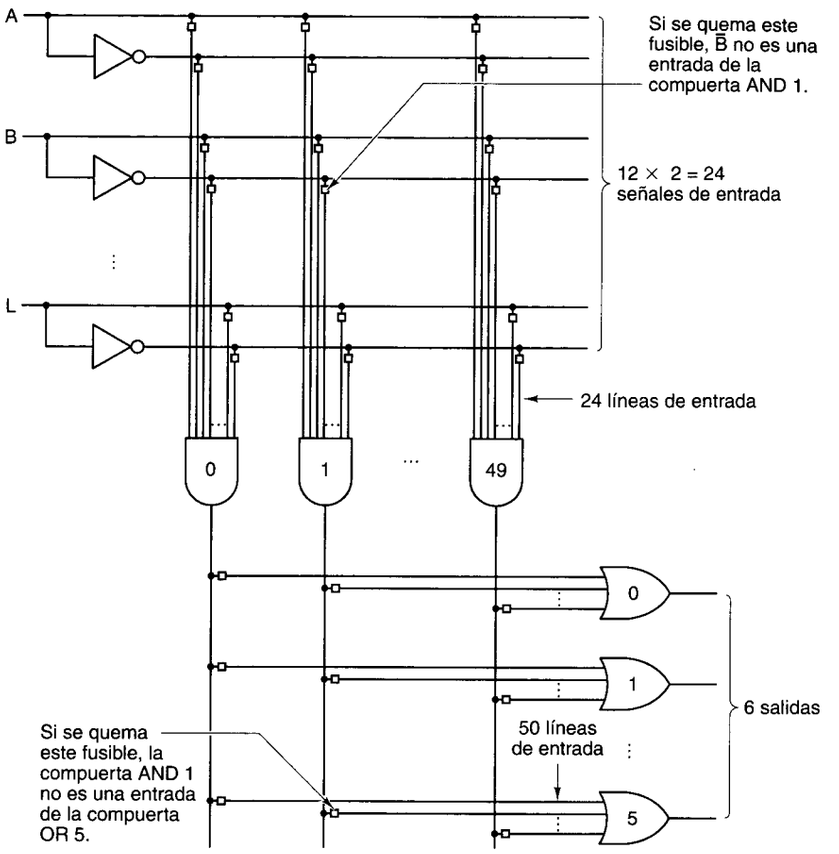


Figura 3-15. Arreglo lógico programable de 12 entradas y 6 salidas. Los cuadritos representan fusibles que pueden quemarse para determinar la función que se calculará. Los fusibles están dispuestos en dos matrices; la de arriba para las compuertas AND y la de abajo para las compuertas OR.

desplazamiento hacia la derecha. Cuando $C = 0$, es el miembro izquierdo del par de compuertas AND el que se enciende, y se efectúa un desplazamiento a la izquierda.

Sumadores

Una computadora que no puede sumar enteros es casi inconcebible. Por tanto, un circuito de hardware para efectuar una suma es un componente indispensable de toda CPU. La tabla de verdad para la suma de enteros de un bit se muestra en la figura 3-17(a). Hay dos salidas: la suma de las entradas, A y B, y el acarreo a la siguiente posición (hacia la izquierda). En la figura 3-17(b) se ilustra un circuito para calcular tanto el bit de suma como el bit de acarreo. Este sencillo circuito se conoce generalmente como **medio sumador**.

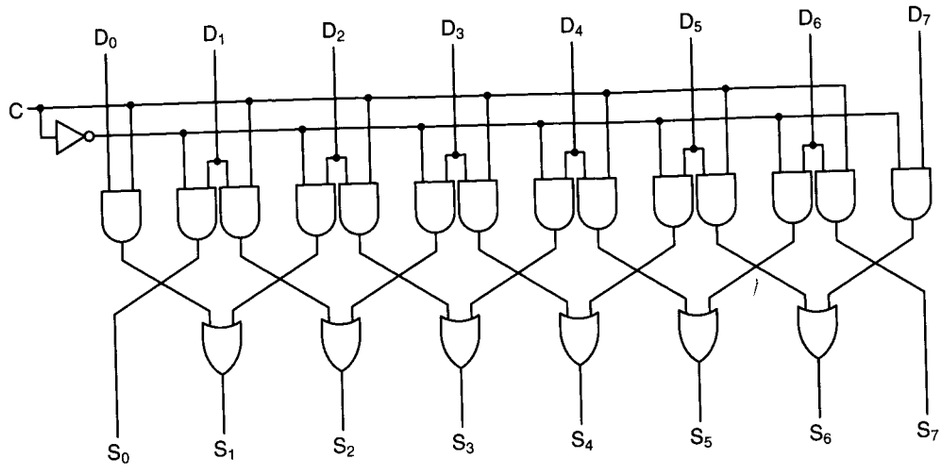


Figura 3-16. Desplazador a la izquierda/derecha de un bit.

A	B	Suma	Acarreo
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

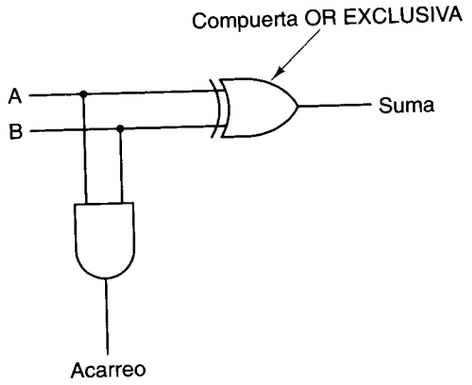
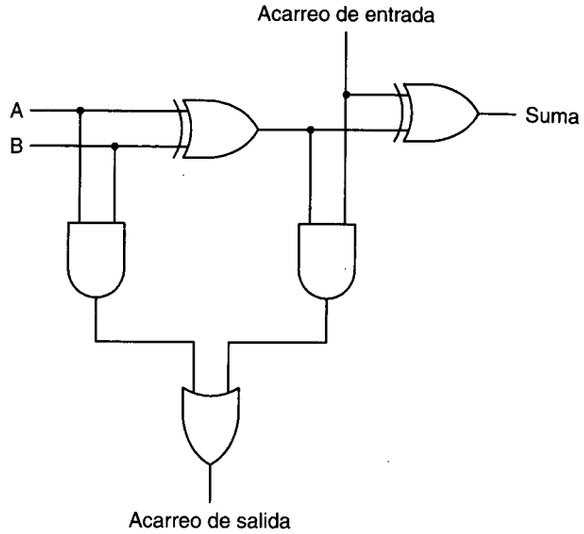


Figura 3-17. (a) Tabla de verdad para la suma de un bit. (b) Circuito de un medio sumador.

Aunque un medio sumador es suficiente para sumar los bits de orden bajo de dos palabras de entrada de varios bits, no sirve para una posición de bit interior de la palabra porque no maneja el acarreo que llega de la posición que está a su derecha. Se necesita el **sumador completo** de la figura 3-18. Si examinamos el circuito nos quedará claro que un sumador completo se construye con dos medio sumadores. La línea de salida *Suma* es 1 si el número de unos en *A*, *B* y el *Acarreo de entrada* es impar. *Acarreo de salida* es 1 si *A* y *B* son ambos 1 (entrada izquierda de la compuerta OR) o uno y sólo uno de ellos es 1 y el bit *Acarreo de entrada* es también 1. Juntos, los dos medio sumadores generan los bits tanto de suma como de acarreo.

Si queremos construir un sumador para, digamos, dos palabras de 16 bits, basta con repetir el circuito de la figura 3-18(b) 16 veces. El acarreo de salida de un bit se usa como

A	B	Acarreo ent.	Suma	Acarreo sal.
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(a) (b)

Figura 3-18. (a) Tabla de verdad para un sumador completo. (b) Circuito de un sumador completo.

acarreo de entrada de su vecino izquierdo. El acarreo de entrada del bit de la extrema derecha se conecta a 0. Este tipo de sumador se llama **sumador con propagación de acarreo** porque, en el peor de los casos, que es sumar 1 a 111...111 (binario), la suma no puede completarse hasta que el acarreo se ha propagado como una onda desde el bit de la extrema derecha hasta el de la extrema izquierda. También existen, y generalmente se prefieren, sumadores que no tienen este retraso y por tanto son más rápidos.

Como ejemplo sencillo de sumador más rápido, considere la separación de un sumador de 32 bits en una mitad inferior de 16 bits y una mitad superior de 16 bits. Cuando se inicia la suma, el sumador superior no puede ponerse a trabajar todavía porque no va a conocer su acarreo de entrada sino hasta dentro de 16 tiempos de suma.

Sin embargo, consideremos esta modificación. En lugar de tener una sola mitad superior demos al sumador dos mitades superiores en paralelo duplicando el hardware de la mitad superior. Así el circuito consiste ahora en tres sumadores de 16 bits: una mitad superior y dos mitades superiores, *U0* y *U1* que se ejecutan en paralelo. Se alimenta un 0 a *U0* como acarreo; y se alimenta un 1 a *U1* como acarreo. Ahora ambos pueden comenzar a trabajar al mismo tiempo que la mitad inferior, pero sólo uno dará el resultado correcto. Después de 16 tiempos de suma, se sabrá si el acarreo de entrada de la segunda mitad es 0 o 1, y podrá seleccionarse la mitad superior correcta de entre las dos respuestas disponibles. Este truco reduce el tiempo de suma en un factor de dos. Un sumador de este tipo se llama **sumador por selección de acarreo**. El truco puede repetirse para construir cada sumador de 16 bits con sumadores de 8 bits repetidos, y así sucesivamente.

Unidades aritmética lógica

Casi todas las computadoras contienen un solo circuito para obtener el AND, el OR y la suma de dos palabras de máquina. Por lo regular, un circuito de este tipo para palabras de n bits se construye con n circuitos idénticos para las posiciones de bit individuales. La figura 3-19 es un ejemplo sencillo de un circuito de este tipo, llamado **unidad aritmética lógica** o **ALU** (*Arithmetic Logic Unit*). Una unidad aritmética y lógica puede calcular cualquiera de cuatro funciones: A AND B, A OR B, $\overline{B}$, o $A + B$, dependiendo de si las líneas de entrada de selección de función F_0 y F_1 contienen 00, 01, 10 o 11 (binario). Cabe señalar que aquí $A + B$ significa la suma aritmética de A y B, no el AND booleano.

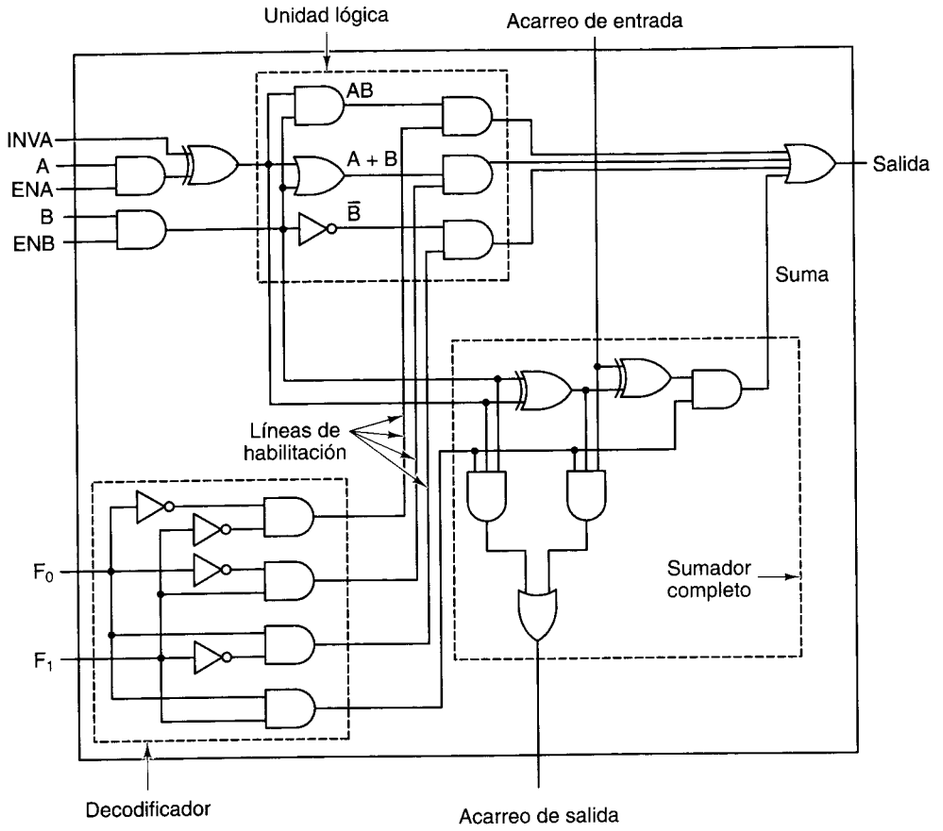


Figura 3-19. Una ALU de un bit.

La esquina inferior izquierda de nuestra ALU contiene un decodificador de 2 bits que genera señales de habilitación para las cuatro operaciones, con base en las señales de control F_0 y F_1. Dependiendo de los valores de F_0 y F_1, una y sólo una de las cuatro líneas de habilitación se selecciona. Un valor de 1 en una de estas líneas permite que la salida de la función seleccionada pase hasta la compuerta OR final y de ahí a la salida.

La esquina superior izquierda contiene la lógica necesaria para calcular A AND B, A OR B y $\overline{B}$, pero sólo uno de estos resultados pasa a la compuerta OR final, dependiendo de las líneas de habilitación que salen del decodificador. Dado que una y sólo una de las líneas del decodificador es 1, una y sólo una de las cuatro compuertas AND que alimentan a la compuerta OR estará habilitada; las otras tres producirán 0, sean cuales sean los valores de A y B.

Además de poder usar A y B como entradas de operaciones aritméticas o lógicas, es posible hacer que cualquiera de las dos sea 0 negando ENA o ENB, respectivamente. También es posible obtener $\overline{A}$ poniendo INVA. Veremos algunos usos para INVA, ENA y ENB en el capítulo 4. En condiciones normales, tanto ENA como ENB son 1 para habilitar ambas entradas, e INVA es 0. En este caso, A y B simplemente se alimentan a la unidad lógica sin modificación.

La esquina inferior derecha de la ALU contiene un sumador completo para calcular la suma de A y B, e incluye manejo de acarreos porque es probable que se conecten juntos varios de estos circuitos para efectuar operaciones con palabras enteras. Es posible conseguir circuitos como el de la figura 3-19, llamado **porción de bit (*bit slice*)**. Estas porciones permiten al diseñador de una computadora construir una ALU de la capacidad deseada. La figura 3-20 muestra una ALU de 8 bits formada por ocho ALU de porción de bit. La señal INC sólo es útil en operaciones de suma; si está presente, incrementa (es decir, suma 1 a) el resultado, lo que permite calcular sumas como $A + 1$ y $A + B + 1$.

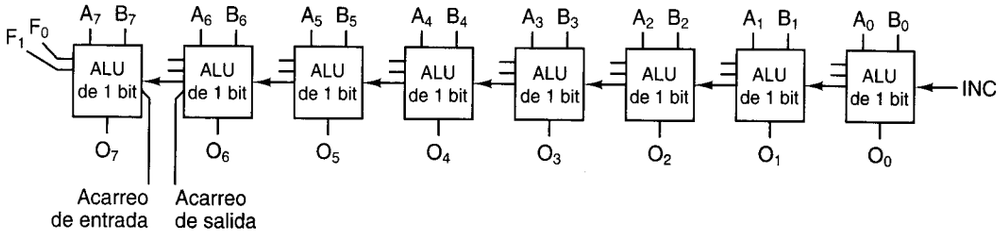


Figura 3-20. Ocho "tajadas" de ALU de un bit conectadas para formar una ALU de ocho bits. Por sencillez, no se muestran las señales de habilitación y de inversión.

3.2.4 Relojes

En muchos circuitos digitales el orden en que ocurren los sucesos es crucial. En ocasiones un suceso debe preceder a otro, a veces dos sucesos deben ocurrir simultáneamente. Para que los diseñadores puedan establecer las relaciones de temporización requeridas, muchos circuitos digitales emplean relojes que hacen posible la sincronización. Un **reloj** en este contexto es un circuito que emite una serie de pulsaciones con una anchura de pulsación precisa y un intervalo preciso entre pulsaciones consecutivas. El periodo entre los flancos correspondientes de dos pulsaciones consecutivas se denomina **tiempo de ciclo del reloj**. Las frecuencias de pulsación

suelen ser de 1 a 500 MHz, lo que corresponde a ciclos de reloj de 1000 ns a 2 ns. Para lograr gran exactitud, la frecuencia del reloj normalmente se controla con un oscilador de cristal.

En una computadora pueden ocurrir muchos sucesos durante un solo ciclo de reloj. Si dichos sucesos deben ocurrir en un orden específico, el ciclo de reloj debe dividirse en subciclos. Una forma común de obtener una definición más fina que la del reloj básico consiste en sacar una derivación de la línea primaria del reloj e insertar un circuito que tenga un retraso conocido. Esto genera una señal de reloj secundaria desfasada respecto a la primaria, como se muestra en la figura 3-21(a). El diagrama de temporización de la figura 3-21(b) proporciona cuatro referencias de tiempo para sucesos discretos:

1. Flanco ascendente de C1.
2. Flanco descendente de C1.
3. Flanco ascendente de C2.
4. Flanco descendente de C2.

Si se vinculan diferentes sucesos con los distintos flancos, se podrá lograr la sucesión requerida. Si se necesitan más de cuatro referencias de tiempo dentro de un ciclo de reloj, pueden derivarse más líneas secundarias de la primaria, con diferentes retardos.

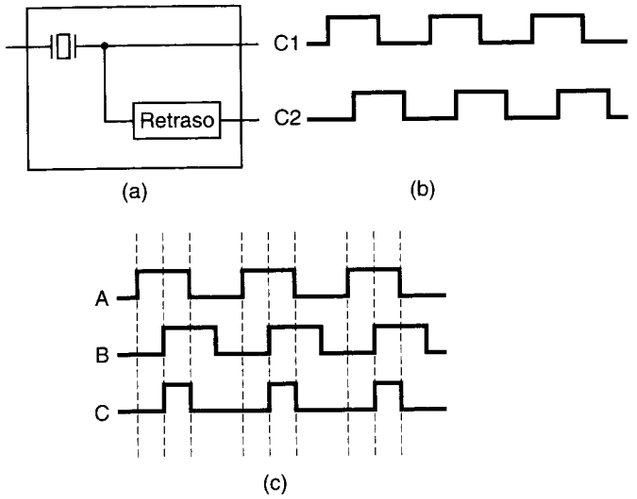


Figura 3-21. (a) Un reloj. (b) Diagrama de temporización del reloj. (c) Generación de un reloj asimétrico.

En algunos circuitos lo que interesa son los periodos más que los instantes discretos en el tiempo. Por ejemplo, podría permitirse que ocurra cierto suceso en cualquier momento en que C1 esté alto, no precisamente en el flanco ascendente. Otro suceso podría suceder sólo cuando C2 está alto. Si se requieren más de dos intervalos, pueden proporcionarse más líneas

de reloj, o puede hacerse que los estados altos de los dos relojes se traslapen de manera parcial en el tiempo. En este último caso pueden distinguirse cuatro intervalos distintos: $\overline{C1}$ AND $\overline{C2}$, $\overline{C1}$ AND C2, C1 AND $\overline{C2}$ y C1 AND C2.

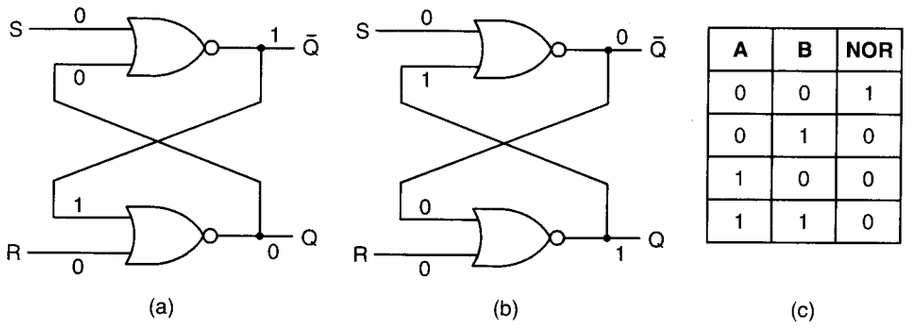
Por cierto, los relojes son simétricos en el sentido de que pasan tanto tiempo en el estado alto como en el estado bajo, como se muestra en la figura 3-21(b). Si se desea generar un tren de pulsaciones asimétrico, el reloj básico se desplaza utilizando un circuito de retraso y luego se hace AND con la señal original, como se muestra en la figura 3-21(c) para producir C.

3.3 MEMORIA

Un componente indispensable de toda computadora es su memoria. Sin memoria no podría haber computadoras como las que conocemos. La memoria sirve para almacenar las instrucciones que se van a ejecutar, y también los datos. En las secciones que siguen examinaremos los componentes básicos de un sistema de memoria comenzando desde el nivel de compuerta, para ver cómo funcionan y cómo se combinan para producir memorias grandes.

3.3.1 Latches

Para crear una memoria de un bit necesitamos un circuito que de alguna manera "recuerde" los valores de entrada anteriores. Podemos construir un sistema así a partir de dos compuertas NOR, como se ilustra en la figura 3-22(a). También hay circuitos análogos que se basan en compuertas NAND, pero no los mencionaremos más porque son idénticos en lo conceptual a las versiones con NOR.



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Figura 3-22. (a) Latch NOR en el estado 0. (b) Latch NOR en el estado 1. (c) Tabla de verdad para el NOR.

El circuito de la figura 3-22(a) se llama **latch SR**. Este circuito tiene dos entradas, *S* (*Set*) para establecer el latch (es decir, ponerlo en 1), y *R* (*Reset*) para restablecerlo (es decir, borrarlo o ponerlo en 0). También hay dos salidas, *Q* y $\overline{Q}$, que son complementarias, como veremos en breve. A diferencia de los circuitos combinacionales, las salidas de un latch no están determinadas de forma única por las entradas vigentes.

Para saber la razón, supongamos que tanto S como R son 0, que es lo más común. Por ahora, asumamos también que $Q = 0$. Puesto que Q se realimenta a la compuerta NOR superior, las dos entradas de ésta son 0, así que su salida, $\overline{Q}$ es 1. El 1 se realimenta a la compuerta inferior, cuyas entradas son entonces 1 y 0, lo que da $Q = 0$. Este estado es al menos consistente y se muestra en la figura 3-22(a).

Imaginemos ahora que Q no es 0, sino 1, y que R y S siguen siendo 0. Las entradas de la compuerta superior son 0 y 1, así que produce una salida $\overline{Q}$ de 0, la cual se realimenta a la compuerta inferior. Este estado, que se muestra en la figura 3-22(b), también es consistente. Un estado en el que ambas salidas son iguales a 0 es inconsistente, porque obliga a ambas compuertas a tener dos ceros como entradas, y ello produciría una salida de 1, no de 0. Asimismo, es imposible que ambas salidas sean iguales a 1, porque eso obligaría a que las entradas fueran 0 y 1, lo que produce 0, no 1. Nuestra conclusión es sencilla: con $R = S = 0$, el latch tiene dos estados estables, a los que llamaremos 0 y 1, que dependen de Q.

Examinemos ahora el efecto de las entradas sobre el estado del latch. Supongamos que S se vuelve 1 mientras $Q = 0$. Las entradas de la compuerta superior son entonces 1 y 0, y esto hace que la salida $\overline{Q}$ sea 0. Este cambio hace que las dos entradas de la compuerta inferior sean 0, y por tanto que su salida sea 1. Así, establecer S (o sea, ponerlo en 1) conmuta el estado de 0 a 1. Poner R en 1 cuando el latch está en el estado 0 no tiene ningún efecto porque la salida de la compuerta NOR inferior es 0 tanto para las entradas 10 como para 11.

Por un razonamiento similar, es fácil ver que poner S en 1 mientras $Q = 1$ no surte ningún efecto, pero poner R en 1 hace que el latch pase al estado $Q = 0$. En síntesis, cuando S se pone en 1 momentáneamente, el latch termina en el estado $Q = 1$, sin importar cuál haya sido su estado anterior. De igual manera, poner R en 1 hace que el latch pase al estado $Q = 0$. El circuito "recuerda" si fue S o R la última que estuvo en 1. Podemos aprovechar esta propiedad para construir memorias de computadora.

Latches SR con reloj

En muchos casos es conveniente impedir que el latch cambie de estado como no sea en ciertos momentos específicos. Para ello, modificamos un poco el circuito básico, como se muestra en la figura 3-23, para obtener un **latch SR con reloj**.

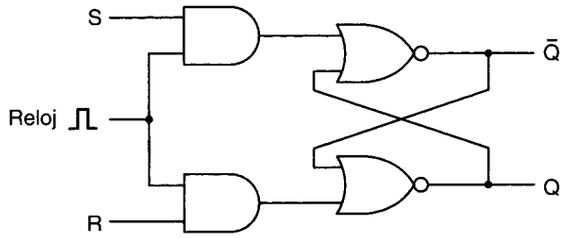


Figura 3-23. Latch SR con reloj.

Este circuito tiene una entrada adicional, el reloj, que normalmente es 0. Con el reloj en 0, ambas compuertas AND producen 0, sean cuales sean los valores de S y R, y el latch no cambia

de estado. Cuando el reloj está en 1, el efecto de las compuertas AND desaparece y el latch es sensible a S y R. A pesar de su nombre, la señal de reloj no tiene que provenir de un reloj. También se usan ampliamente los términos **enable** (habilitar) y **strobe** (señal estroboscópica) para indicar que la entrada de reloj es 1; es decir, que el circuito es sensible al estado de S y R.

Hasta ahora hemos evitado considerar el problema que sucede cuando tanto S como R son 1, y hemos tenido una buena razón para hacerlo: el circuito se vuelve no determinista cuando por fin R y S regresan a 0. El único estado consistente para $S = R = 1$ es $Q = \overline{Q} = 0$, pero tan pronto como ambas entradas regresen a 0 el latch deberá saltar a uno de sus dos estados estables. Si una de las dos entradas regresa a 0 antes que la otra, la que se quedó en 1 más tiempo gana, porque cuando sólo una de las entradas es 1, determina el estado del latch. Si ambas entradas regresan a 0 al mismo tiempo (lo cual es muy poco probable), el latch salta a uno de sus estados estables al azar.

Latches D con reloj

Una buena forma de resolver la ambigüedad del latch SR (que surge cuando $S = R = 1$) es evitar que ocurra. La figura 3-24 muestra un circuito de latch que sólo tiene una entrada, D. Puesto que la entrada a la compuerta AND inferior siempre es el complemento de la entrada de la compuerta AND superior, nunca surge el problema de que ambas entradas sean 1. Cuando $D = 1$ y el reloj es 1, el latch pasa al estado $Q = 1$. Cuando $D = 0$ y el reloj es 1, el latch pasa al estado $Q = 0$. En otras palabras, cuando el reloj es 1, se muestrea el valor actual de D y se almacena en el latch. Este circuito, llamado **latch D con reloj**, es una verdadera memoria de un bit. El valor almacenado siempre está disponible en Q. Para cargar el valor actual de D en la memoria, se aplica una pulsación positiva a la línea del reloj.

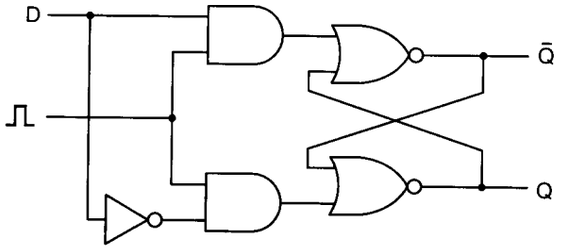


Figura 3-24. Latch D con reloj.

Este circuito requiere 11 transistores. Circuitos más avanzados (pero menos obvios) pueden almacenar 1 bit con menos transistores, hasta un mínimo de seis. En la práctica ésos son los diseños normalmente usados.

3.3.2 Flip-flops

En muchos circuitos es necesario muestrear el valor que hay en una línea dada en un instante dado, y almacenarlo. En esta variante, llamada **flip-flop**, la transición de estado no ocurre cuando el reloj es 1, sino durante la transición del reloj de 0 a 1 (flanco ascendente) o de 1 a

0 (flanco descendente). Así, la longitud del pulso de reloj no importa, en tanto las transiciones sean rápidas.

Conviene hacer hincapié en la diferencia entre un flip-flop y un latch. Un flip-flop **se dispara por flanco**, mientras que un latch **se dispara por nivel**. Cabe advertir, empero, que en la literatura hay veces en que se confunden estos términos. Muchos autores usan "flip-flop" cuando se refieren a un latch, y viceversa.

Hay varias formas de diseñar un flip-flop. Por ejemplo, si hubiera alguna forma de generar una pulsación muy corta en el flanco ascendente de la señal del reloj, esa pulsación podría alimentarse a un latch D. De hecho, sí hay una forma de hacerlo, y el circuito correspondiente se muestra en la figura 3-25(a).

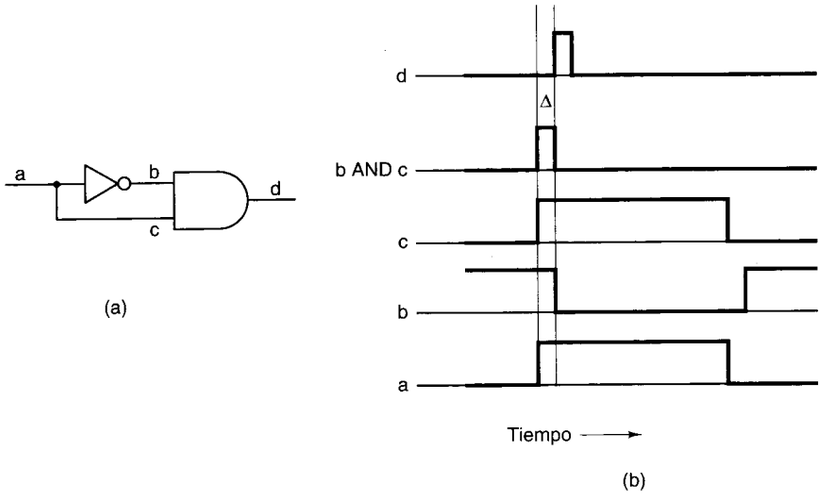


Figura 3-25. (a) Generador de pulsaciones. (b) Temporización en cuatro puntos del circuito.

A primera vista, podría parecer que la salida de la compuerta AND siempre sería cero, ya que el AND de cualquier señal con su inverso es cero, pero la situación es un poco más compleja. El inversor tiene un retraso de propagación pequeño, pero distinto de cero, y ese retraso es lo que hace que el circuito funcione. Suponga que medimos el voltaje en los cuatro puntos de medición *a*, *b*, *c* y *d*. La señal de entrada, medida en *a*, es un pulso largo de reloj, como se muestra en la figura 3-25(b) en la parte inferior. Arriba se muestra la señal en *b*. Observe que está invertida y también tiene un pequeño retraso, por lo regular de unos cuantos nanosegundos dependiendo del tipo de inversor que se use.

La señal en *c* también tiene un retraso, pero éste es sólo por el tiempo de propagación de la señal (a la velocidad de la luz). Si la distancia física entre *a* y *c* es, por ejemplo, de 20 micras, el retraso de propagación es de 0.0001 ns, que sin duda es insignificante en comparación con el tiempo que la señal tarda en propagarse a través del inversor. Por ello, en la práctica podemos considerar que la señal en *c* es idéntica a la señal en *a*.

Cuando se obtiene el AND de las entradas de la compuerta AND, *b* y *c*, el resultado es un pulso corto, como se muestra en la figura 3-25(b), donde la anchura de la pulsación, Δ, es igual al retardo de compuerta del inversor, que por lo regular es de 5 ns o menos. La salida de la compuerta AND no es más que esta pulsación desplazada por el retraso de la compuerta AND, como se aprecia en la parte superior de la figura 3-25(b). Este desplazamiento en el tiempo sólo implica que el latch D se activará con un retraso fijo después del flanco ascendente del reloj, pero no tiene efecto sobre la anchura de la pulsación. En una memoria con un tiempo de ciclo de 50 ns, una pulsación de 5 ns que le indica cuándo debe muestrear la línea *D* podría ser lo bastante corta, en cuyo caso el circuito completo puede ser el de la figura 3-26. Vale la pena señalar que este diseño de flip-flop es bonito porque es fácil de entender, pero en la práctica normalmente se usan flip-flops más complejos.

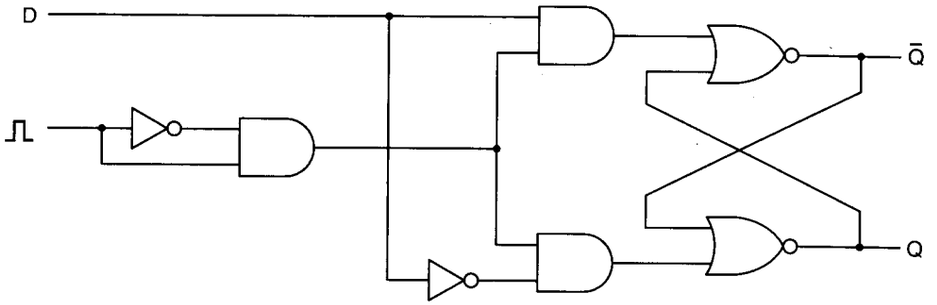


Figura 3-26. Flip-flop D.

Los símbolos estándar para los latches y flip-flops se muestran en la figura 3-27. La figura 3-27(a) es un latch cuyo estado se carga cuando el reloj, *CK*, es 1, en contraste con la figura 3-27(b), que es un latch cuyo reloj normalmente es 1 pero baja a 0 momentáneamente para cargar el estado de *D*. Las figuras 3-27(c) y (d) son flip-flops, no latches, lo que se indica con el símbolo angular en la entrada del reloj. El de la figura 3-27(c) cambia de estado con el flanco ascendente del pulso de reloj (transición 0 a 1), mientras que el de la figura 3-27(d) cambia de estado con el flanco descendente (transición 1 a 0). Muchos latches y flip-flops, pero no todos, también tienen $\overline{Q}$ como salida, y algunos tienen dos entradas adicionales, *Establecer* o *Preestablecer* (*Set* o *Preset*, para forzar el estado a $Q = 1$) y *Restablecer* o *Borrar* (*Reset* o *Clear*, para forzar el estado a $Q = 0$).

3.3.3 Registros

Los flip-flops se pueden conseguir en diversas configuraciones. Una de las más sencillas, que contiene dos flip-flops D independientes con señales de borrar y preestablecer, se ilustra en la figura 3-28(a). Aunque se empaquetan en el mismo chip de 14 terminales, los dos flip-flops son independientes. Una configuración muy distinta es el flip-flop octal de la figura 3-28(b). Aquí los ocho (de ahí el término "octal") flip-flops D no sólo carecen de las líneas $\overline{Q}$ y *preset*, sino que todas las líneas de reloj se juntan y se alimentan de la terminal 11. Los flip-flops

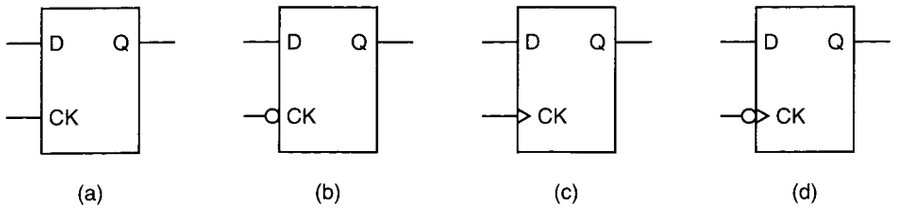


Figura 3-27. Latches y flip-flops D.

mismos son del tipo del de la figura 3-27(d), pero las burbujas de inversión en los flip-flops se cancelan con el inversor conectado a la terminal 11, de modo que los flip-flops se cargan durante la transición ascendente. Las ocho señales de borrar también se juntan, de modo que cuando la terminal 1 baja a 0 todos los flip-flops se fuerzan a su estado 0. Por si se está preguntando por qué la terminal 11 se invierte en la entrada y luego se invierte otra vez en cada señal CK, una señal de entrada podría no tener la suficiente corriente para alimentar a los ocho flip-flops; el inversor de entrada se está usando en realidad como amplificador.

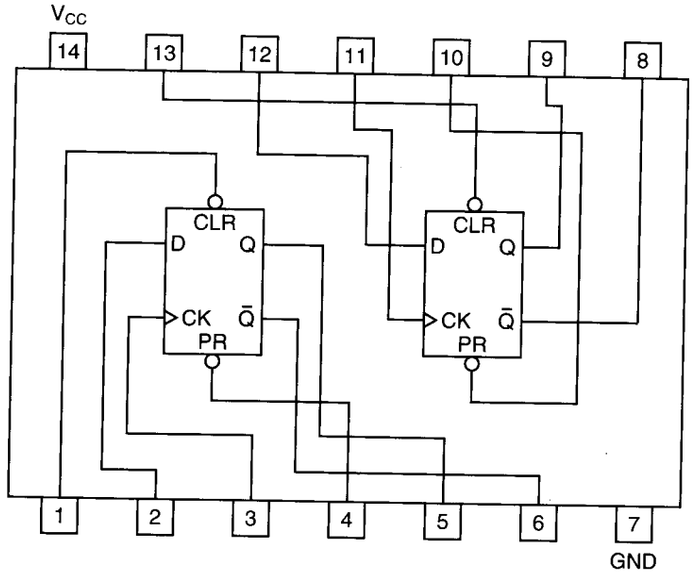
Si bien una razón para juntar las líneas de reloj y de borrar en la figura 3-28(b) es ahorrar terminales, en esta configuración el chip se usa de forma diferente de como se usarían ocho flip-flops independientes: se utiliza como un solo registro de ocho bits. También podrían usarse dos de estos chips en paralelo para formar un registro de 16 bits conectando sus respectivas terminales 1 y 11. Examinaremos los registros y sus aplicaciones con mayor detalle en el capítulo 4.

3.3.4 Organización de la memoria

Aunque ya avanzamos de la sencilla memoria de un bit de la figura 3-24 a la memoria de ocho bits de la figura 3-28(b), para construir memorias grandes se requiere una organización diferente, en la cual sea posible direccionar palabras individuales. Una organización de memoria muy utilizada que satisface este criterio se muestra en la figura 3-29. Este ejemplo ilustra una memoria con cuatro palabras de 3 bits. Cada operación lee o escribe una palabra de 3 bits completa. Si bien la capacidad total de memoria de 12 bits apenas rebasa la de nuestro flip-flop octal, este diseño requiere menos terminales y, lo que es más importante, es fácil extenderla a memorias grandes.

Aunque la memoria de la figura 3-29 podría parecer complicada a primera vista, en realidad es muy sencilla gracias a su estructura regular. Este circuito tiene ocho líneas de entrada y tres líneas de salida. Tres de las entradas son datos: I_0, I_1 e I_2; dos son para la dirección: A_0 y A_1; y tres son de control: CS para seleccionar el chip (*Chip Select*), RD para distinguir entre lectura (*ReaD*) y escritura, y OE para habilitar la salida (*Output Enable*). Las tres salidas son de datos: D_0, D_1 y D_2. En principio, esta memoria podría colocarse en un paquete de 14 terminales, incluida la de potencia y la de tierra, en comparación con 20 terminales para el flip-flop octal.

Para seleccionar este chip de memoria, la lógica externa debe poner CS en 1 y también poner RD en 1 si se desea leer o en 0 si se desea escribir. Las dos líneas de dirección deben ajustarse de modo que indiquen cuál de las cuatro palabras de 3 bits se desea leer o escribir.



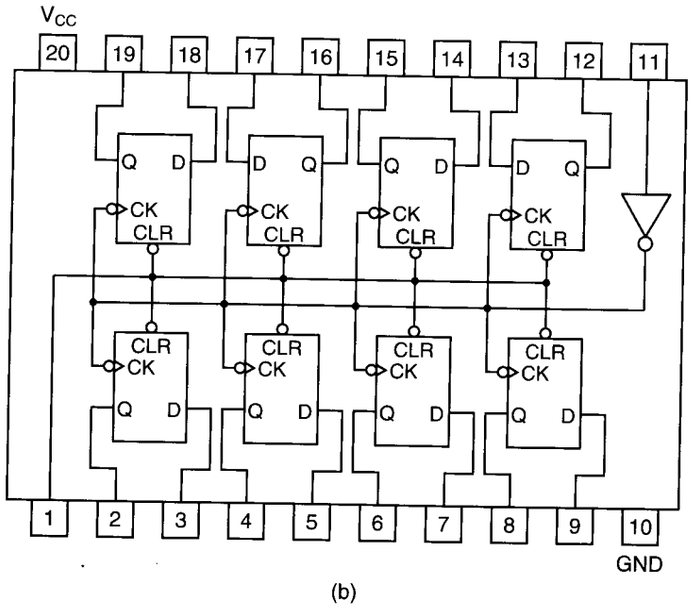


Figura 3-28. (a) Flip-flop D dual. (b) Flip-flop octal.

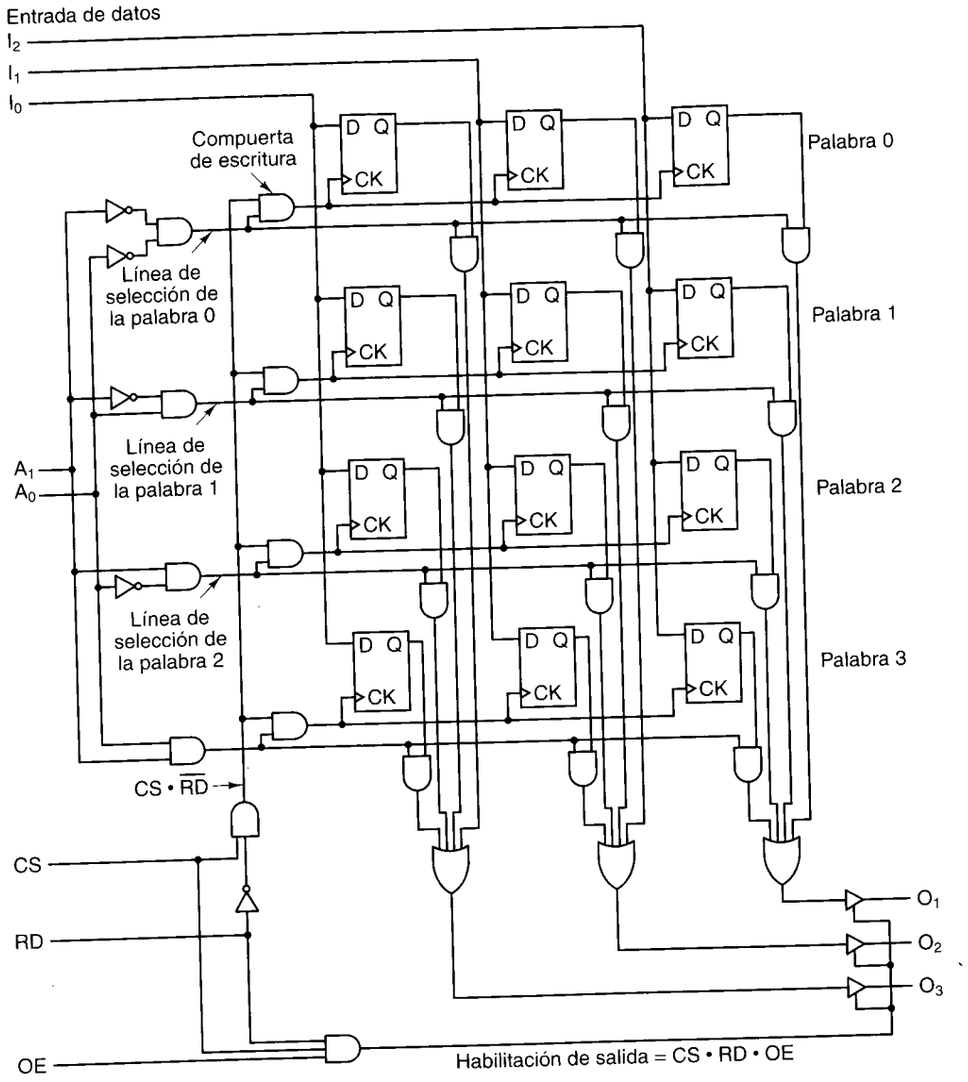


Figura 3-29. Diagrama lógico de una memoria 4 × 3. Cada renglón es una de las cuatro palabras de 3 bits. Una operación de lectura o escritura siempre lee o escribe una palabra completa.

En el caso de una operación de lectura, las líneas de entrada de datos no se usan, pero la palabra seleccionada se coloca en las líneas de salida de datos. Para una operación de escritura, los bits presentes en las líneas de entrada de datos se cargan en la palabra de memoria seleccionada; las líneas de salida de datos no se usan.

Ahora examinemos con detenimiento la figura 3-29 para observar el modo de funcionamiento. Las cuatro compuertas AND de selección de palabra a la izquierda de la memoria forman un decodificador. Los inversores de entrada se colocaron de modo que cada compuerta se habilite (su salida sea alta) con una dirección diferente. Cada compuerta alimenta una línea de selección de palabra, de arriba hacia abajo, para las palabras 0, 1, 2 y 3. Si el chip se seleccionó para escritura, la línea vertical rotulada CS· $\overline{\text{RD}}$ estará alta y habilitará una de las cuatro compuertas de escritura, dependiendo de cuál línea de selección de palabra esté alta. La salida de la compuerta de escritura alimenta todas las señales CK para la palabra seleccionada, cargando los datos de entrada en los flip-flops de esa palabra. Sólo se efectúa una escritura si CS está alta y RD está baja, y aun entonces sólo se escribe la palabra seleccionada por A_0 y A_1; las demás palabras no se modifican.

La lectura es similar a la escritura. La decodificación de la dirección es exactamente la misma que en una escritura, pero ahora la línea CS · $\overline{\text{RD}}$ está baja, lo que deshabilita todas las compuertas de escritura y ninguno de los flip-flops se modifica. En vez de ello, la línea de selección de palabra habilita las compuertas AND conectadas a los bits *Q* de la palabra seleccionada. Así, la palabra seleccionada pasa sus datos a las compuertas OR de cuatro entradas que están en la parte inferior de la figura, mientras que las otras tres palabras pasan ceros. Por consiguiente, la salida de las compuertas OR es idéntica al valor almacenado en la palabra seleccionada. Las tres palabras que no se seleccionaron no contribuyen a la salida.

Aunque podríamos haber diseñado un circuito en el que las tres compuertas OR se alimentaran directamente a las tres líneas de datos de salida, hacerlo así a veces causa problemas. En particular, hemos mostrado que las líneas de entrada de datos y las de salida de datos son diferentes, pero en las memorias reales se emplean las mismas líneas. Si hubiéramos conectado las compuertas OR a las líneas de salida de datos, el chip trataría de producir datos (o sea, forzar cada línea a tener un valor específico) aun en las escrituras, y esto interferiría los datos de entrada. Por esta razón, es deseable tener una forma de conectar las compuertas OR a las líneas de salida de datos durante las lecturas pero desconectarlas totalmente durante las escrituras. Lo que necesitamos es un conmutador electrónico que puede establecer o romper una conexión en unos cuantos nanosegundos.

Por fortuna, existen tales conmutadores. En la figura 3-30(a) se muestra el símbolo de un **buffer no inversor**. Este buffer tiene una entrada de datos, una salida de datos y una entrada de control. Cuando la entrada de control está en estado alto, el buffer actúa como un alambre, como se muestra en la figura 3-30(b). Cuando la entrada de control está baja, el buffer actúa como un circuito abierto, como se muestra en la figura 3-30(c); es como si alguien desconectara la salida de datos del resto del circuito con una pinza de corte. La diferencia respecto a la técnica de la pinza de corte es que la conexión se puede restaurar después en unos cuantos nanosegundos con sólo hacer que la señal de control sea alta otra vez.

La figura 3-30(d) muestra un **buffer inversor**, que actúa como un inversor normal cuando la señal de control es alta y desconecta la salida del circuito cuando la señal de control es baja. Ambos tipos de buffers son **dispositivos de tres estados**, porque pueden producir 0, 1 o ninguna de las dos (circuito abierto). Los buffers también amplifican las señales, por lo que pueden alimentar a muchas entradas simultáneamente. En muchos casos se usan buffers en los circuitos por esta razón, aun cuando no se necesitan sus propiedades de interrupción.

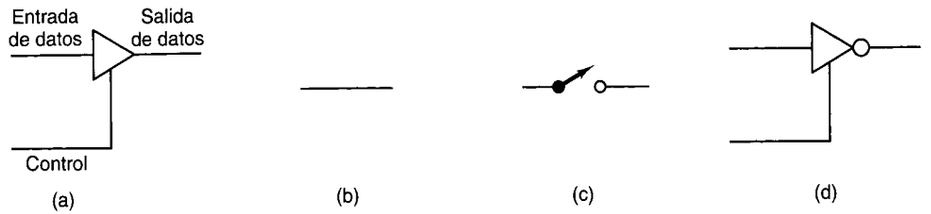


Figura 3-30. (a) Buffer no inversor. (b) Efecto de (a) cuando el control está alto. (c) Efecto de (a) cuando el control está bajo. (d) Buffer inversor.

Volviendo al circuito de memoria, ahora deberá quedar claro para qué sirven los tres buffers no inversores en las líneas de salida de datos. Cuando CS, RD y OE están todas altas, la señal de habilitación de salida también está alta; esta señal habilita los buffers, y se coloca una palabra en las líneas de salida. Cuando alguna de las señales CS, RD u OE está baja, las salidas de datos se desconectan del resto del circuito.

3.3.5 Chips de memoria

Lo bueno de la memoria de la figura 3-29 es que fácilmente se puede extender a un mayor tamaño. Tal como la dibujamos, la memoria es de 4×3, es decir, cuatro palabras de tres bits cada una. Para extenderla a 4×8 basta con añadir cinco columnas más de cuatro flip-flops cada una, además de otras cinco líneas de entrada y otras cinco líneas de salida. Para convertir esta memoria de 4×3 a 8×3 necesitaremos añadir otras cuatro filas de tres flip-flops cada una, además de una línea de dirección A_2. Con este tipo de estructura, el número de palabras en la memoria debe ser una potencia de 2 si se desea una eficiencia máxima, pero el número de bits en una palabra puede ser el que se desee.

Puesto que la tecnología de circuitos integrados se presta a fabricar chips cuya estructura interna sea un patrón bidimensional repetitivo, los chips de memoria son una aplicación ideal de esa tecnología. Al mejorar la tecnología, el número de bits que se pueden colocar en un chip aumenta continuamente, duplicándose por lo regular cada 18 meses (ley de Moore). Los chips más grandes no siempre hacen que los más pequeños se vuelvan obsoletos, debido a las diferentes características de capacidad, rapidez, potencia, precio y facilidad de conexión. En general, los chips más grandes en el mercado tienen precios inflados y por tanto son más costosos por bit que los chips más viejos y pequeños.

Para un tamaño de memoria dado, hay diversas formas de organizar el chip. La figura 3-31 muestra dos posibles organizaciones de un chip de 4 Mbit: $512K \times 8$ y 4096×1. (Por cierto, los tamaños de los chips de memoria generalmente se dan en bits, no en bytes, y adoptaremos esa convención aquí.) En la figura 3-31(a) se requieren 19 líneas de dirección para direccionar uno de los 2^{19} bytes, y se requieren ocho líneas de datos para cargar o almacenar el byte seleccionado.

Es pertinente incluir aquí una nota sobre terminología. En algunas terminales, el voltaje alto hace que ocurra alguna acción. En otras, el voltaje bajo es el que causa la acción. Para evitar confusiones, diremos siempre que una señal se **habilita o aserta** (en lugar de decir que

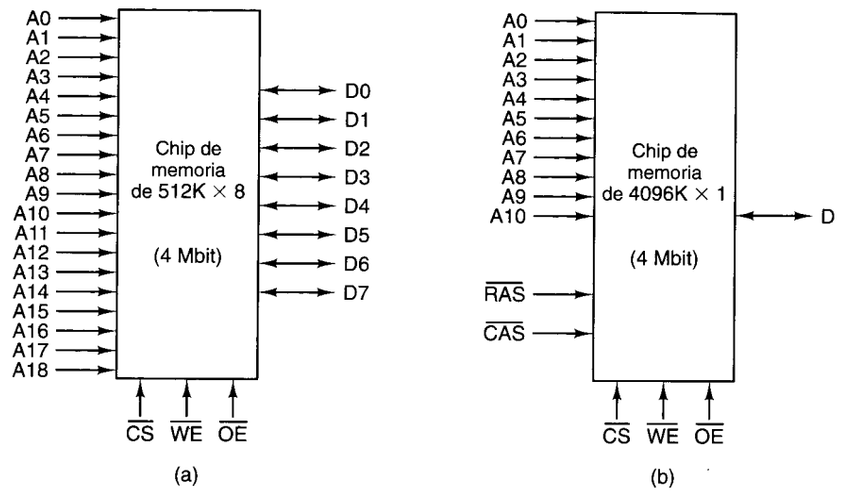


Figura 3-31. Dos formas de organizar un chip de memoria de 4 Mbit.

se vuelve alta o se vuelve baja) para indicar que se ajusta de modo que cause alguna acción. Así, en algunas terminales, habilitar la señal significa hacerla alta; en otras, significa hacerla baja. Las terminales que se habilitan bajando la señal reciben nombres que llevan una barra o testa arriba. Por tanto, una señal llamada CS se habilita llevándola a nivel alto, pero una llamada $\overline{\text{CS}}$ se habilita llevándola a nivel bajo. Lo contrario de habilitar es **deshabilitar**. Cuando no está sucediendo nada especial, las terminales están deshabilitadas.

Volvamos ahora a nuestro chip de memoria. Puesto que una computadora normalmente tiene muchos chips de memoria, se requiere una señal para seleccionar el chip que se necesita en ese momento, de modo que responda y los demás no lo hagan. Se incluye la señal $\overline{\text{CS}}$ (*Chip Select*) para este fin. La señal se habilita para activar el chip. Además, se necesita una forma de distinguir las lecturas de las escrituras. La señal $\overline{\text{WE}}$(*Write Enable*, habilitar escritura) sirve para indicar que se van a escribir datos, no a leerse. Por último, la señal $\overline{\text{OE}}$(*Output Enable*) se habilita para alimentar las señales de salida; si no está habilitada, la salida del chip se desconecta del circuito.

En la figura 3-31(b) se emplea un esquema de direccionamiento distinto. Internamente, este chip está organizado como una matriz 2048×2048 de celdas de un bit, lo que da 4 Mbit. Para direccionar el chip, primero se selecciona una localidad (renglón) colocando su número de 11 bits en las terminales de dirección. Luego se habilita la señal $\overline{\text{RAS}}$(*Row Address Strobe*, habilitar dirección de fila). Después, se coloca un número de columna en las terminales de dirección y se habilita $\overline{\text{CAS}}$(*Column Address Strobe*, habilitar dirección de columna). El chip responde aceptando o produciendo un bit de datos.

Los chips de memoria grandes a menudo se construyen como matrices $n \times n$ que se direccionan por renglón y columna. Esta organización reduce el número de terminales requeridas pero también hace más lento el direccionamiento del chip, ya que se necesitan dos ciclos de direccionamiento, uno para el renglón y otro para la columna. Para recuperar algo de la velocidad que se pierde por este diseño, algunos chips de memoria pueden aceptar una

dirección de fila seguida de una sucesión de direcciones de columna para accesar bits consecutivos de un renglón.

Hace años, los chips de memoria más grandes en general estaban organizados como la figura 3-31(b). Ahora que las palabras de memoria han crecido de 8 bits a 32 bits o más, los chips con una anchura de un bit dejaron de ser prácticos. Para construir una memoria con palabras de 32 bits a partir de chips de 4096K × 1 se requieren 32 chips en paralelo. Estos 32 chips tienen una capacidad total de por lo menos 16 MB. En cambio, si se usan chips de 512K × 8 sólo se requieren cuatro chips en paralelo, y pueden tenerse memorias pequeñas, de 2 MB o más. Para evitar tener 32 chips de memoria, casi todos los fabricantes de chips producen familias con anchuras de 1, 4, 8 y 16 bits.

3.3.6 Las memorias RAM y las ROM

Todas las memorias que hemos estudiado hasta aquí pueden leerse y escribirse. Tales memorias se llaman **RAM** (memoria de acceso aleatorio, *Random Access Memory*), aunque el nombre es engañoso porque es posible accesar aleatoriamente a todos los chips, no sólo los de RAM, pero el término ya está demasiado bien establecido como para deshacerse de él a estas alturas. Hay dos variedades de RAM, estática y dinámica. Una **RAM estática** (**SRAM**, *Static RAM*)se construye internamente empleando circuitos similares a nuestro flip-flop D básico. Estas memorias tienen la propiedad de que su contenido se conserva en tanto se sigue alimentando el circuito: segundos, minutos, horas, y hasta días. Las RAM estáticas son muy rápidas. Un tiempo de acceso típico es de unos cuantos nanosegundos. Por esta razón, las SRAM son populares como memoria caché de nivel 2.

Las **RAM dinámicas** (**DRAM**), en cambio, no usan flip-flops. En vez de ello, una RAM dinámica es una matriz de celdas, cada una de las cuales contiene un transistor y un diminuto condensador. Los condensadores pueden cargarse o descargarse, lo que permite almacenar ceros y unos. Puesto que la carga eléctrica tiende a fugarse, cada bit de una RAM dinámica debe **refrescarse** (cargarse de nuevo) cada pocos milisegundos para evitar que los datos se pierdan. Puesto que el proceso de refresco corre a cargo de la lógica externa, las RAM dinámicas requieren interfaces más complejas que las estáticas, aunque en muchas aplicaciones esta desventaja se compensa por su mayor capacidad.

Puesto que las RAM dinámicas sólo necesitan un transistor y un condensador por bit (en comparación con seis transistores por bit para la mejor RAM estática), las RAM dinámicas tienen una densidad muy alta (muchos bits por chip). Por esta razón, las memorias principales casi siempre se construyen con RAM dinámicas. Sin embargo, esta gran capacidad tiene un precio: las RAM dinámicas son lentas (decenas de nanosegundos). La combinación de una caché de RAM estática y una memoria principal de RAM dinámica intenta combinar las propiedades ventajosas de cada una.

Existen varios tipos de chips de RAM dinámica. El tipo más antiguo que continúa en uso es la DRAM **FPM** (**modo de página rápida**, *Fast Page Mode*). Internamente, la organización es una matriz de bits; el hardware presenta una dirección de renglón y luego recorre las direcciones de columna, como describimos con $\overline{RAS}$ y $\overline{CAS}$ en el contexto de la figura 3-31(b).

La DRAM FPM está siendo reemplazada gradualmente por la DRAM **edo** (**salida de datos extendida**, *Extended Data Output*), que permite iniciar una segunda referencia a la

memoria antes de que la anterior se haya completado. Este sencillo uso de conductos no hace que una sola referencia a la memoria sea más rápida, pero sí mejora el ancho de banda de la memoria, que produce más palabras por segundo.

Tanto los chips FPM como los EDO son asincrónicos, lo que significa que las líneas de dirección y de datos no están controladas por un mismo reloj. En contraste, la **SDRAM** (**DRAM sincrónica**) es un híbrido de RAM estática y dinámica y es controlado por un solo reloj sincrónico. La SDRAM se usa a menudo en las cachés grandes y podría ser la tecnología preferida para las memorias principales en el futuro.

Las RAM no son el único tipo de chips de memoria. En muchas aplicaciones, como juguetes, aparatos domésticos y automóviles, el programa y algunos de los datos deben conservarse en la memoria aunque se interrumpa el suministro de electricidad. Además, una vez instalados, ni el programa ni los datos llegan a modificarse. Estos requisitos han dado pie a la creación de las **ROM** (memorias sólo de lectura, *Read-Only Memories*), que no pueden modificarse ni borrarse, intencionalmente o por accidente. Los datos de una ROM se insertan durante su fabricación, básicamente exponiendo un material fotosensible a través de una máscara que contiene el patrón de bits deseado y luego eliminando por grabado la superficie expuesta (o la no expuesta). La única forma de modificar el programa de una ROM es cambiar todo el chip.

Las ROM son mucho más económicas que las RAM cuando se producen en grandes cantidades, pues así el costo de elaborar la máscara se diluye. Sin embargo, estas memorias son inflexibles, porque no pueden modificarse después de su fabricación, y el tiempo que tarda en surtirse un pedido de ROM puede ser de semanas. A fin de facilitar el desarrollo de productos nuevos basados en ROM, se inventó la **PROM** (ROM programable, *Programmable ROM*). Una PROM es parecida a una ROM, excepto que puede programarse (una vez) en el campo, lo que elimina el retraso por surtido. Muchas PROM contienen una matriz de diminutos fusibles. Un fusible específico puede quemarse seleccionando su renglón y columna y aplicando después un voltaje alto a una terminal especial del chip.

El siguiente avance en este campo fue la **EPROM** (PROM borrable, *Erasable PROM*), que no sólo puede programarse en el campo, sino también borrarse en el campo. Cuando la ventana de cuarzo de una EPROM se expone a una luz ultravioleta intensa durante 15 minutos, todos los bits se ponen en 1. Si se esperan muchos cambios durante el ciclo de diseño, las EPROM son mucho más económicas que las PROM porque pueden reutilizarse. Las EPROM suelen tener la misma organización que las RAM estáticas. La EPROM 27C040 de 4 bits, por ejemplo, emplea la organización de la figura 3-31(a), que es típica de una RAM estática.

Aun mejor que la EPROM es la **EEPROM**, que puede borrarse por aplicación de pulsaciones, sin tener que colocarse en una cámara especial para exponerla a luz ultravioleta. Además, una EEPROM puede reprogramarse instalada, mientras que una EPROM tiene que insertarse en un dispositivo especial para programar las EPROM. En el aspecto negativo, las EEPROM suelen ser mucho más pequeñas (por lo regular 1/64 que las EPROM comunes) y dos veces más lentas. Las EEPROM no pueden competir con las DRAM ni las SRAM porque son 10 veces más lentas, 100 veces más pequeñas y mucho más costosas; sólo se utilizan en situaciones en las que su no volatilidad es crucial.

Una clase de EEPROM más reciente es la **memoria flash**. A diferencia de la EPROM, que se borra por exposición a luz ultravioleta, y la EEPROM, que se puede borrar por byte, la

memoria flash puede borrarse y reescribirse por bloque. Al igual que la EEPROM, la memoria flash puede borrarse sin sacarla del circuito. Varios fabricantes producen pequeñas tarjetas de circuitos impresos con decenas de megabytes de memoria flash para usarse como cinta de video y almacenar imágenes en cámaras digitales, además de muchos otros usos. Algún día podrán usarse las memorias flash para sustituir a los discos, lo cual representaría un avance importante, en vista de sus tiempos de acceso de 100 ns. El principal problema de ingeniería en la actualidad es que las memorias flash se desgastan después de unos 10,000 borrados, mientras que los discos duran años, por más frecuentemente que se reescriban. En la figura 3-32 se presenta un resumen de los diversos tipos de memoria.

Tipo	Categoría	Borrado	Alterable por byte	Volátil	Uso típico
SRAM	Lectura/escritura	Eléctrico	Sí	Sí	Caché nivel 2
DRAM	Lectura/escritura	Eléctrico	Sí	Sí	Memoria principal
ROM	Sólo lectura	Imposible	No	No	Aparatos en vol. grandes
PROM	Sólo lectura	Imposible	No	No	Equipos en vol. pequeños
EPROM	Princ. lectrua	Luz UV	No	No	Prototipos de dispositivos
EEPROM	Princ. lectura	Eléctrico	Sí	No	Prototipos de dispositivos
Flash	Lectura/escritura	Eléctrico	No	No	Película para cámara digital

Figura 3-32. Comparación de diversos tipos de memoria.

3.4 CHIPS DE CPU Y BUSES

Ahora que poseemos información acerca de los chips SSI, MSI y de memoria, podemos comenzar a juntar todas las piezas y examinar sistemas completos. En esta sección, examinaremos primero algunos aspectos generales de las CPU desde la perspectiva del nivel de lógica digital, incluida la **conexión de terminales** o **pinout** (el significado de las señales de las distintas terminales). Puesto que las CPU están íntimamente relacionadas con el diseño de los buses que usan, también proporcionaremos en esta sección una introducción al diseño de buses. En secciones posteriores daremos ejemplos detallados de CPU y de sus buses.

3.4.1 Chips de CPU

Todas las CPU modernas están contenidas en un solo chip. Esto hace que su interacción con el resto del sistema esté bien definida. Cada chip de CPU tiene un conjunto de terminales, a través de las cuales debe pasar toda su comunicación con el mundo exterior. Algunas terminales proporcionan señales de la CPU; otras aceptan señales del mundo exterior; algunas pueden hacer las dos cosas. Si entendemos la función de todas las terminales, podremos conocer cómo la CPU interactúa con la memoria y los dispositivos de E/S en el nivel de lógica digital.

Las terminales de un chip de CPU se pueden dividir en tres tipos: dirección, datos y control. Estas terminales se conectan a terminales similares de los chips de memoria y de E/S a través de una colección de alambres paralelos llamado bus. Para buscar una instrucción, la CPU primero coloca la dirección de memoria de esa instrucción en sus terminales de dirección, y luego habilita una o más líneas de control para informar a la memoria que quiere leer (por ejemplo) una palabra. La memoria contesta colocando la palabra solicitada en las terminales de datos de la CPU y habilitando una señal para indicar que ya terminó. Cuando la CPU ve esta señal, acepta la palabra y ejecuta la instrucción.

La instrucción podría requerir leer o escribir palabras de datos, en cuyo caso se repite todo el proceso para cada palabra adicional. Más adelante veremos los detalles de cómo funciona la lectura y la escritura. Por ahora, lo que es importante entender es que la CPU se comunica con la memoria y los dispositivos de E/S tanto transmitiendo como recibiendo señales en sus terminales. No es posible ninguna otra comunicación.

Dos de los parámetros fundamentales que determinan el desempeño de una CPU son el número de líneas de dirección y el número de líneas de datos. Un chip que tiene m líneas de dirección puede direccionar hasta 2^m localidades de memoria. Los valores comunes de m son 16, 20, 32 y 64. De igual manera, un chip con n líneas de datos puede leer o escribir una palabra de n bits en una sola operación. Los valores comunes de n son 8, 16, 32, 36 y 64. Una CPU con ocho líneas de datos requiere cuatro operaciones para leer una palabra de 32 bits, mientras que una con 32 líneas de datos puede realizar la misma tarea en una sola operación. Así, el chip con 32 terminales de datos es mucho más rápido, pero siempre es más costoso.

Además de las líneas de dirección y de datos, toda CPU tiene algunas líneas de control, las cuales regulan el flujo y la temporización de los datos que entran en la CPU y salen de ella, además de tener varios otros usos. Todas las CPU tienen terminales para la alimentación (casi siempre +3.3 volts o + 5 volts), tierra y una señal de reloj (una onda cuadrada), pero las demás terminales varían considerablemente de un chip a otro. No obstante, las líneas de control pueden agruparse a grandes rasgos en las siguientes categorías principales:

1. Control de bus.
2. Interrupciones.
3. Arbitraje de bus.
4. Señalización de coprocesador.
5. Situación.
6. Diversos.

Describiremos brevemente cada una de estas categorías a continuación. Cuando examinemos los chips Pentium II, UltraSPARC II y picoJava II más adelante, daremos más detalles. En la figura 3-33 se muestra un chip de CPU genérico que utiliza estos grupos de señales.

Las terminales de control de bus son principalmente salidas de la CPU al bus (por tanto, entradas a los chips de memoria y de E/S) que dicen si la CPU quiere leer o escribir en la

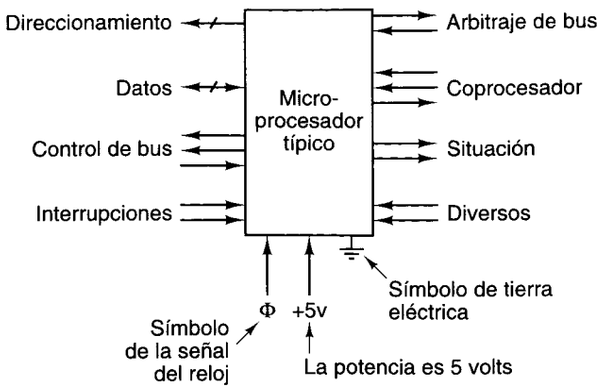


Figura 3-33. Conexión lógica de terminales de una CPU genérica. Las flechas indican señales de entrada y de salida. Las líneas diagonales cortas indican que se usan varias terminales. En el caso de una CPU específica, se daría un número para indicar cuántas.

memoria o hacer alguna otra cosa. La CPU utiliza estas terminales para controlar el resto del sistema y decirle lo que quiere hacer.

Las terminales de interrupciones son entradas a la CPU provenientes de dispositivos de E/S. En casi todos los sistemas, la CPU puede ordenarle a un dispositivo de E/S que inicie una operación y luego dedicarse a hacer algo útil mientras el lento dispositivo de E/S está haciendo su trabajo. Una vez completada la E/S, el chip controlador de E/S habilita una señal en una de estas terminales para interrumpir a la CPU y pedirle que atienda al dispositivo de E/S (digamos, para ver si ocurrieron errores de E/S). Algunas CPU tienen una terminal de salida para indicar que recibieron la señal de interrupción.

Las terminales de arbitraje de bus son necesarias para regular el tráfico en el bus e impedir que dos dispositivos traten de usarlo al mismo tiempo. Para fines de arbitraje, la CPU cuenta como un dispositivo.

Algunos chips de CPU están diseñados para operar con coprocesadores, como chips de punto flotante, de gráficos o de otro tipo. Para facilitar la comunicación entre la CPU y el coprocesador, se incluyen terminales especiales para realizar y atender diversas solicitudes.

Además de estas señales, algunas CPU tienen varias terminales de uso diverso. Algunas proporcionan o aceptan información de estado, otras son útiles para restablecer la computadora, y otras más están presentes para asegurar la compatibilidad con chips de E/S antiguos.

3.4.2 Buses de computadora

Un **bus** es un camino eléctrico común entre varios dispositivos. Los buses pueden clasificarse por su función. Pueden usarse internamente en la CPU para transportar datos a y de la ALU, o externos a la CPU para conectarla con la memoria o los dispositivos de E/S. Cada tipo de bus tiene sus propios requisitos y propiedades. En esta sección y en las que siguen nos con-

centraremos en los buses que conectan a la CPU con la memoria y los dispositivos de E/S. En el siguiente capítulo examinaremos con más detenimiento los buses internos de la CPU.

Las primeras computadoras personales tenían un solo bus externo o **bus de sistema** que constaba de 50 a 100 hilos de cobre paralelos grabados en la tarjeta madre, con conectores espaciados a intervalos regulares para insertar tarjetas de memoria y de E/S. Las computadoras personales modernas generalmente tienen un bus de propósito especial entre la CPU y la memoria y (por lo menos) un bus más para los dispositivos de E/S. En la figura 3-34 se ilustra un sistema mínimo, con un bus de memoria y un bus de E/S.

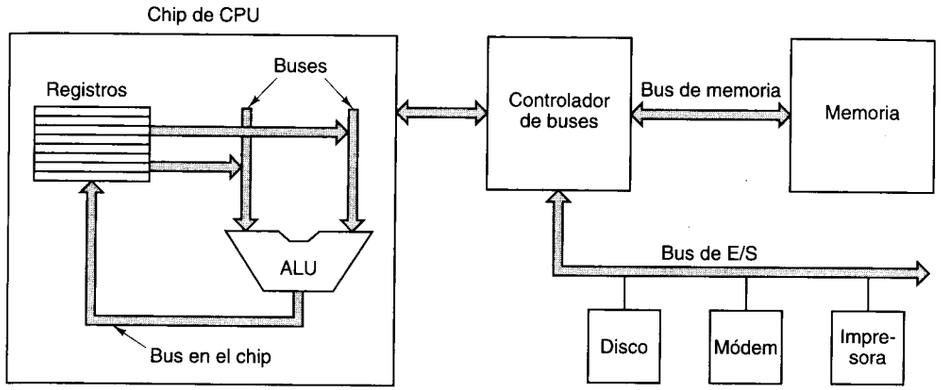


Figura 3-34. Sistema de cómputo con varios buses.

En la literatura los buses a veces se dibujan como flechas gruesas, como en esta figura. La distinción entre una flecha gruesa y una sola línea atravesada por una línea diagonal corta y un número de bits junto a ella es sutil. Cuando todos los bits son del mismo tipo, digamos bits de dirección o bits de datos, es común usar la representación con la línea diagonal corta. Cuando intervienen líneas de dirección, datos y control, la flecha gruesa es más común.

Si bien los diseñadores de la CPU están en libertad de usar el tipo de bus que les plazca dentro del chip, si se quiere que tarjetas diseñadas por terceros puedan conectarse al bus del sistema deben existir reglas bien definidas que digan cómo funciona el bus, mismas que todos los dispositivos conectados a él deberán obedecer. Estas reglas son el **protocolo de bus**. Además, debe haber especificaciones mecánicas y eléctricas para que las tarjetas de terceros ajusten en la caja de tarjetas y tengan conectores que coincidan con los de la tarjeta madre tanto mecánicamente como en términos de voltajes, temporización, etcétera.

En el mundo de las computadoras se usan ampliamente varios buses. Unos de los más conocidos, tanto actuales como históricos (con ejemplos) son el Ómnibus (PDP-8), Unibus (PDP-11), Multibus (8086), el bus de IBM PC (PC/XT), el bus ISA (PC/AT), el bus EISA (80386), Microchannel (PS/2), el bus PCI (muchas PC), el bus SCSI (muchas PC y estaciones de trabajo), Nubus (Macintosh), Bus Serial Universal (PC modernas), FireWire (electrónica para consumidor), el bus VME (equipo de laboratorio de física) y el bus CAMAC (física de alta energía). Es probable que el mundo fuera un mejor lugar si todos menos uno se

desaparecieran repentinamente de la faz de la Tierra (está bien, de acuerdo, ¿qué tal todos menos dos?). Desafortunadamente, la estandarización en esta área es poco probable porque ya se ha invertido mucho en todos estos sistemas incompatibles.

Iniciemos ahora nuestro estudio del funcionamiento de los buses. Algunos dispositivos que se conectan a un bus son activos y pueden iniciar transferencias de bus, mientras que otros son pasivos y esperan solicitudes. Los activos se llaman **amos** o **maestros**; los pasivos se llaman **esclavos**. Cuando la CPU ordena a un controlador de disco que lea o escriba un bloque, la CPU está actuando como amo y el controlador de disco está actuando como esclavo. Sin embargo, unos momentos después el controlador de disco podría actuar como amo al ordenar a la memoria que acepte las palabras que está leyendo de la unidad de disco. En la figura 3-35 se enumeran varias combinaciones típicas de amo y esclavo. En ninguna circunstancia la memoria puede ser un amo.

Amo	Esclavo	Ejemplo
CPU	Memoria	Traer instrucciones y datos
CPU	Dispositivo E/S	Iniciar transferencia de datos
CPU	Coprocesador	Delegación de instrucciones de la CPU al coprocesador
E/S	Memoria	DMA (acceso directo a memoria)
Coprocesador	CPU	Obtención de operandos de la CPU por el coprocesador

Figura 3-35. Ejemplos de amos (controladores) y esclavos de bus.

Las señales binarias que los dispositivos de cómputo producen, a menudo no son lo bastante fuertes como para impulsar un bus, sobre todo si éste es relativamente largo y tiene conectados muchos dispositivos. Por esta razón, casi todos los amos de bus se conectan al bus con un chip llamado **controlador** o ***driver* de bus**, que en esencia es un amplificador digital. Así mismo, casi todos los esclavos se conectan al bus con un **receptor de bus**. En el caso de dispositivos que pueden actuar como amo y como esclavo, se emplea un chip combinado llamado **transceptor de bus**. Estos chips de interfaz con el bus suelen ser dispositivos de tres estados, para poder "flotar" (desconectarse) cuando no se necesitan, o estar conectados de forma un tanto distinta, llamada **de colector abierto**, cuyo efecto es el mismo. Cuando dos o más dispositivos de una línea de colector abierto habilitan la línea al mismo tiempo, el resultado es un OR booleano de todas las señales. Este sistema se conoce como **OR alambrado**. En la mayor parte de los buses, algunas de las líneas son de tres estados y otras, que necesitan la propiedad de OR alambrado, son de colector abierto.

Al igual que una CPU, un bus también tiene líneas de dirección, de datos y de control. Sin embargo, no necesariamente hay una correspondencia uno a uno entre las terminales de la CPU y las señales del bus. Por ejemplo, algunas CPU tienen tres terminales que codifican si se está efectuando una lectura de memoria, una escritura de memoria, una lectura de E/S, una escritura de E/S o alguna otra operación. Un bus típico podría tener una línea para lectura de memoria, una segunda para escritura de memoria, otra para lectura de E/S, una cuarta para escritura de E/S, etc. En tal caso se necesitaría un chip decodificador entre la CPU y el bus para coordinar ambos lados, es decir, para convertir la señal codificada de tres bits en señales individuales que se alimenten a las líneas del bus.

El diseño y la operación de buses son temas lo bastante complejos como para que se hayan escrito libros enteros sobre ellos (Stanley y Anderson, 1995b; Solari, 1993; y Solari y Willse, 1998). Los aspectos principales del diseño de un bus son su anchura, temporización, arbitraje y operaciones. Cada una de estas cuestiones tiene un impacto considerable sobre la rapidez y el ancho de banda del bus. Examinaremos cada uno de estos aspectos en las cuatro secciones que siguen.

3.4.3 Ancho de bus

El ancho de bus es el parámetro de diseño más obvio, Cuantas más líneas de dirección tenga un bus, más memoria podrá direccionar la CPU directamente. Si un bus tiene n líneas de dirección, una CPU podrá usarlo para direccionar 2^n localidades de memoria distintas. Para manejar memorias grandes los buses necesitan muchas líneas de dirección. Sencillo, ¿no?

El problema es que los buses anchos necesitan más líneas que los angostos; además, ocupan más espacio físico (por ejemplo, en la tarjeta madre) y necesitan conectores más grandes. Todos estos factores hacen que el bus sea más caro. Por tanto, hay un equilibrio entre tamaño máximo de memoria y costo del sistema. Un sistema con un bus de direcciones de 64 líneas y 2^{32} bytes de memoria costará más que uno con 32 líneas de dirección y los mismos 2^{32} bytes de memoria. La posibilidad de una expansión futura no es gratuita.

El resultado de esta observación es que muchos diseñadores de sistemas tienden a ser miopes, y las consecuencias pueden ser graves. La IBM PC original contenía una CPU 8088 y un bus de direcciones de 20 bits, como se muestra en la figura 3-36(a). Estos 20 bits permitían a la PC direccionar 1 MB de memoria.

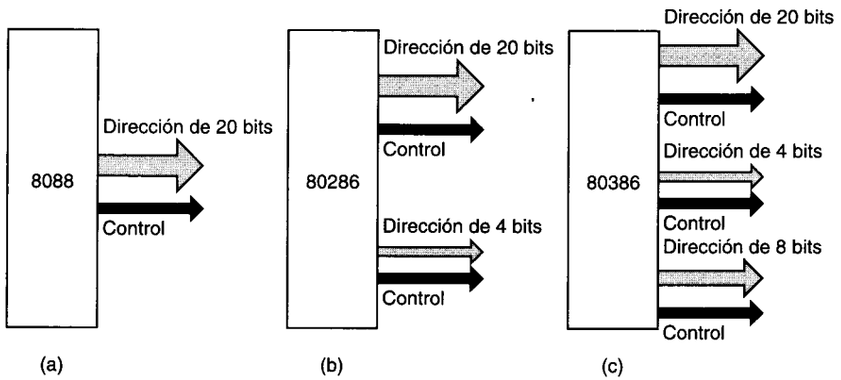


Figura 3-36. Crecimiento de un bus de direcciones con el tiempo.

Cuando surgió el chip de CPU (el 80286), Intel decidió incrementar el espacio de direcciones a 16 MB, por lo que fue necesario añadir cuatro líneas más de bus de dirección (sin meterse con las 20 originales, por razones de compatibilidad con modelos anteriores), como se muestra en la figura 3-36(b). Lamentablemente, fue necesario añadir más líneas de control para ocuparse de las nuevas líneas de dirección. Cuando apareció el 80386, se añadieron

otras ocho líneas de dirección, junto con más líneas de control, como se muestra en la figura 3-36(c). El diseño resultante (el bus EISA) es mucho menos elegante de lo que habría sido si hubiera tenido 32 líneas desde el principio.

El número de líneas de dirección no es lo único que tiende a aumentar con el tiempo; lo mismo sucede con el número de líneas de datos, pero por una razón un poco distinta. Hay dos formas de incrementar el ancho de banda de datos de un bus: reducir el tiempo de ciclo de bus (más transferencias por segundo) o incrementar la capacidad del bus de datos (más bits/transferencia). Es posible acelerar el bus, pero no es fácil porque las señales de las diferentes líneas viajan a velocidades ligeramente distintas, problema que se conoce como **sesgo de bus**. Cuanto más rápido es el bus, más importante es el efecto del sesgo de bus.

Otro problema que surge al hacer más rápido un bus es que se vuelve incompatible con lo existente. Las tarjetas antiguas diseñadas para el bus más lento no funcionan con el nuevo. Hacer obsoletas las tarjetas viejas pone de mal humor tanto a los dueños de las tarjetas como a sus fabricantes. Por ello, la estrategia usual para mejorar el desempeño es añadir más líneas de datos, algo análogo a lo que se muestra en la figura 3-36. Sin embargo, como era de esperarse, este crecimiento incremental no produce un diseño claro al final. La IBM PC y sus sucesoras, por ejemplo, pasaron de 8 líneas de datos a 16 y luego a 32, con prácticamente el mismo bus.

Para aliviar el problema de los buses excesivamente anchos, algunos diseñadores optan por usar un **bus multiplexado**. En este diseño, en lugar de tener líneas de direcciones y de datos por separado, hay, por ejemplo, 32 líneas para ambos, direcciones y datos. Al principio de una operación de bus las líneas se usan para la dirección; más adelante, se usan para datos. En el caso de una escritura en memoria, por ejemplo, esto implica que las líneas de dirección deben configurarse y propagarse a la memoria antes de que los datos puedan colocarse en el bus. Con líneas separadas, la dirección y los datos pueden enviarse al mismo tiempo. La multiplexión de las líneas reduce la anchura (y el costo) del bus, pero hace más lento al sistema. Los diseñadores tienen que sopesar cuidadosamente todas estas opciones al tomar sus decisiones.

3.4.4 Temporización del bus

Los buses se pueden dividir en dos categorías según su temporización. Un **bus sincrónico** tiene una línea alimentada por un oscilador de cristal. La señal de esta línea consiste en una onda cuadrada cuya frecuencia generalmente está entre 5 MHz y 100 MHz. Todas las actividades del bus tardan un número entero de estos ciclos, llamados **ciclos de bus**. El otro tipo de bus, el **bus asincrónico**, no tiene un reloj maestro. Los ciclos de bus pueden tener la longitud que se requiera y no tiene que ser la misma entre todos los pares de dispositivos. A continuación examinaremos los dos tipos de buses.

Buses sincrónicos

Como ejemplo del funcionamiento de un bus sincrónico, considere la temporización de la figura 3-37(a). En este ejemplo, usaremos un reloj de 40 MHz, lo que da un ciclo de bus de 25 ns. Si bien esto podría parecer algo lento en comparación con las velocidades de CPU de 500 MHz y más, pocos buses de PC existentes son mucho más rápidos. Por ejemplo, el bus

ISA que se usa en todas las PC basadas en Intel opera a 8.33 MHz, e incluso el popular bus PCI por lo regular opera a 33 MHz o bien 66 MHz. Ya explicamos las razones por las que los buses actuales son lentos: problemas técnicos de diseño como el sesgo de bus, y la necesidad de compatibilidad con modelos anteriores.

En nuestro ejemplo supondremos también que una lectura de la memoria tarda 40 ns desde el momento en que la dirección es estable. Como veremos en breve, con estos parámetros se requieren tres ciclos de bus para leer una palabra. El primer ciclo inicia en el flanco ascendente de T_1 y el tercero termina en el flanco ascendente de T_4, como se muestra en la figura. Observe que ninguno de los flancos ascendentes o descendentes se dibujó vertical, porque ninguna señal eléctrica puede cambiar su valor en un tiempo cero. En este ejemplo supondremos que una señal tarda 1 ns en cambiar. Las líneas de reloj, ADDRESS (dirección), DATA (datos), $\overline{\text{MREQ}}$ (solicitud de memoria), $\overline{\text{RD}}$ (leer) y $\overline{\text{WAIT}}$ (esperar) aparecen en la misma escala de tiempo.

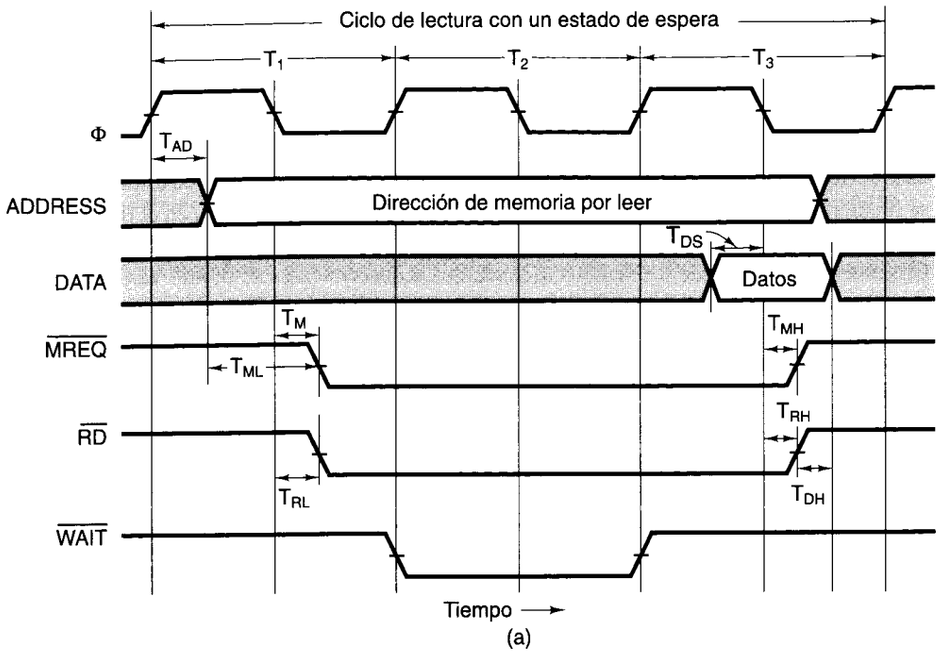
El inicio de T_1 está definido por el flanco ascendente del reloj. Ya iniciado T_1, la CPU coloca la dirección de la palabra que quiere en las líneas de dirección. Puesto que la dirección no es un valor individual, como el reloj, no podemos mostrarlo como una sola línea en la figura; en vez de ello, se muestra como dos líneas, con un cruce en el momento en que cambia la dirección. Además, el sombreado previo al cruce indica que el valor antes del cruce no es importante. Utilizando la misma convención de sombreado, vemos que el contenido de las líneas de datos no es significativo sino hasta ya bien entrado T_3.

Una vez que las líneas de dirección han tenido tiempo de estabilizarse en sus nuevos valores, se asertan $\overline{\text{MREQ}}$ y $\overline{\text{RD}}$. La primera indica que se está accediendo a la memoria (no a un dispositivo de E/S), y la segunda se habilita si se trata de una lectura y se deshabilita si se trata de una escritura. Puesto que la operación de memoria tarda 40 ns después de que la dirección es estable (ya iniciado el primer ciclo de reloj), no puede proporcionar los datos solicitados durante T_2. Para avisarle a la CPU que no los espere, la memoria habilita la línea $\overline{\text{WAIT}}$ al principio de T_2. Esta acción insertará **estados de espera** (ciclos de bus extra) hasta que la memoria termine y deshabilite $\overline{\text{WAIT}}$. En nuestro ejemplo se insertó un estado de espera (T_2) porque la memoria es demasiado lenta. Al principio de T_3, cuando la memoria está segura de que tendrá los datos durante el ciclo en curso, deshabilita $\overline{\text{WAIT}}$.

Durante la primera mitad de T_3 la memoria coloca los datos en las líneas de datos. En el flanco descendente de T_3 la CPU habilita (es decir, lee) las líneas de datos y almacena el valor en un registro interno. Después de leer los datos, la CPU invalida $\overline{\text{MREQ}}$ y $\overline{\text{RD}}$. Si es necesario, se puede iniciar otro ciclo de memoria en el siguiente flanco ascendente del reloj.

En la especificación de temporización de la figura 3-37(b) se aclaran ocho símbolos que ocurren en el diagrama de temporización. T_{AD}, por ejemplo, es el intervalo de tiempo entre el flanco ascendente del reloj T_1 y el establecimiento de las líneas de dirección. Según la especificación de temporización, $T_{AD} \leq 11$ ns. Esto implica que el fabricante de la CPU garantiza que durante cualquier ciclo de lectura la CPU producirá la dirección que se va a leer en un tiempo de 11 ns después del punto medio del flanco ascendente de T_1.

Las especificaciones de temporización también requieren que los datos estén disponibles en las líneas de datos al menos T_{DS} (5 ns) antes del flanco descendente de T_3, para que tengan tiempo de estabilizarse antes de que la CPU habilite su entrada. La combinación de



Símbolo	Parámetro	Mín	Máx	Unidad
T_{AD}	Retraso de salida de dirección		11	ns
T_{ML}	Dirección estable antes de $\overline{MREQ}$	6		ns
T_M	Retraso de $\overline{MREQ}$ desde el flanco desc. Φ en T_1		8	ns
T_{RL}	Retraso de RD desde el flanco desc. de Φ en T_1		8	ns
T_{DS}	Tiempo de preparación de datos antes del flanco desc. de Φ	5		ns
T_{MH}	Retraso de $\overline{MREQ}$ desde el flanco desc. de Φ en T_3		8	ns
T_{RH}	Retraso de $\overline{RD}$ desde el flanco desc. de Φ en T_3		8	ns
T_{DH}	Tiempo de retención de datos desde la deshabilitación de $\overline{RD}$	0		ns

(b)

Figura 3-37. (a) Temporización de lectura en un bus sincrónico. (b) Especificación de algunos tiempos críticos.

restricciones sobre T_{AD} y T_{DS} implica que, en el peor de los casos, la memoria sólo dispondrá de 62.5 – 11 – 5 = 46.5 ns desde el momento en que aparezca la dirección hasta que tenga que producir los datos. Puesto que bastan 40 ns, aun en el peor de los casos, una memoria de 40 ns

siempre puede responder durante T_3. Una memoria de 50 ns, en cambio, tendría que insertar un segundo estado de espera y responder durante T_4.

La especificación de temporización garantiza además que la dirección estará preparada al menos 6 ns antes de que se habilite $\overline{MREQ}$. Este tiempo puede ser importante si $\overline{MREQ}$ controla la selección de chip en la memoria, porque algunas memorias requieren un tiempo de preparación de la dirección previo a la selección del chip. Es obvio que el diseñador del sistema no debe escoger un chip de memoria que requiera un tiempo de preparación de 10 ns.

Las restricciones sobre T_M y T_{RL} implican que tanto $\overline{MREQ}$ como $\overline{RD}$ se asertarán en un plazo de 8 ns después del reloj descendente T_1. En el peor de los casos, el chip de memoria tendrá sólo 25 + 25 – 8 – 5 = 37 ns después de la aserción de $\overline{MREQ}$ y $\overline{RD}$ para colocar sus datos en el bus. Esta restricción es adicional respecto (e independiente) del intervalo de 40 ns que se requiere después de que la dirección se estabiliza.

T_{MH} y T_{RH} nos dicen cuánto tardan $\overline{MREQ}$ y $\overline{RD}$ en deshabilitarse una vez que se ha habilitado la entrada de los datos. Por último, T_{DH} nos dice cuánto tiempo la memoria debe mantener los datos en el bus después de que $\overline{RD}$ se ha deshabilitado. En lo que a nuestra CPU de ejemplo concierne, la memoria puede quitar los datos del bus tan pronto como $\overline{RD}$ se ha deshabilitado; sin embargo, en algunas CPU reales los datos deben mantenerse estables durante un tiempo corto después de eso.

Cabe señalar que la figura 3-37 es una versión muy simplificada de las restricciones de temporización reales. En la práctica siempre se especifican muchos más tiempos críticos. No obstante, esto da cierta idea de cómo funciona un bus sincrónico.

Como último punto que vale la pena mencionar es que las señales de control pueden ser asignadas, como altas o bajas. Compete a los diseñadores del bus determinar qué es lo más conveniente, pero la elección es esencialmente arbitraria. Uno puede considerarlo como un equivalente de hardware de la opción de un programador para representar bloques de disco libres en un mapa de bits como ceros *versus* unos.

Buses asincrónicos

Aunque es fácil trabajar con buses sincrónicos gracias a sus intervalos de tiempo discretos, también tienen ciertos problemas. Por ejemplo, todo funciona en múltiplos del reloj del bus. Si una CPU y una memoria pueden completar una transferencia en 3.1 ciclos, tendrán que prolongarlo a 4.0 porque están prohibidas las fracciones de ciclo.

Peor aún, una vez que se ha escogido un ciclo de bus, y que se han construido tarjetas de memoria y de E/S para él, es difícil aprovechar mejoras futuras en la tecnología. Por ejemplo, suponga que, unos cuantos años después de construirse el sistema de la figura 3-37, salen al mercado memorias con tiempos de acceso de 20 ns en lugar de 40 ns. Ya no sería necesario el estado de espera, y la máquina trabajaría más rápidamente. Ahora suponga que aparecen memorias de 10 ns. El desempeño del sistema no mejoraría, porque el tiempo mínimo para una lectura es de 2 ciclos con este diseño.

Expresando este hecho en términos un poco diferentes, si un bus sincrónico tiene una colección heterogénea de dispositivos, algunos rápidos y otros lentos, el bus tiene que ajustarse al más lento, y los rápidos no podrán aprovechar todo su potencial.

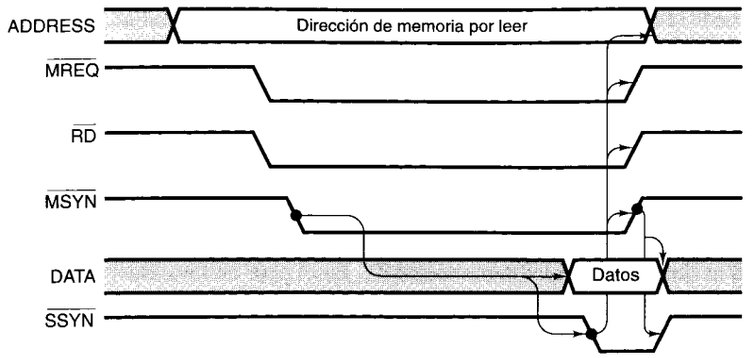


Figura 3-38. Funcionamiento de un bus asincrónico.

La tecnología mixta puede manejarse cambiando a un bus asincrónico, es decir, uno que no tenga reloj maestro, como el de la figura 3-38. En lugar de vincular todo con el reloj, una vez que el amo del bus ha habilitado la dirección, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ y todo lo demás que necesite habilitar, habilita una señal especial que llamaremos $\overline{\text{MSYN}}$ (sincronización de amo, *Master SYNchronization*). Cuando el esclavo percibe esto, realiza su trabajo lo más rápidamente que puede. Cuando termina, habilita $\overline{\text{SSYN}}$ (sincronización de esclavo, *Slave SYNchronization*).

Tan pronto como el amo reconoce que se habilitó $\overline{\text{SSYN}}$, sabe que los datos están disponibles, así que los almacena y luego deshabilita las líneas de dirección junto con MREQ, RD y $\overline{\text{MSYN}}$. Cuando el esclavo detecta la deshabilitación de $\overline{\text{MSYN}}$, sabe que el ciclo se completó y deshabilitó $\overline{\text{SSYN}}$, con lo que estamos otra vez en la situación original, con todas las señales deshabilitadas, esperando el siguiente amo.

Los diagramas de temporización de buses asincrónicos (y a veces también los sincrónicos) utilizan flechas para indicar causa y efecto, como en la figura 3-38. La habilitación de $\overline{\text{MSYN}}$ hace que las líneas de datos se habiliten y también hace que el esclavo habilite $\overline{\text{SSYN}}$. La habilitación de $\overline{\text{SSYN}}$, a su vez, causa la deshabilitación de las líneas de dirección, $\overline{\text{MREQ}}$, RD y $\overline{\text{MSYN}}$. Por último, la deshabilitación de $\overline{\text{MSYN}}$ causa la deshabilitación de SSYN, con lo que termina la lectura.

Un conjunto de señales que interactúa de esta manera se llama **saludo completo**. La parte fundamental consiste en cuatro sucesos:

1. Habilitar $\overline{\text{MSYN}}$.
2. Habilitar $\overline{\text{SSYN}}$ como respuesta a $\overline{\text{MSYN}}$.
3. Deshabilitar $\overline{\text{MSYN}}$ como respuesta a $\overline{\text{SSYN}}$.
4. Deshabilitar $\overline{\text{SSYN}}$ como respuesta a la invalidación de $\overline{\text{MSYN}}$.

Debe ser obvio que los saludos completos son independientes de la temporización. Cada suceso es causado por un suceso anterior, no por un pulso de reloj. Si un par amo-esclavo dado es lento, no afectará en absoluto a un par amo-esclavo subsecuente que sea mucho más rápido.

La ventaja de un bus asincrónico ha quedado clara, pero el hecho es que casi todos los buses son sincrónicos. La razón es que es más fácil construir un sistema sincrónico. La CPU simplemente habilita sus señales, y la memoria simplemente reacciona. No hay retroalimentación (causa y efecto), pero si los componentes se escogieron debidamente, todo funcionará bien sin saludos. Además, existe una inversión considerable en la tecnología de los buses sincrónicos.

3.4.5 Arbitraje del bus

Hasta ahora, hemos supuesto tácitamente que sólo hay un controlador de bus, la CPU. En realidad, los chips de E/S tienen que convertirse en controladores de bus para leer y escribir en la memoria, y también para causar interrupciones. Los coprocesadores también podrían necesitar convertirse en controladores de bus. Entonces surge la pregunta: "¿Qué sucede si dos o más dispositivos quieren convertirse en controlador del bus al mismo tiempo? La respuesta es que se requiere algún mecanismo de **arbitraje de bus** para evitar el caos.

Los mecanismos de arbitraje pueden ser centralizados o descentralizados. Consideremos primero el arbitraje centralizado. Una forma especialmente sencilla de arbitraje centralizado se muestra en la figura 3-39(a). En este esquema, un solo árbitro de bus determina quién sigue. Muchas CPU tienen el árbitro integrado a su chip, pero a veces se requiere un chip aparte. El bus contiene una sola línea de solicitud de OR alambrado que uno o más dispositivos pueden habilitar en cualquier momento. El árbitro no tiene forma de saber cuántos dispositivos solicitaron el bus. Las únicas categorías que puede distinguir son: hay solicitudes y no hay solicitudes.

Cuando el árbitro detecta una solicitud de bus, emite una concesión habilitando la línea de otorgamiento del bus. Esta línea está conectada a todos los dispositivos de E/S en serie, como una serie barata de luces de Navidad. Cuando el dispositivo que está más cercano físicamente al árbitro percibe la concesión, verifica si él emitió una solicitud. Si lo hizo, se apodera del bus pero no propaga la concesión línea abajo. Si no hizo una solicitud, propaga la concesión al siguiente dispositivo en la línea, que se comporta de la misma manera, y así hasta que algún dispositivo acepta la concesión y se apodera del bus. Este esquema se llama **encadenamiento circular**, y tiene la propiedad de que en la práctica asigna prioridades a los dispositivos con base en su cercanía al árbitro. El dispositivo más cercano gana.

Para supeditar las prioridades implícitas basadas en la distancia al árbitro, muchos buses tienen varios niveles de prioridad. Para cada nivel de prioridad hay una línea de solicitud de bus y una línea de otorgamiento de bus. El bus de la figura 3-39(b) tiene dos niveles, 1 y 2 (los buses reales a menudo tienen 4, 8 o 16 niveles). Cada dispositivo se conecta a uno de los niveles de solicitud de bus, y los dispositivos para los que el tiempo es más crítico se conectan a los de más alta prioridad. En la figura 3-39(b) los dispositivos 1, 2 y 4 usan la prioridad 1 mientras que los dispositivos 3 y 5 usan la prioridad 2.

Si hay solicitudes en varios niveles de prioridad al mismo tiempo, el árbitro emite una concesión sólo en el de más alta prioridad. Entre dispositivos con la misma prioridad se usa el encadenamiento circular. En la figura 3-39(b), si hay conflictos, el dispositivo 2 vence al

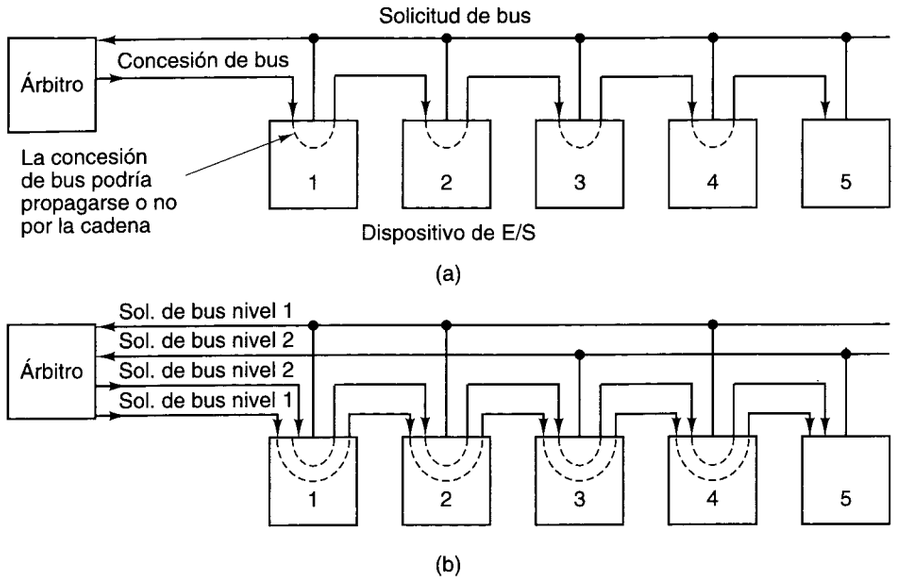


Figura 3-39. (a) Árbitro de bus centralizado de un nivel con encadenamiento circular. (b) El mismo árbitro, pero con dos niveles.

dispositivo 4, el cual vence al 3. El dispositivo 5 tiene la prioridad más baja porque está al final de la cadena de más baja prioridad.

Por cierto, no es técnicamente necesario conectar la línea de otorgamiento de bus del nivel 2 en serie a los dispositivos 1 y 2, porque no pueden hacer solicitudes en ese nivel. No obstante, por comodidad de implementación, se conectan todas las líneas de otorgamiento a todos los dispositivos, en lugar de conectar un cableado especial que dependa de cuál dispositivo tiene cuál prioridad.

Algunos árbitros tienen una tercera línea que un dispositivo habilita cuando aceptó una concesión y se apoderó del bus. Tan pronto como el dispositivo habilita esta línea de acuse, las líneas de solicitud y de otorgamiento pueden deshabilitarse. El resultado es que otros dispositivos pueden solicitar el bus mientras el primer dispositivo está usando el bus. Para cuando termina la transferencia actual en el bus, ya se ha seleccionado el siguiente controlador de bus, el cual puede iniciar tan pronto como se deshabilita la línea de acuse, momento en el cual se puede iniciar la siguiente ronda de arbitraje. Este esquema requiere una línea de bus extra y más lógica en cada dispositivo, pero aprovecha mejor los ciclos de bus.

En sistemas en los que la memoria está en el bus principal, la CPU debe competir con todos los dispositivos de E/S por el bus en casi cada ciclo. Una solución común para esta situación es dar a la CPU la prioridad más baja, de modo que reciba el bus sólo cuando nadie más lo quiere. La idea aquí es que la CPU siempre puede esperar, pero los dispositivos de E/S con frecuencia deben adquirir el bus rápidamente o perder datos de entrada. Los discos que giran a alta velocidad no pueden esperar. Este problema se evita en muchas computadoras

modernas colocando la memoria en un bus distinto del de los dispositivos de E/S, para que éstos no tengan que competir por el acceso al bus.

Otra posibilidad es el arbitraje de bus descentralizado. Por ejemplo, una computadora podría tener 16 líneas de solicitud de bus por prioridades. Cuando un dispositivo quiere usar el bus, habilita su línea de solicitud. Todos los dispositivos monitorean todas las líneas de solicitud, así que al término de cada ciclo de bus cada dispositivo sabe si fue el solicitante con más alta prioridad y por tanto si tiene permiso de usar el bus durante el siguiente ciclo. En comparación con el arbitraje de bus centralizado, este método de arbitraje requiere más líneas de bus pero evita los posibles costos del árbitro; además, limita el número de dispositivos, que no puede ser mayor que el número de líneas de solicitud.

Otro tipo de arbitraje de bus descentralizado, que se muestra en la figura 3-40, sólo usa tres líneas, por más dispositivos que estén presentes. La primera línea de bus es una de OR alambrado para solicitar el bus. La segunda se llama BUSY (ocupado) y es asertada por el amo de bus vigente. La tercera línea sirve para arbitrar el bus, y se conecta en serie a todos los dispositivos. La cabeza de esta cadena se mantiene habilitada conectándola a la fuente de alimentación de 5 volts.

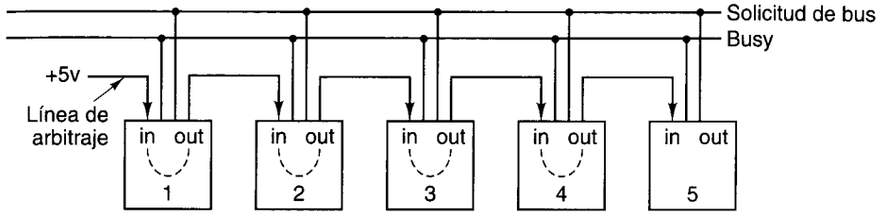


Figura 3-40. Arbitraje de bus descentralizado.

Si ningún dispositivo quiere el bus, la línea de arbitraje asertada se propaga por todos los dispositivos. Si un dispositivo quiere adquirir el bus, primero ve si el bus está inactivo y la señal de arbitraje que está recibiendo, IN, está asertada. Si IN está invalidada, el dispositivo no podrá convertirse en controlador del bus e invalidará OUT. En cambio, si IN está habilitada, el dispositivo deshabilitará OUT. Esto hará que su vecino cadena abajo vea IN deshabilitado y deshabilite su OUT. Así, todos los dispositivos cadena abajo ven IN deshabilitado y por tanto deshabilitan OUT. Al final, sólo un dispositivo tendrá IN habilitado y OUT deshabilitado. Este dispositivo se convertirá en el controlador del bus, habilitará BUSY y OUT, e iniciará su transferencia.

Si lo pensamos, veremos que el dispositivo más a la izquierda que quiera el bus lo recibirá. Así, este esquema es similar al arbitraje de cadena circular original, excepto que no tiene árbitro, por lo que es más económico, más rápido, y no es vulnerable a un fallo del árbitro.

3.4.6 Operaciones de bus

Hasta ahora, sólo hemos visto ciclos de bus ordinarios, en los que un controlador (generalmente la CPU) lee desde un esclavo (generalmente la memoria) o escribe en él. De hecho, existen varios tipos de ciclos de bus. A continuación veremos algunos de ellos.

Normalmente, sólo se transfiere una palabra a la vez. Sin embargo, cuando se usa caché, es deseable traer toda una línea de caché (por ejemplo, 16 palabras consecutivas de 32 bits) a la vez. En muchos casos es posible efectuar transferencias de bloques con mayor eficiencia que transferencias individuales sucesivas. Cuando se inicia una lectura de bloque, el controlador del bus le dice al esclavo cuántas palabras debe transferir, por ejemplo, colocando la cuenta de palabras en las líneas de datos durante T_1. En lugar de devolver sólo una palabra, el esclavo produce una palabra durante cada ciclo hasta agotar la cuenta. La figura 3-41 muestra una versión modificada de la figura 3-37(a), pero ahora con una señal adicional, $\overline{\text{BLOCK}}$, que se habilita para indicar que se solicita una transferencia de bloque. En este ejemplo, la lectura de un bloque de 4 palabras tarda 6 ciclos en lugar de 12.

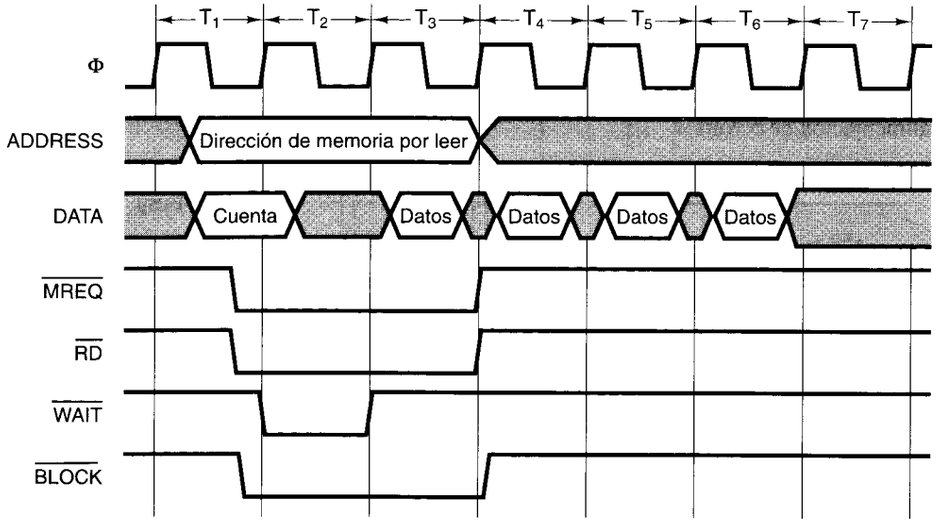


Figura 3-41. Transferencia de un bloque.

También existen otras clases de ciclos de bus. Por ejemplo, en un sistema multiprocesador que tiene dos o más CPU en el mismo bus, a menudo es necesario asegurarse de que sólo una CPU a la vez utilice alguna estructura de datos crítica en la memoria. Una forma común de lograr esto es tener una variable en la memoria que sea 0 cuando ninguna CPU esté usando la estructura de datos y 1 cuando esté en uso. Si una CPU quiere accesar a la estructura de datos, primero deberá leer la variable y si es 0, ponerla en 1. El problema es que, con un poco de mala suerte, dos CPU podrían leerla en ciclos de bus consecutivos. Si los dos ven que la variable es 0, los dos la pondrán en 1 y cada una creerá que es la única CPU que está usando la estructura de datos. Esta serie de sucesos lleva al caos.

Para evitar esta situación, los sistemas multiprocesador a menudo tienen un ciclo de bus especial de leer-modificar-escribir que permite a cualquier CPU leer una palabra de la memoria, inspeccionarla y modificarla, y escribirla de vuelta en la memoria, todo sin desocupar el

bus. Este tipo de ciclo evita que las CPU competidoras puedan usar el bus y así interfieran la operación de la primera CPU.

Otro tipo importante de ciclo de bus sirve para manejar interrupciones. Cuando la CPU ordena a un dispositivo de E/S hacer algo, generalmente espera una interrupción una vez que se concluye el trabajo. La señalización de la interrupción requiere el bus.

Puesto que varios dispositivos podrían querer causar una interrupción simultáneamente, surgen los mismos tipos de problemas de arbitraje que teníamos con los ciclos de bus ordinarios. La solución usual es asignar prioridades a los dispositivos, y usar un árbitro centralizado para dar prioridad a los dispositivos para los que el tiempo es más crítico. Existen chips controladores de interrupciones estándar y se usan ampliamente. La IBM PC y todas sus sucesoras usan el chip Intel 8259A, que se ilustra en la figura 3-42.

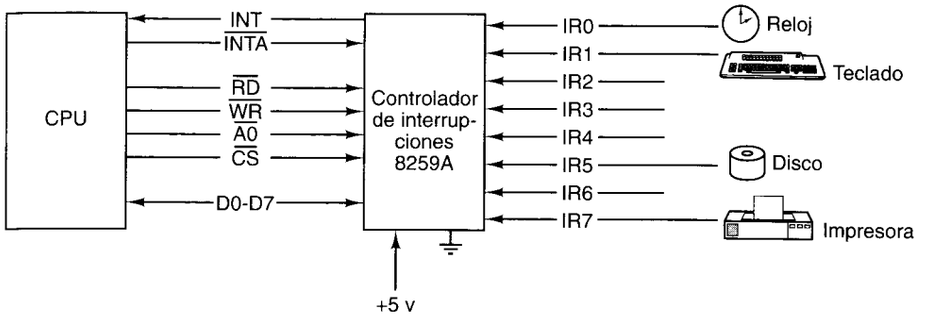


Figura 3-42. Uso del controlador de interrupciones 8259A.

Hasta ocho chips controladores de E/S se pueden conectar directamente a las ocho entradas IRx (solicitud de interrupción, *Interrupt Request*) del 8259A. Cuando cualquiera de estos dispositivos desea causar una interrupción, habilita su línea de entrada. Cuando se habilita una o más entradas, el 8259A habilita INT (INTerrupción), que alimenta directamente la terminal de interrupción de la CPU. Cuando la CPU está en condiciones de atender la interrupción, devuelve un pulso al 8259A por INTA (acuse de interrupción, *INTerrupt Acknowledge*). En ese momento el 8259A deberá especificar cuál entrada causó la interrupción colocando el número de esa entrada en el bus de datos. Esta operación requiere un ciclo de bus especial. Luego, el hardware de la CPU utiliza ese número como índice de una tabla de apuntadores, llamados **vectores de interrupción**, para encontrar la dirección del procedimiento que debe ejecutar para atender la interrupción.

El 8259A contiene varios registros que la CPU puede leer y escribir empleando ciclos de bus ordinarios y las líneas $\overline{\text{RD}}$, $\overline{\text{WR}}$ (escribir, *WRite*), $\overline{\text{CS}}$ y $\overline{\text{A0}}$. Una vez que el software ha manejado la interrupción y está listo para recibir la siguiente, escribe un código especial en uno de los registros, y esto hace que el 8259A deshabilite INT, a menos que tenga otra interrupción pendiente. Estos registros también pueden modificarse para colocar el 8259A en uno de varios modos, enmascarar un conjunto de interrupciones y habilitar otras funciones.

Cuando están presentes más de ocho dispositivos de E/S, los 8259A pueden conectarse en cascada. En el caso más extremo, las ocho entradas pueden conectarse a las salidas de

ocho 8259A adicionales, lo que permite tener hasta 64 dispositivos de E/S en una red de interrupciones de dos etapas. El 8259A tiene unas cuantas líneas dedicadas a manejar esta conexión en cascada, que hemos omitido para simplificar.

Si bien estamos muy lejos de agotar el tema del diseño de buses, el material anterior deberá bastar para entender los fundamentos del funcionamiento de los buses y la interacción entre ellos y las CPU. Pasemos ahora de lo general a lo específico y examinemos algunos ejemplos de CPU reales y sus buses.

3.5 EJEMPLOS DE CHIPS DE CPU

En esta sección examinaremos los chips Pentium II, UltraSPARC II y picoJava II con cierto detalle en el nivel de hardware.

3.5.1 El Pentium II

El Pentium II es un descendiente directo de la CPU 8088 que se usó en la IBM PC original. Aunque el Pentium II con sus 7.5 millones de transistores está muy lejos del 8088 con 29,000 transistores, es totalmente compatible con el 8088 y puede ejecutar programas binarios para el 8088 sin modificación (además de programas para todos los procesadores intermedios).

Desde el punto de vista del software, el Pentium II es una máquina de 32 bits completa. Tiene la misma ISA en el nivel de usuario que los chips 80386, 80486, Pentium y Pentium Pro, incluidos los mismos registros, las mismas instrucciones y una implementación completa en el chip del estándar de punto flotante IEEE 754.

Desde el punto de vista del hardware, el Pentium II es algo más porque puede direccionar 64 GB de memoria física y puede transferir datos de y a la memoria en unidades de 64 bits. Aunque el programador no puede observar estas transferencias de 64 bits, hacen que la máquina sea más rápida que una máquina de 32 bits pura.

Internamente, en el nivel de microarquitectura, el Pentium II es básicamente un Pentium Pro al que se añadieron las instrucciones MMX. Las instrucciones en el nivel de ISA se obtienen de la memoria con mucha anticipación y se descomponen en microoperaciones tipo RISC. Estas microoperaciones se almacenan en un buffer, y tan pronto como una dispone de los recursos necesarios para ejecutarse, puede iniciarse. Se pueden iniciar varias microoperaciones en el mismo ciclo, lo que convierte al Pentium II en una máquina superescalar.

El Pentium II tiene una caché de dos niveles. Hay un par de cachés en el chip, 16 KB para instrucciones y 16 KB para datos, además de una caché unificada de segundo nivel de 512 KB. El tamaño de la línea de caché es de 32 bytes. La caché de segundo nivel se ejecuta a la mitad de la frecuencia de reloj de la CPU. Hay relojes de CPU de 233 MHz o más.

Se usan dos buses externos primarios en los sistemas Pentium II, ambos sincrónicos. El bus de memoria sirve para accesar la DRAM principal; el bus PCI se usa para comunicarse con los dispositivos de E/S. A veces se conecta un bus de **legado** (o sea, antiguo) al bus de PCI para poder conectar a él dispositivos periféricos de tecnología anterior.

Un sistema Pentium II puede tener uno o dos CPU que comparten una memoria común. En un sistema de dos CPU, existe el peligro de que si una palabra se lee y se coloca en una caché, y se modifica ahí sin que se escriba de vuelta en la memoria, si la otra CPU trata de leer la palabra obtendrá un valor incorrecto. Se incluye apoyo especial (espionaje) para evitar este problema.

Una diferencia importante entre el Pentium II y todos sus predecesores es su empaquetado. Desde la 8088 hasta el Pentium Pro, todas las CPU de Intel eran chips normales, con terminales en los lados o la base que se podían conectar en zócalos. En contraste, el Pentium II está provisto de lo que Intel llama un **SEC** (**cartucho de una sola arista**, *Single Edge Cartridge*). Como puede verse en la figura 3-43, un SEC es un encapsulado de plástico relativamente grande que contiene la CPU, la caché nivel 2 y un conector de arista para exportar las señales. El SEC de Pentium II tiene 242 conectores.

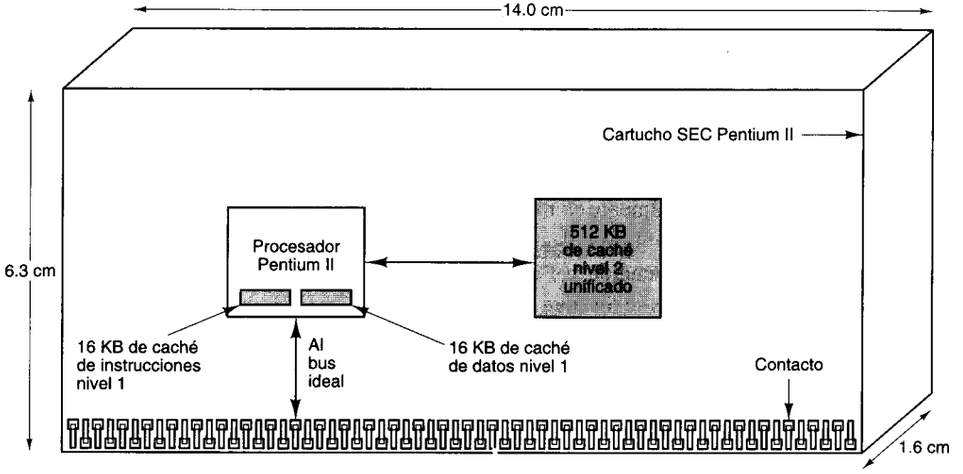


Figura 3-43. El paquete SEC del Pentium II.

Aunque sin duda Intel tuvo sus motivos para cambiar a este modelo de encapsulado, produjo un problema en una dimensión que jamás previó. Al parecer, muchos clientes tienen la costumbre de abrir sus computadoras para buscar el chip de CPU. En los primeros Pentium II que salieron a la venta, los clientes no podían encontrar la CPU y se quejaron a grandes voces ("¡Mi computadora no tiene CPU!"). Intel resolvió este problema pegando una imagen del chip de CPU (en realidad, un holograma) en la parte delantera de todos los SEC que se distribuyeron posteriormente.

El control de energía es un aspecto importante del Pentium II. La cantidad de calor que se desprende depende de la frecuencia de reloj, pero es del orden de 30-50 watts. Esta cantidad es enorme cuando se trata de un chip de computadora. Para tener una idea de cómo se sienten 50 watts, acerque su mano a una bombilla de 50 watts (sin tocarla) que ha estado encendida durante un rato. Por consiguiente, el SEC está configurado para aceptar un

radiador de calor, que es necesario para disipar el calor generado. Esto implica que cuando un Pentium II haya dejado de ser útil como CPU siempre podrá usarse como estufa para acampar.

Según las leyes de la física, cualquier cosa que emita gran cantidad de calor deberá consumir mucha energía. En una computadora portátil en la que la carga de la batería es limitada no es deseable consumir mucha energía. Para subsanar este problema, Intel ha hecho posible poner a la CPU a dormir cuando está ociosa y en una narcolepsia profunda cuando parece que va a estar ociosa durante mucho tiempo. Cuando la CPU está en el sueño profundo, los valores de la caché y los registros se conservan, pero el reloj y todas las unidades internas se desactivan. No se sabe si un Pentium II sueña cuando está profundamente dormido.

La conexión lógica de terminales del Pentium II

Los 242 conectores de arista del SEC se usan para 170 señales, 27 conexiones de potencia (a diferentes voltajes), 35 tierras y 10 repuestos para uso futuro. Algunas de las señales lógicas emplean dos o más terminales (como la dirección de memoria solicitada), así que sólo hay 53 señales distintas. En la figura 3-44 se muestra la configuración de conexión lógica de terminales (*pinout*) un tanto simplificada. En el lado izquierdo de la figura están los seis grupos principales de señales del bus de memoria; en el lado derecho están señales de diversos tipos. Los nombres que están totalmente en mayúsculas son los nombres reales dados por Intel a las señales. Los demás son nombres colectivos de varias señales relacionadas.

Intel emplea una convención para asignar nombres que es importante entender. Dado que todos los chips actuales se diseñan con computadoras, es necesario poder representar los nombres de las señales como texto ASCII. Es demasiado difícil usar testas para indicar las señales que se habilitan en nivel bajo, por lo que en vez de ello Intel coloca el símbolo # después del nombre. Así, $\overline{\text{BPRI}}$ se expresa como BPRI#. Como puede verse en la figura, casi todas las señales de Pentium II se habilitan en nivel bajo.

Examinemos las señales, comenzando con las de bus. El primer grupo de señales sirve para solicitar el bus (es decir, efectuar arbitraje). BPRI# permite a un dispositivo hacer una solicitud de alta prioridad, que tiene precedencia respecto a una normal. LOCK# permite a una CPU bloquear el bus para evitar que la otra lo use antes de que la primera termine.

Una vez que una CPU u otro controlador de bus se ha apropiado del bus, puede presentar una solicitud utilizando el siguiente grupo de señales. Las direcciones son de 36 bits, pero los tres bits de orden bajo siempre deben ser 0 y por tanto no se les asignan líneas. Por tanto, A# sólo tiene 33 líneas. Todas las transferencias son de 8 bytes, alineadas a una frontera de 8 bytes. Con 36 bits de dirección, el máximo de memoria direccionable es 2^{36}, que es 64 GB.

Cuando una dirección se coloca en el bus, la señal ADS# se habilita para indicar al objetivo (o sea, la memoria) que las líneas de dirección son válidas. El tipo de ciclo de bus (por ejemplo, leer una palabra o escribir un bloque) se coloca en las líneas REQ#. Dos de las señales de paridad protegen a A# y una protege a ADS# y REQ#. Las cinco líneas de error son utilizadas por el esclavo para informar errores de paridad y por todos los demás dispositivos para informar de otros errores.

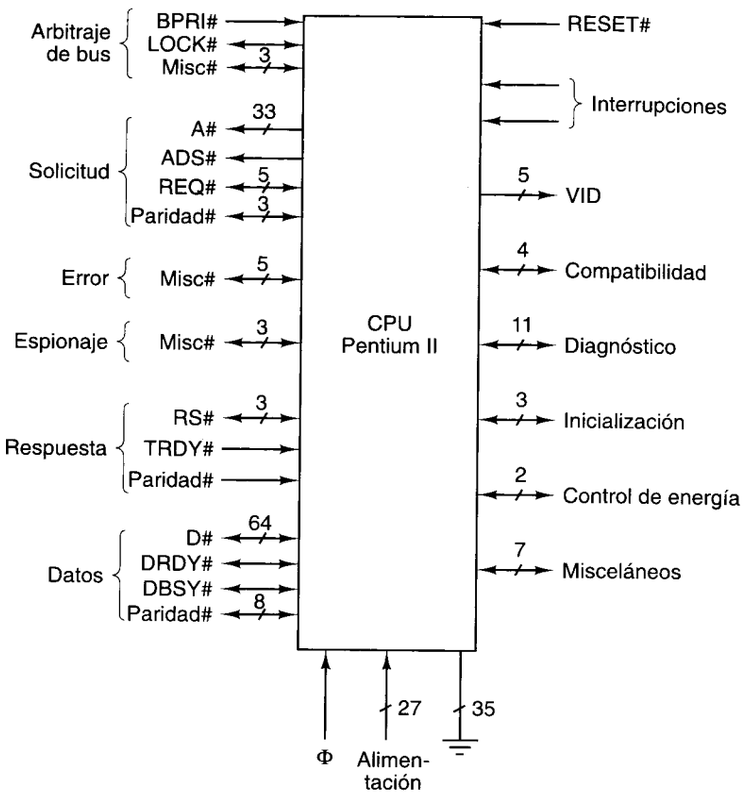


Figura 3-44. Conexión lógica de terminales del Pentium II. Los nombres en mayúsculas son los nombres oficiales de Intel para señales individuales. Los nombres en mayúsculas y minúsculas son grupos de señales relacionadas o descripciones de señales.

El grupo de Espionaje se usa en sistemas multiprocesador para que una CPU pueda averiguar si una palabra que necesita está en la caché de la otra CPU. Describiremos la forma de espiar del Pentium II en el capítulo 8.

El grupo Respuesta contiene señales que el esclavo utiliza para enviar informes al amo (controlador). RS# contiene el código de situación. TRDY# indica que el esclavo (el objetivo) está listo para aceptar datos del amo. Estas señales también tienen verificación de paridad.

El último grupo del bus sirve para transferir realmente los datos. D# sirve para colocar 8 bytes de datos en el bus. Una vez que están ahí, se aserta DRDY# para anunciar su presencia. DSBY# sirve para decirle al mundo que el bus está ocupado.

RESET# sirve para restablecer la CPU en caso de ocurrir una calamidad. El Pentium II puede configurarse de modo que use las interrupciones igual que el 8088 (para fines de compatibilidad con lo existente) pero también puede usar un nuevo sistema de interrupciones con un dispositivo llamado **APIC** (**controlador de interrupciones programable avanzado**, *Advanced Programmable Interrupt Controller*).

El Pentium II puede operar a varios voltajes, pero necesita saber cuál. Las señales VID sirven para seleccionar automáticamente el voltaje de la fuente de potencia. Las señales de Compatibilidad sirven para engañar a dispositivos antiguos en el bus que piensan que están hablando con un 8088. El grupo de Diagnóstico contiene señales para probar y depurar sistemas según la norma de prueba IEEE 1149.1 JTAG. El grupo de Inicialización se ocupa de arrancar el sistema (*booting*). El grupo de Control de energía permite poner a dormir a la CPU normal y profundamente. Por último, el grupo Misceláneo es una revoltura de señales que incluye una que la CPU habilita si su temperatura interna llega a 130°C (266°F). Si una CPU alcanza alguna vez esta temperatura, probablemente está soñando en retirarse y convertirse en una estufa para acampar.

Filas de procesamiento en el bus de memoria del Pentium II

Las CPU modernas como el Pentium II son mucho más rápidas que las memorias DRAM modernas. Para evitar que la CPU muera de inanición por falta de datos, es indispensable obtener un rendimiento máximo de la memoria. Por esta razón, el bus de memoria del Pentium II usa las filas de procesamiento, y se pueden estar ejecutando hasta ocho transacciones de bus al mismo tiempo. Ya vimos el concepto de filas de procesamiento en el capítulo 2 en el contexto de una CPU con filas de procesamiento (vea la figura 2-4), pero las memorias también pueden usarlos.

Para ello, las solicitudes de memoria del Pentium II, llamadas **transacciones**, tienen seis etapas:

1. La fase de arbitraje de bus.
2. La fase de solicitud.
3. La fase de informar errores.
4. La fase de espionaje.
5. La fase de respuesta
6. La fase de datos.

No todas las fases se necesitan en todas las transacciones. La fase de arbitraje de bus determina cuál de los posibles controladores de bus toma su turno. La fase de solicitud permite colocar la dirección en el bus y hacer la solicitud. La fase de informar errores permite al esclavo anunciar que la dirección tuvo un error de paridad o que algo más anda mal. La fase de espionaje permite a una CPU espiar a la otra, algo que sólo se necesita en un sistema multiprocesador. La fase de respuesta es aquella donde el controlador se entera de si está a punto de obtener los datos que quiere o no. Por último, la fase de datos permite devolver los datos.

El secreto del bus de memoria con conductos del Pentium II es que cada fase utiliza diferentes señales de bus, de modo que cada una es totalmente independiente de las demás. Los seis grupos de señales requeridas son los que se muestran en la figura 3-44 a la izquierda.

Por ejemplo, una CPU puede tratar de obtener el bus utilizando las señales de arbitraje. Una vez que adquirió el derecho a ser el siguiente, libera estas líneas del bus y comienza a usar las líneas del grupo de Solicitud. Mientras tanto, la otra CPU o algún dispositivo de E/S puede ingresar en la fase de arbitraje del bus, y así sucesivamente. La figura 3-45 muestra cómo varias transacciones de bus pueden estar pendientes al mismo tiempo.

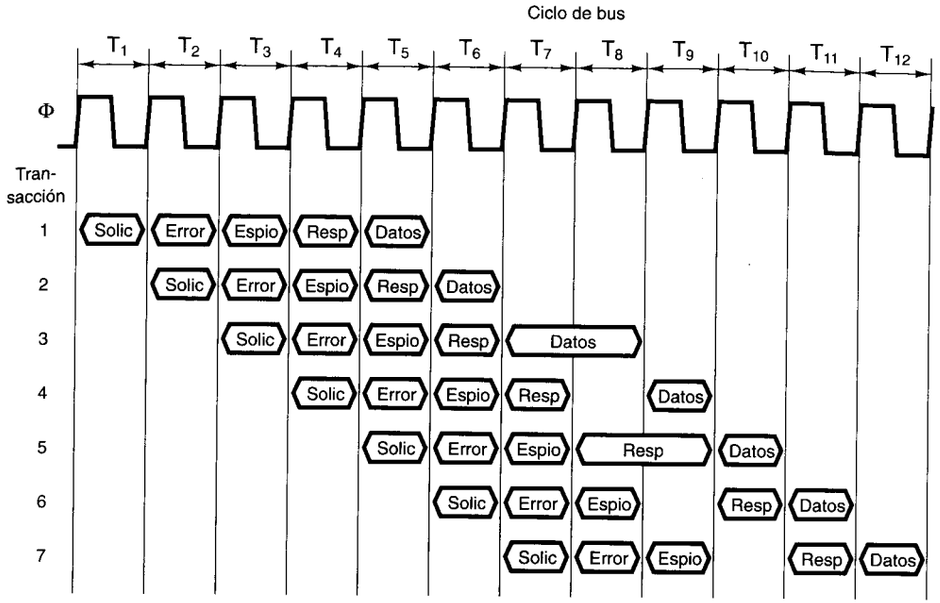


Figura 3-45. Colocación de solicitudes en filas de procesamiento en el bus de memoria del Pentium II.

En la figura 3-45 no se muestra la fase de arbitraje de bus porque no siempre se necesita. Por ejemplo, si el dueño actual del bus (por lo regular la CPU) quiere ejecutar otra transacción, no tiene que readquirir el bus; sólo tiene que pedir el bus otra vez después de ceder la propiedad del bus a otro dispositivo solicitante. Las transacciones 1 y 2 son sencillas: cinco fases en cinco ciclos de bus. La transacción 3 introduce una fase de datos más larga, por ejemplo, porque es una transferencia de bloque o porque la memoria direccionada insertó un estado de espera. Por consiguiente, la transacción 4 no puede iniciar su fase de datos cuando quisiera hacerlo. La transacción 4 observa que la señal DBSY# todavía está habilitada y simplemente espera que la deshabiliten. En la transacción 5 vemos que la fase de respuesta también puede tardar varios ciclos de bus, lo que retarda la transacción 6. Por último, en la transacción 7 observamos que una vez que se ha introducido una burbuja en la fila de procesamiento permanece ahí si se siguen iniciando consecutivamente nuevas transacciones. Sin embargo, en la práctica es poco probable que la CPU intente iniciar una nueva transacción en cada ciclo de bus, así que las burbujas no duran tanto.

3.5.2 El UltraSPARC II

Como segundo ejemplo de chip de CPU, examinaremos la familia UltraSPARC de Sun. Esta familia es la línea de las CPU SPARC de 64 bits de Sun. La familia UltraSPARC se ajusta cabalmente a la arquitectura SPARC versión 9, que también es para las CPU de 64 bits, y se usa en estaciones de trabajo y servidores Sun, así como en varias aplicaciones más. Esta familia incluye el UltraSPARC I, el UltraSPARC II y el UltraSPARC III, cuya arquitectura es muy similar; difieren principalmente en la fecha de introducción y la rapidez del reloj. Para que la explicación que sigue sea más concreta, nos referiremos al UltraSPARC II, pero casi todo aplica también a los demás UltraSPARC.

El UltraSPARC II es una máquina RISC tradicional y es plenamente compatible en lo binario con la arquitectura SPARC V8 de 32 bits. Puede ejecutar programas binarios para SPARC V8 de 32 bits sin modificación porque la arquitectura SPARC V9 es compatible con la arquitectura SPARC V8. El único aspecto en que el UltraSPARC II se desvía de la arquitectura SPARC V9 es en la adición del conjunto de instrucciones para multimedia VIS, que se diseñó para aplicaciones gráficas, decodificación MPEG en tiempo real, etcétera.

El UltraSPARC II se diseñó para construir multiprocesadores de cuatro nodos con memoria compartida sin la adición de circuitos externos, y multiprocesadores más grandes con un mínimo de circuitos externos. En otras palabras, gran parte del "adhesivo" que se necesita para construir un multiprocesador ya está incluido en cada chip UltraSPARC II.

A diferencia del SEC del Pentium II, la CPU UltraSPARC II es un chip autónomo, aunque moderadamente grande, con 5.4 millones de transistores. Tiene 787 terminales en su base, dispuestas como se muestra en la figura 3-46. Esta gran cantidad de terminales se explica en parte por el uso de 64 bits para las direcciones y 128 bits para los datos, pero también se debe en parte a la forma como funcionan las cachés. Además, muchas de las terminales no se usan o son redundantes. Se escogió el número 787 para poder usar un encapsulado estándar en la industria. La industria probablemente considera que es de buena suerte tener un número primo de terminales.

El UltraSPARC II tiene dos cachés internas: 16 KB para instrucciones y 16 KB para datos. Al igual que el Pentium II, también usa una caché de nivel 2 externo al chip, pero a diferencia del Pentium II esa caché no viene empaquetada con el UltraSPARC II en un cartucho patentado. Los diseñadores de sistemas están en libertad de escoger cualquier chip de caché comercial que deseen para la caché de nivel 2.

La decisión de integrar la caché nivel 2 en el Pentium II y separarlo en el UltraSPARC II se debe en parte a las opciones técnicas y en parte a los diferentes modelos comerciales que Intel y Sun usan. En el aspecto técnico, una caché externa es más flexible (las cachés del UltraSPARC II pueden ir de 512 KB a 16 MB; las del Pentium II están fijas en 512 KB), pero podría ser más lento a causa de su mayor distancia de la CPU. Además, se requieren más señales visibles para direccionar la caché (los 242 conectores del cartucho SEC del Pentium II no incluyen señales para la caché, ya que la interacción CPU-caché es interna al cartucho), pero las 787 terminales del UltraSPARC II sí incluyen control de caché.

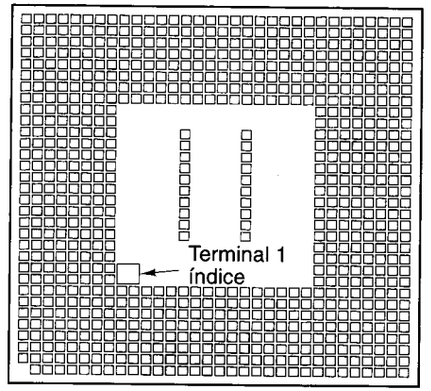


Figura 3-46. El chip de CPU UltraSPARC II.

En el aspecto comercial, Intel es un fabricante de semiconductores y puede diseñar y fabricar su propio chip de caché de nivel 2 y conectarlo a la CPU con una interfaz patentada de alto desempeño. Sun, en cambio, es un fabricante de computadoras, no de chips. Sí diseña algunos de sus chips (como los UltraSPARC), pero subcontrata fabricantes de semiconductores como Texas Instruments y Fujitsu. Cuando puede, Sun prefiere usar chips comerciales que han sido depurados por el mercado tan competitivo. Las SRAM que se usan en las cachés de nivel 2 se pueden conseguir de muchos proveedores de chips, por lo que Sun no sintió una necesidad especial de diseñar su propia SRAM. Esta decisión implica hacer la caché de nivel 2 independiente del chip de la CPU.

Casi todas las estaciones de trabajo Sun tienen un bus asincrónico de 25 MHz llamado **SBus**. Los dispositivos de E/S se pueden conectar al SBus, pero éste es demasiado lento para la memoria, así que Sun creó un mecanismo diferente para que una o más CPU UltraSPARC se comuniquen con una o más memorias: la **UPA** (**arquitectura de ultrapuerto**, *Ultra Port Architecture*). La UPA se puede implementar como un bus, un conmutador, o una combinación de las dos cosas. Diferentes modelos de estaciones de trabajo y servidores usan diferentes implementaciones de la UPA. A la CPU no le importa qué implementación de la UPA se usa porque la interfaz con la UPA está perfectamente definida, y es esta interfaz la que el chip de la CPU debe reconocer (y reconoce).

En la figura 3-47 vemos el corazón de un sistema UltraSPARC II, que muestra el chip de CPU, la interfaz UPA y la caché de nivel 2 (dos SRAM comerciales). La figura también incluye un chip **UDB II** (**UltraSPARC Data Buffer II**), cuya función se explicará más adelante. Cuando la CPU necesita una palabra de la memoria, primero la busca en una de sus cachés internas (nivel 1). Si la encuentra, continúa su ejecución a toda velocidad; si no encuentra la palabra en la caché de nivel 1, prueba en la de nivel 2.

Aunque analizaremos el uso de cachés con detalle en el capítulo 4, conviene hablar un poco al respecto aquí. Toda la memoria principal está dividida en líneas de caché (bloques) de

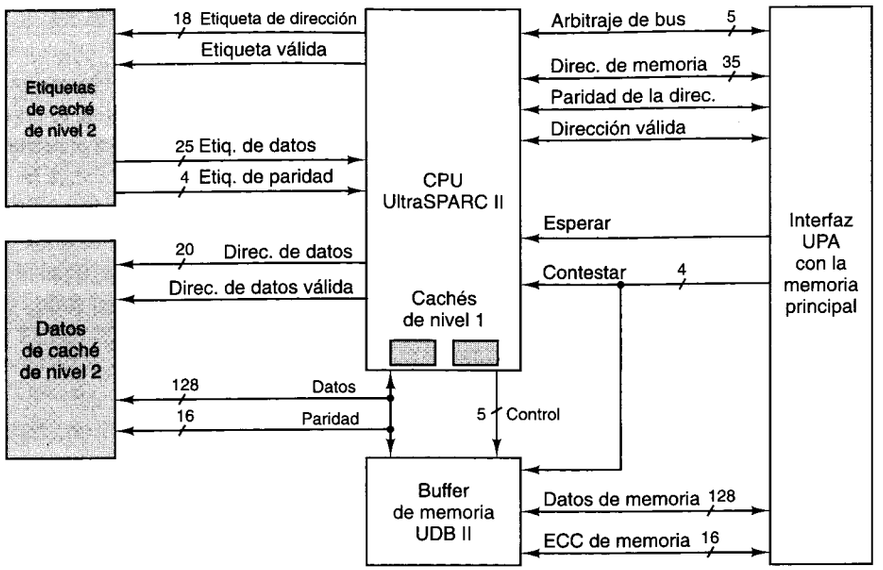


Figura 3-47. Principales características del corazón de un sistema UltraSPARC II.

64 bytes. Las 256 líneas de instrucciones que más se usan y las 256 líneas de datos que más se usan están en la caché nivel 1. Las líneas de caché que se usan mucho pero que no caben en la caché de nivel 1 se guardan en la caché de nivel 2. Éste contiene líneas de datos y de instrucciones mezcladas aleatoriamente, y se almacenan en el rectángulo rotulado "Datos de caché de nivel 2". El sistema tiene que saber cuáles líneas están en la caché de nivel 2. Esta información se guarda en una segunda SRAM, rotulada "Etiquetas de caché de nivel 2".

Cuando no se encuentra una línea en la caché de nivel 1, la CPU envía su identificador (dirección de etiqueta) a la caché de nivel 2. La respuesta (datos de etiqueta) proporciona la información que permite a la CPU saber si la línea está o no en la caché de nivel 2 y, si está, en qué estado está. Si la línea está guardada ahí, la CPU la trae. Las transferencias de datos tienen una anchura de 16 bytes, por lo que se necesitan cuatro ciclos para traer toda una línea a la caché de nivel 1.

Si la línea de caché no está en la caché de nivel 2, deberá traerse de la memoria principal vía la interfaz UPA. La UPA del UltraSPARC II se implementa con un controlador centralizado. Las señales de dirección y control de la CPU (de todas las CPU si hay más de una) llegan ahí. Para accesar la memoria, la CPU debe usar primero las líneas de arbitraje de bus para obtener permiso de continuar. Una vez concedido el permiso, la CPU envía la dirección de memoria, especifica el tipo de solicitud, y habilita la línea de dirección válida. (Estas líneas son bidireccionales porque otras CPU de un multiprocesador UltraSPARC II necesitan tener acceso a las cachés remotas para mantener la coherencia de todas las cachés.) La dirección y el tipo de ciclo de bus se colocan en las líneas de Dirección en dos ciclos: el renglón en el primer ciclo y la columna en el segundo, como vimos en la figura 3-31.

Mientras espera los resultados, la CPU bien podría continuar con otros trabajos. Por ejemplo, una caché no inhibe la ejecución de una o más instrucciones que ya se obtuvieron mientras se preobtiene una instrucción , las cuales podrían hacer referencia a datos que no están en ninguna caché. Así, podría haber varias transacciones con la UPA pendientes en un momento dado. La UPA puede manejar dos flujos de transacción independientes (por lo regular lecturas y escrituras), cada uno con múltiples transacciones pendientes. Corresponde al controlador centralizado seguir la pista a todo esto y efectuar las solicitudes de memoria propiamente dichas en el orden más eficiente.

Cuando por fin llegan los datos de la memoria, podrían venir en grupos de 8 bytes, y con un código de corrección de errores de 16 bits para mayor confiabilidad. Una transacción podría pedir todo un bloque de caché, una palabra cuádruple (8 bytes) o incluso menos bytes. Todos los datos que llegan se colocan en buffers en el UDB. El propósito del UDB es desacoplar aun más la CPU del sistema de memoria, de modo que puedan operar asincrónicamente. Por ejemplo, si la CPU tiene que escribir una palabra o línea de caché en la memoria, en lugar de esperar para accesarla UPA puede escribir los datos en el UDB de inmediato y dejar que el UDB se encargue de hacerlos llegar a la memoria posteriormente. El UDB también genera y verifica el código de corrección de errores. Cabe señalar que la descripción del UltraSPARC II que acabamos de presentar, al igual que la del Pentium II que la precedió, se ha simplificado considerablemente, aunque se describió la esencia del funcionamiento.

3.5.3 El picoJava II

Tanto el Pentium II como el UltraSPARC II son ejemplos de CPU de alto desempeño diseñadas para construir PC y estaciones de trabajo extremadamente rápidas. Cuando la gente piensa en las computadoras, éste es el tipo de sistemas en el que tienden a concentrarse. Sin embargo, existe todo un mundo de computadoras distinto que en realidad es mucho más grande: los sistemas incorporados. En esta sección daremos un vistazo a ese mundo.

Es probable que no exageremos mucho si decimos que cualquier dispositivo eléctrico que cueste más de 100 dólares probablemente incluye una computadora. Sin duda los televisores, teléfonos celulares, organizadores personales electrónicos, hornos de microondas, cámaras de video, videograbadoras, impresoras láser, alarmas contra robo, aditamentos para la sordera, juegos electrónicos y otros dispositivos están controlados por computadoras. Las computadoras contenidas en estos artículos suelen estar optimadas para tener un precio bajo más que para tener un buen desempeño, lo que da lugar a concesiones diferentes de las que se hacen en las CPU de alta tecnología que hemos estado estudiando hasta ahora.

Tradicionalmente, los procesadores incorporados se han programado en lenguaje ensamblador, pero a medida que los aparatos se vuelven más complicados y las consecuencias de los errores de software se hacen más severos, otros enfoques se han vuelto competitivos. En particular, el uso de Java como lenguaje de programación para sistemas incorporados es atractivo en virtud de su relativa facilidad de programación, el tamaño reducido del código y su independencia de la plataforma. La principal desventaja de usar Java en aplicaciones incorporadas es la necesidad de tener un intérprete grande en software que ejecute el código JVM producido por el compilador de Java, y la lentitud del proceso de interpretación.

Sun y otras compañías han atacado este problema diseñando y construyendo chips de CPU que tienen JVM como su conjunto de instrucciones nativo. Este enfoque combina las ventajas de usar Java como lenguaje de programación, la transportabilidad y el tamaño compacto del código JVM binario producido por el compilador de Java, y la rapidez de ejecución por hardware especializado. En esta sección examinaremos una arquitectura de CPU moderna basada en Java diseñada específicamente para el mercado de los sistemas incorporados.

La CPU es la picoJava II de Sun en que se basa el chip microJava 701 de Sun, aunque Sun también ha otorgado licencias para el diseño a otras compañías. Es una CPU de un solo chip con dos interfaces de bus, una para el bus de memoria que tiene una anchura de 64 bits y una para el bus PCI que tiene una anchura de 32 bits, como se muestra en la figura 3-48. Al igual que el Pentium II y el UltraSPARC II, el picoJava II tiene una caché nivel 1 (opcional) dividida en el chip, con hasta 16 KB para instrucciones y hasta 16 KB para datos. Sin embargo, a diferencia de esas dos CPU, el picoJava II no tiene una caché nivel 2 porque uno de los parámetros de diseño fundamentales de los sistemas incorporados es el bajo costo. A continuación describiremos la implementación del picoJava II hecha por Sun: el microJava 701. El chip es pequeño según los criterios actuales: sólo dos millones de transistores para el núcleo más otro millón y medio para las dos cachés de 16 KB juntas.

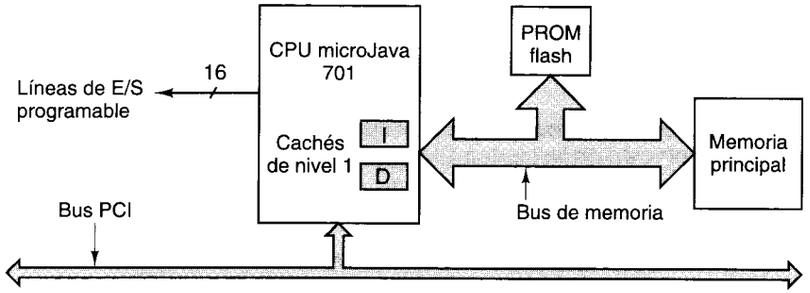


Figura 3-48. Un sistema microJava 701.

Tres características de la figura 3-48 saltan a la vista. Primera, el microJava 701 utiliza el bus PCI (a 33 MHz o bien 66 MHz). Este bus fue creado por Intel para usarse en los sistemas Pentium del extremo superior, pero es independiente del procesador. La ventaja de usar el bus PCI es que es un estándar, y así se evita tener que diseñar un bus nuevo. Además, ya existen muchas tarjetas hechas para insertarse en él. Aunque la existencia de tarjetas para PCI no es gran ventaja cuando se están construyendo teléfonos celulares, en el caso de televisores de Web y otros aparatos más grandes sí es un punto en favor.

Segunda, un sistema microJava 701 normalmente incluye una PROM flash. La cuestión aquí es que en el caso de un aparato una buena parte del programa, si no es que todo, tiene que estar incluido ya en el dispositivo. Una PROM flash es un buen lugar para almacenar el programa, así que es útil tener una interfaz para una de ellas. Otro chip que se puede añadir al sistema (no se muestra) contiene las interfaces de E/S serial y paralela que se encuentran en una PC.

Tercera, el microJava 701 tiene 16 líneas de E/S programable que pueden conectarse a botones, interruptores y lámparas del aparato. Por ejemplo, un horno de microondas suele tener un teclado numérico y algunos otros botones que podrían conectarse directamente al chip de CPU. Tener líneas de E/S programable directamente en la CPU hace innecesario incluir chips controladores de E/S programables, así que el diseño es más simple y menos costoso. El chip también tiene tres temporizadores programables incluidos, lo cual también es útil porque los aparatos a menudo operan en tiempo real.

El microJava 701 se vende en un encapsulado **BGA** (**Ball Grid Array**) de 316 terminales que es un estándar de la industria. De estas terminales, 59 están conectadas al bus PCI. Estudiaremos este bus en una sección posterior del capítulo. Otras 123 terminales son para el bus de memoria, incluidas 64 terminales de datos bidireccionales así como terminales de dirección aparte. Otras terminales se usan para control (7), temporizadores (3), interrupciones (11), pruebas (10) y E/S programable (16). Algunas de las terminales restantes se usan (con redundancia) para la alimentación y la tierra, pero otras no se usan. Otros fabricantes del picoJava II están en libertad de escoger un bus distinto, otro paquete, etcétera.

El chip también tiene varias características de hardware más, como un modo inactivo para ahorrar energía de baterías, un controlador de interrupciones en el chip y apoyo pleno para el estándar de prueba IEEE 1149.1 JTAG.

3.6 EJEMPLOS DE BUSES

Los buses son el tejido conectivo de los sistemas de computación. En esta sección examinaremos de cerca algunos buses muy utilizados: el bus ISA, el bus PCI y el Bus Serial Universal. El bus ISA es una expansión del bus original de la IBM PC. Por razones de compatibilidad hacia atrás, se sigue incluyendo en todas las PC basadas en Intel, aunque todas estas máquinas tienen también un segundo bus, más rápido: el PCI. El bus PCI es más ancho que el ISA y opera con una tasa de reloj más alta. El Bus Serial Universal es un bus de E/S cada vez más popular para periféricos de baja velocidad, como los ratones y los teclados. En las secciones que siguen examinaremos cada uno de estos buses.

3.6.1 El bus ISA

El bus de la IBM PC era de hecho el estándar en los sistemas basados en el 8088 porque casi todos los fabricantes de clones de la PC lo copiaron para poder usar en sus sistemas las muchas tarjetas de E/S existentes hechas por otros fabricantes. Ese bus tenía 62 líneas de señal, incluidas 20 para una dirección de memoria, 8 para datos y una en cada caso para habilitar lectura de memoria, escritura de memoria, lectura de E/S y escritura de E/S. También había señales para solicitar y conceder interrupciones y usar DMA, y nada más. Era un bus muy sencillo.

Físicamente, el bus estaba grabado en la tarjeta madre de la PC, con una media docena de conectores situados a intervalos de 2 cm para insertar en ellos tarjetas (*plug-ins*). Cada tarjeta

tenía una pestaña que embonaba en el conector. La pestaña tenía 31 tiras chapeadas en oro en cada lado que hacían contacto eléctrico con el conector.

Cuando IBM introdujo la PC/AT basada en el 80286, enfrentó un problema grave. Si hubiera comenzado desde cero y diseñado un bus de 16 bits totalmente nuevo, muchos clientes en potencia habrían dudado en comprar la máquina porque ninguna de las numerosas tarjetas para PC insertables hechas por terceros fabricantes habría funcionado con la nueva máquina. Por otra parte, seguir con el bus de PC y sus 20 líneas de dirección y 8 líneas de datos habría impedido aprovechar la capacidad del 80286 para direccionar 16M de memoria y transferir palabras de 16 bits.

La solución que se escogió fue extender el bus de PC. Las tarjetas insertables para PC tienen un conector de arista con 62 contactos, pero que no corre a todo lo largo de la tarjeta. La solución para la PC/AT fue colocar un segundo conector de arista en la parte inferior de la tarjeta, junto al principal, y diseñar los circuitos de la AT de modo que funcionaran con ambos tipos de tarjetas. la idea general se ilustra en la figura 3-49.

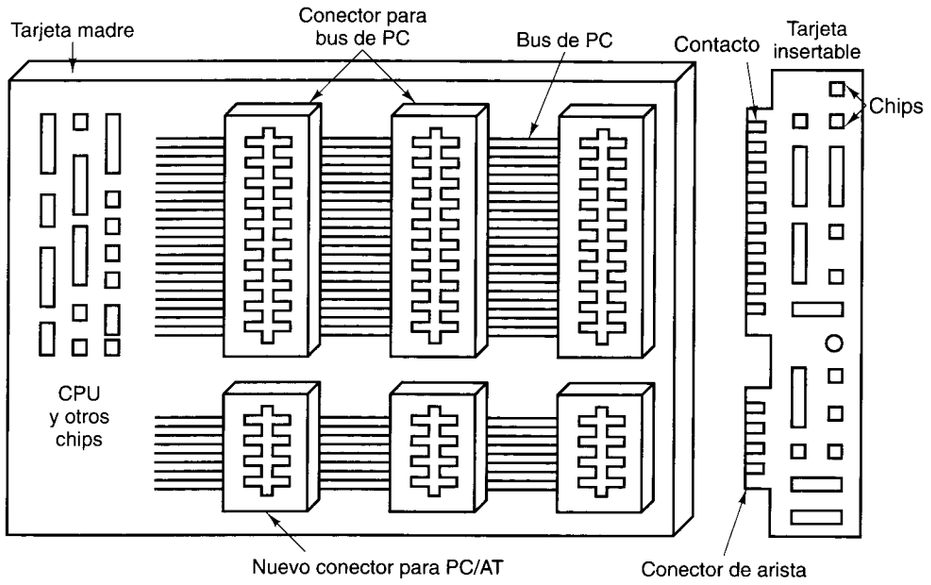


Figura 3-49. El bus PC/AT tiene dos componentes, la parte de la PC original y la parte nueva.

El segundo conector del bus PC/AT contiene 36 líneas. De éstas, 31 se dedican a más líneas de dirección, más líneas de datos, más líneas de interrupciones y más canales de DMA, además de alimentación y tierra. El resto se ocupa de diferencias entre las transferencias de 8 bits y las de 16 bits.

Cuando IBM introdujo la serie PS/2 como sucesora de la PC y la PC/AT, decidió que era hora de comenzar de nuevo. En parte, esta decisión pudo haber sido técnica (el bus de PC a

esas alturas ya era realmente obsoleto), pero en parte se debió sin duda a un deseo de poner un obstáculo en el camino de las compañías que fabricaban clones de la PC, las cuales se habían apoderado de una porción preocupante del mercado. Así, las máquinas PS/2 de precio medio y alto se equiparon con un bus, el Microchannel, que era totalmente nuevo y que estaba protegido por una pared de patentes respaldada por un ejército de abogados.

El resto de la industria de las computadoras personales reaccionó adoptando su propio estándar, el bus **ISA** (**arquitectura estándar de la industria**, *Industry Standard Architecture*), que es básicamente el bus de la PC/AT a 8.33 MHz. La ventaja de este enfoque es que mantiene la compatibilidad con las máquinas y las tarjetas existentes; además, se basa en un bus para el que IBM había otorgado generosamente licencias a muchas compañías con el fin de asegurar que el mayor número de fabricantes posible produjera tarjetas para el PC original, algo que resultó contraproducente para IBM. Todas las PC basadas en Intel tienen aún este bus, aunque generalmente acompañado por uno o más buses distintos. Se puede encontrar una descripción exhaustiva del bus ISA en (Shanley y Anderson, 1995a).

Más adelante, el bus ISA se extendió a 32 bits con unas cuantas funciones adicionales (por ejemplo, para multiprocesamiento). Este nuevo bus se llamó **EISA** (**ISA extendido**). Sin embargo, casi no se han producido tarjetas para él.

3.6.2 El bus PCI

En la IBM PC original, casi todas las aplicaciones se basaban en texto. Gradualmente, con la introducción de Windows, comenzaron a usarse las interfaces gráficas con el usuario. Ninguna de estas aplicaciones forzaba mucho al bus ISA, pero al pasar el tiempo cada vez más aplicaciones, sobre todo los juegos de multimedia, comenzaron a usar computadoras para exhibir video de pantalla completa y pleno movimiento, y la situación cambió radicalmente.

Hagamos un cálculo sencillo. Consideremos una pantalla de 1024×768 que se usa para imágenes en movimiento de color natural (3 bytes/pixel). Una pantalla contiene 2.25 MB de datos. Para que el movimiento sea continuo, se requieren al menos 30 pantallas por segundo, lo que corresponde a una tasa de datos de 67.5 MB/s. De hecho, la situación es peor, porque para exhibir un video desde un disco duro, CD-ROM o DVD los datos deben pasar por el bus de la unidad de disco a la memoria. Luego, para exhibirlos, los datos deben pasar otra vez por el bus hacia el adaptador de gráficos. Así, pues, necesitamos un ancho de banda de bus de 135 MB/s sólo para el video, sin contar el ancho de banda que necesitan la CPU y otros dispositivos.

El bus ISA opera con una rapidez máxima de 8.33 MHz y puede transferir dos bytes por ciclo, lo que da un ancho de banda máximo de 16.7 MB/s. El bus EISA puede transferir cuatro bytes por ciclo, así que alcanza los 33.3 MB/s. Es evidente que ninguno de ellos se acerca siquiera a lo que se necesita para video de pantalla completa.

En 1990 Intel previó esto y diseñó un nuevo bus con un acho de banda mucho mayor que incluso el del bus EISA, y lo llamó **bus PCI** (**interconexión de componentes periféricos**, *Peripheral Component Interconnect*). Para fomentar su uso, Intel patentó el bus PCI y luego puso todas las patentes en el dominio público para que cualquier compañía pudiera construir periféricos para él sin tener que pagar regalías. Además, Intel formó un consorcio de la industria, el PCI Special Interest Group, para controlar el futuro del bus PCI. Como resultado de

estas acciones, el bus PCI se ha popularizado enormemente. Casi todas las computadoras basadas en Intel a partir de la Pentium tienen un bus PCI, y muchas otras computadoras también lo tienen. Incluso Sun tiene una versión del UltraSPARC que usa el bus PCI, el UltraSPARC IIi. El bus PCI se cubre con lujo de detalle en (Shanley y Anderson 1995b; y Solari y Willse, 1998).

El bus de PCI original transfería 32 bits por ciclo y operaba a 33 MHz (tiempo de ciclo de 30 ns), para un ancho de banda total de 133 MB/s. En 1993 se introdujo PCI 2.0, y en 1995 salió PCI 2.1. El PCI 2.2 cuenta con funciones para computadoras móviles (principalmente para ahorrar la energía de las baterías). El bus PCI opera hasta 66 MHz y puede manejar transferencias de 64 bits, para un ancho de banda total de 528 MB/s. Con este tipo de capacidad es posible tener video de pantalla completa y pleno movimiento (suponiendo que el disco y el resto del sistema están a la altura del reto). En todo caso, el bus PCI no será el cuello de botella.

Aunque 528 MB/s parece muy rápido, aún tiene dos problemas. Primero, no es lo bastante bueno como para ser bus de memoria. Segundo, no es compatible con todas las viejas tarjetas para ISA que todavía existen. La solución que ideó Intel fue diseñar computadoras con tres o más buses, como se muestra en la figura 3-50. Aquí vemos que la CPU puede comunicarse con la memoria principal por un bus de memoria especial, y que se puede conectar un bus ISA al bus PCI. Esta organización satisface todos los requisitos, así que prácticamente todas las computadoras Pentium II emplean esta arquitectura.

Dos componentes clave de esta arquitectura son los dos chips de puente (que Intel fabrica: de ahí su interés en todo este proyecto). El puente PCI conecta la CPU, la memoria y el bus PCI. El puente ISA conecta el bus PCI al bus ISA y también apoya uno o dos discos IDE. Casi todos los sistemas Pentium II vienen con una o más ranuras PCI libres para añadir nuevos periféricos de alta velocidad, y una o más ranuras ISA para agregar periféricos de baja velocidad.

La gran ventaja del diseño de la figura 3-50 es que la CPU tiene un ancho de banda a la memoria gigantesco. Al usar un bus de memoria patentado, el bus PCI tiene un ancho de banda muy grande para periféricos rápidos como discos SCSI, adaptadores de gráficos, etc., y es posible seguir usando tarjetas ISA viejas. El cuadro USB de la figura se refiere al Bus Serial Universal, que veremos más adelante en este capítulo.

Aunque ilustramos un sistema con un bus PCI y un bus ISA, es posible tener varios ejemplares de cada uno. Existen chips de puente PCI a PCI que conectan dos buses PCI, de modo que los sistemas más grandes pueden tener dos o más buses PCI independientes. También es posible tener dos o más chips de puente PCI a ISA en un sistema, y así incluir varios buses ISA.

Habría sido útil que sólo existiera un tipo de tarjeta para PCI. Lamentablemente, no es así. Se ofrecen opciones de voltaje, capacidad y temporización. Muchas computadoras viejas usan 5 volts, mientras que las más nuevas tienden a usar 3.3 volts, por lo que el bus PCI apoya ambos voltajes. Los conectores son iguales excepto por dos trozos de plástico que impiden que la gente inserte una tarjeta de 5 volts en un bus PCI de 3.3 volts o viceversa. Por fortuna existen tarjetas universales que manejan ambos voltajes y pueden insertarse en cualquier tipo de ranura. Además de la opción de voltaje, hay tarjetas en versiones de 32 bits y de 64 bits.

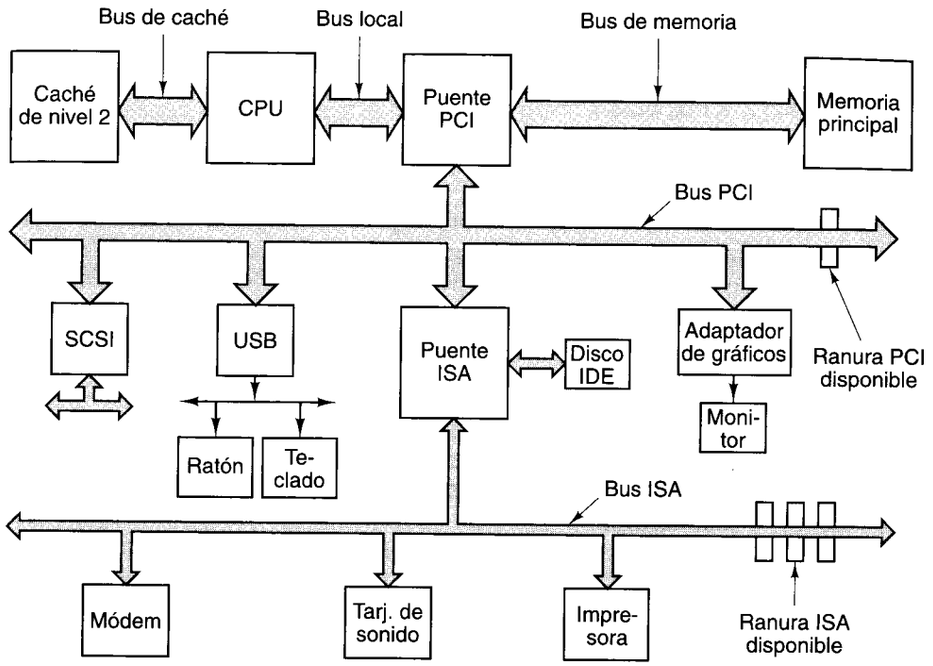


Figura 3-50. Arquitectura de un sistema Pentium II representativo. Los buses más gruesos tienen más ancho de banda que los más delgados.

Las tarjetas de 32 bits tienen 120 terminales; las de 64 bits poseen las mismas 120 terminales más 64 adicionales, semejante a la manera en que se extendió a 16 bits el bus de la IBM PC (vea la figura 3-49). Un sistema de bus PCI que acepta tarjetas de 64 bits también puede aceptar tarjetas de 32 bits, pero no al revés. Por último, los buses y las tarjetas PCI pueden operar a 33 MHz o 66 MHz. La decisión se toma conectando permanentemente una terminal ya sea a la fuente de alimentación o a tierra. Los conectores son idénticos para ambas velocidades.

El bus PCI es sincrónico, como lo fueron todos los buses de PC hasta el de la IBM PC original. Todas las transacciones se efectúan entre un amo, llamado oficialmente **iniciador**, y un esclavo, llamado oficialmente **objetivo**. Para reducir el número de contactos del bus PCI, las líneas de direcciones y de datos se multiplexan. Así, sólo se necesitan 64 terminales en las tarjetas PCI para las señales de dirección y de datos, aunque PCI maneja direcciones de 64 bits y datos de 64 bits.

Las líneas de dirección y datos multiplexadas funcionan como sigue. En una operación de lectura, durante el ciclo 1, el iniciador coloca la dirección en el bus. En el ciclo 2 el iniciador quita la dirección y el bus se invierte para que el esclavo pueda usarlo. En el ciclo 3, el esclavo envía los datos solicitados. En las operaciones de escritura, el bus no tiene que invertirse porque el iniciador coloca en él tanto la dirección como los datos. No obstante, la

transacción mínima sigue siendo de tres ciclos. Si el esclavo no puede responder en tres ciclos, puede insertar estados de espera. También se permiten transferencias de bloques de tamaño ilimitado, así como varios otros tipos de ciclos de bus.

Arbitraje del bus PCI

Para usar el bus PCI un dispositivo primero debe adquirirlo. El arbitraje del bus PCI emplea un sistema centralizado que se muestra en la figura 3-51. En casi todos los diseños, el árbitro de bus viene incorporado en uno de los chips de puente. Todo dispositivo PCI tiene dos líneas dedicadas que van de él al árbitro. Una línea, REQ#, sirve para solicitar el bus. La otra, GNT#, es útil para recibir concesiones de bus.

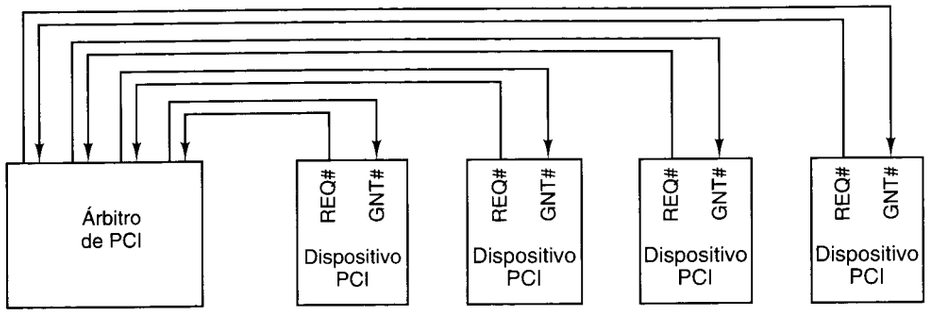


Figura 3-51. El bus PCI emplea un árbitro de bus centralizado.

Para solicitar el bus, un dispositivo PCI (incluida la CPU) aserta REQ# y espera hasta detectar que el árbitro habilitó su línea GNT#. Cuando eso sucede, el dispositivo puede usar el bus en el siguiente ciclo. El algoritmo empleado por el árbitro no está definido en la especificación de PCI. Se permiten arbitrajes por turno circular, por prioridad y otros esquemas. Es evidente que un buen árbitro es justo y no deja que algunos dispositivos esperen en forma indefinida.

Una concesión de bus es válida para una transacción, aunque la longitud de dicha transacción en teoría es ilimitada. Si un dispositivo quiere ejecutar una segunda transacción y ningún otro dispositivo está solicitando el bus, podrá continuar, aunque normalmente es preciso insertar un ciclo ocioso entre las transacciones. Sin embargo, en circunstancias especiales y si no hay competencia por el bus, un dispositivo puede efectuar transacciones una tras otra sin tener que insertar un ciclo ocioso. Si un iniciador de bus está efectuando una transferencia muy larga y algún otro dispositivo solicitó el bus, el árbitro puede deshabilitar la línea GNT#. Se espera que el iniciador de bus en turno vigile la línea GNT#, así que cuando percibe la deshabilitación debe liberar el bus en el siguiente ciclo. Este esquema permite transferencias muy largas (que son eficientes) cuando sólo hay un candidato a iniciador de bus, pero también ofrece una respuesta rápida a los dispositivos competidores.

Señales del bus PCI

El bus PCI tiene varias señales obligatorias que se muestran en la figura 3-52(a), y varias señales opcionales, que se muestran en la figura 3-52(b). El resto de las 120 o 184 terminales se usa para alimentación, tierra y otras funciones diversas relacionadas; no se mostrarán aquí. Las columnas *Amo* (iniciador) y *Esclavo* (objetivo) indican quién habilita la señal en una transacción normal. Si un dispositivo distinto (digamos CLK) habilita la señal, las dos columnas se dejan en blanco.

Examinemos brevemente cada una de las señales del bus PCI. Comenzaremos con las señales obligatorias (de 32 bits); luego pasaremos a las señales opcionales (de 64 bits). La señal CLK controla el bus; casi todas las demás señales son sincrónicas con ella. En contraste con el bus ISA, una transacción de bus PCI se inicia en el flanco descendente de CLK, que está a la mitad del ciclo, no al principio.

Las 32 señales AD son para la dirección y los datos (en transacciones de 32 bits). En general, durante el ciclo 1 se habilita la dirección y durante el ciclo 3 se habilitan los datos. La señal PAR es un bit de paridad para AD. La señal C/BE# se usa para dos cosas distintas: en el ciclo 1, contiene el comando de bus (leer una palabra, leer un bloque, etc.); en el ciclo 2 contiene un mapa de bits de 4 bits que indica cuáles bytes de la palabra de 32 bits son válidos. Con la ayuda de C/BE# es posible leer o escribir cualesquier 1, 2 o 3 bytes, así como una palabra entera.

El iniciador de bus habilita la señal FRAME# para iniciar una transacción de bus. Esta señal le dice al esclavo que la dirección y los comandos de bus ya son válidos. En una lectura casi siempre se habilita IRDY# al mismo tiempo que FRAME#. Esa señal indica que el iniciador está listo para aceptar los datos. En una escritura, IRDY# se habilita después, cuando los datos ya están en el bus.

La señal IDSEL tiene que ver con el hecho de que todo dispositivo PCI debe tener un espacio de configuración de 256 bytes que otros dispositivos pueden leer (habilitando IDSEL). Este espacio de configuración contiene propiedades del dispositivo. La característica de "conectar y operar" (*Plug 'n Play*) de algunos sistemas operativos emplea el espacio de configuración para averiguar cuáles dispositivos están en el bus.

Ahora llegamos a las señales que el esclavo habilita. La primera, DEVSEL#, anuncia que el esclavo detectó su dirección en las líneas AD y está listo para intervenir en la transacción. Si DEVSEL# no se habilita dentro de cierto límite de tiempo, el iniciador supone que el dispositivo direccionado está ausente o bien descompuesto.

La segunda señal de esclavo es TRDY#, la cual habilita en las lecturas para anunciar que los datos están en las líneas AD, y en las escrituras para anunciar que está preparado para aceptar datos.

Las siguientes tres señales son para informar sobre los errores. La primera es STOP#, que el esclavo habilita si ocurre algún desastre y quiere abortar la transacción en curso. La siguiente, PERR#, sirve para informar de un error de paridad de datos en el ciclo anterior. En el caso de una lectura el iniciador la habilita; en una escritura el esclavo lo hace. Corresponde al receptor tomar las medidas apropiadas. Por último, SERR# sirve para informar sobre errores de dirección y errores de sistema.

Señal	Líneas	Amo	Esclavo	Descripción
CLK	1			Reloj (33 MHz o 66 MHz)
AD	32	×	×	Líneas de dirección y datos multiplexados
PAR	1	×		Bit de paridad de dirección o datos
C/BE	4	×		Comando de bus/mapa de bits para bytes habilitados
FRAME#	1	×		Indica que AD y C/BE están asertadas
IRDY#	1	×		Lectura: el amo aceptará; escritura: datos presentes
IDSEL	1	×		Seleccionar espacio de configuración en lugar de memoria
DEVSEL#	1		×	El esclavo decodificó su dirección y está escuchando
TRDY#	1		×	Lectura: datos presentes; escritura, el esclavo aceptará
STOP#	1		×	El esclavo quiere parar la transacción de inmediato
PERR#	1			El receptor detectó error de paridad de datos
SERR#	1			Se detectó error de paridad de dirección o error de sistema
REQ#	1			Arbitraje de bus: solicitud para tener el bus
GNT#	1			Arbitraje de bus: conceder el bus
RST#	1			Restablecer el sistema y todos los dispositivos

(a)

Señal	Líneas	Amo	Esclavo	Descripción
REQ64#	1	×		Solicitud para ejecutar una transacción de 64 bits
ACK64#	1		×	Se da permiso para una transacción de 64 bits
AD	32	×		32 bits de dirección o datos adicionales
PAR64	1	×		Paridad para los 32 bits de dirección/datos extra
C/BE#	4	×		4 bits adicionales para habilitar bytes
LOCK	1	×		Bloquear el bus para efectuar múltiples transacciones
SBO#	1			Acierto en caché remoto (en multiprocesador)
SDONE	1			Espionaje efectuado (en multiprocesador)
INTx	4			Solicitar interrupción
JTAG	5			Señales de prueba IEEE 1149.1 JTAG
M66EN	1			Conectado a potencia o tierra (66 MHz o 33 MHz)

(b)

Figura 3-52. (a) Señales obligatorias del bus PCI. (b) Señales opcionales del bus PCI.

Las señales REQ# y GNT# sirven para efectuar arbitraje de bus. Éstas no las habilita el iniciador de bus vigente, sino un dispositivo que quiere convertirse en iniciador de bus. La última señal obligatoria es RST#, que sirve para restablecer el sistema, sea porque el usuario oprimió el botón RESET o porque algún dispositivo del sistema detectó un error fatal. La habilitación de esta señal restablece todos los dispositivos y activa nuevamente la computadora.

Ahora llegamos a las señales opcionales, que en su mayoría se relacionan con la expansión de 32 a 64 bits. Las señales REQ64# y ACK64# permiten al iniciador pedir permiso para realizar una transacción de 64 bits, y al esclavo, aceptar. Las señales AD, PAR64 y C/BE# no son más que extensiones de las señales de 32 bits correspondientes.

Las siguientes tres señales no tienen que ver con el número de bits, sino con los sistemas mutiprocesador, algo que las tarjetas PCI no están obligadas a apoyar. La señal LOCK permite bloquear el bus para efectuar múltiples transacciones. Las dos señales siguientes tienen que ver con el espionaje de bus para mantener la coherencia de las cachés.

Las señales INTx son para solicitar interrupciones. Una tarjeta PCI puede contener hasta cuatro dispositivos lógicos individuales, y cada uno puede tener su propia línea para solicitar interrupciones. Las señales JTAG son para el procedimiento de prueba IEEE 1149.1 JTAG. Por último, la señal M66EN se pone permanentemente alta o baja, para fijar la velocidad del reloj, y no debe cambiar durante la operación del sistema.

Transacciones en el bus PCI

El bus PCI es en realidad muy sencillo (en comparación con otros buses). Para entenderlo mejor, considere el diagrama de temporización de la figura 3-53. Aquí vemos una transacción de lectura, seguida de un ciclo ocioso y una transacción de escritura por el mismo iniciador de bus.

Cuando el flanco descendente del reloj ocurre durante T_1, el iniciador coloca la dirección de memoria en AD y el comando de bus en C/BE#; luego habilita FRAME# para iniciar la transacción de bus.

Durante T_2, el iniciador deja que flote el bus de dirección a fin de que pueda invertirse como preparación a ser alimentado por el esclavo durante T_3. El iniciador también modifica C/BE# para indicar cuáles bytes de la palabra direccionada quiere habilitar (es decir, leer).

En T_3, el esclavo aserta DEVSEL# para que el iniciador sepa que recibió la dirección y planee responder; además, coloca los datos en las líneas AD y habilita TRDY# para avisarle al iniciador que lo hizo. Si el esclavo no puede responder con tanta rapidez, de todas maneras habilita DEVSEL# para anunciar su presencia pero mantiene TRDY# deshabilitado hasta que puede colocar los datos en el bus. Este procedimiento introduciría uno o más estados de espera.

En este ejemplo (y con frecuencia en la realidad), el siguiente ciclo es ocioso. A partir de T_5 vemos al mismo amo iniciar una escritura. Lo primero que hace es colocar la dirección y el comando en el bus, como siempre, sólo que ahora habilita los datos durante el segundo ciclo. Puesto que el mismo dispositivo está alimentando las líneas AD, no hay necesidad de un ciclo de inversión. En T_7 la memoria acepta los datos.

3.6.3 El bus serial universal

El bus PCI está bien para conectar periféricos de alta velocidad a una computadora, pero es demasiado costoso tener una interfaz PCI para cada dispositivo de E/S de baja velocidad, como un teclado o un ratón. Históricamente, todos los dispositivos de E/S estándar se conectaban a la computadora de una forma especial, dejando algunas ranuras ISA y PCI libres para

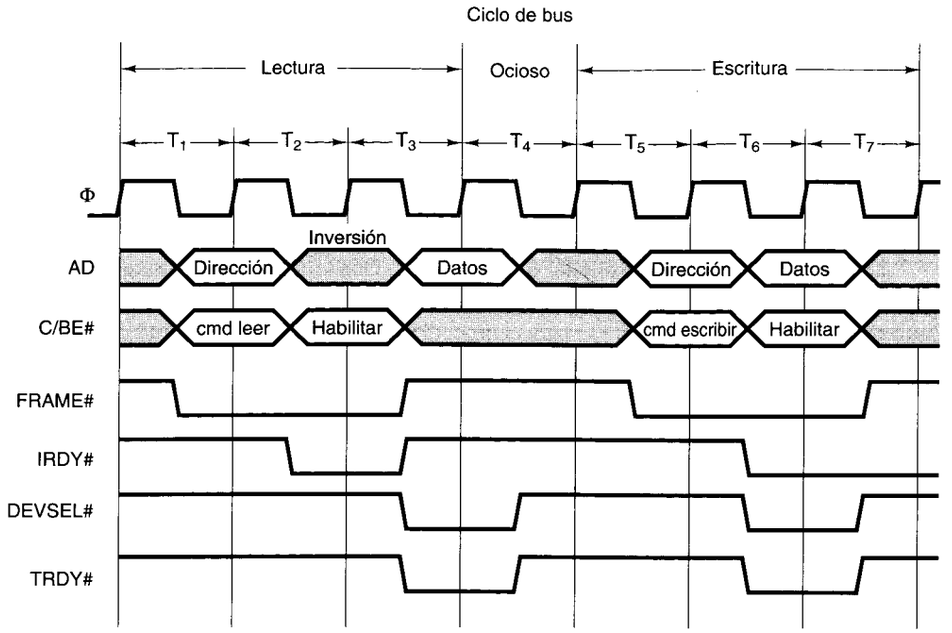


Figura 3-53. Ejemplos de transacciones de bus PCI de 32 bits. Los primeros tres ciclos se emplean para una operación de lectura, luego un ciclo ocioso y luego tres ciclos para una operación de escritura.

añadir nuevos dispositivos. Lamentablemente, este esquema tuvo muchos problemas desde el principio. Por ejemplo, cada dispositivo de E/S nuevo por lo regular viene con su propia tarjeta ISA o PCI. El usuario a menudo tiene la responsabilidad de ajustar interruptores y puenteadores en la tarjeta y asegurarse de que los ajustes no estén en conflicto con otras tarjetas. Luego, el usuario debe abrir el gabinete de la computadora, insertar la tarjeta con mucho cuidado, cerrar el gabinete y rearrancar la computadora. Para muchos usuarios este proceso es difícil y hay una posibilidad alta de cometer errores. Además, el número de ranuras ISA y PCI es muy limitado (dos o tres, por lo regular). Las tarjetas Plug 'n Play eliminan los ajustes de puenteadores, pero de todos modos el usuario tiene que abrir la computadora para insertar la tarjeta, y el número de ranuras sigue siendo limitado.

Para resolver este problema, a mediados de la década de los noventa se reunieron representantes de siete compañías (Compaq, DEC, IBM, Intel, Microsoft, NEC y Northern Telecom) para diseñar una mejor manera de conectar dispositivos de E/S de baja velocidad a una computadora. Desde entonces, cientos de compañías más se les han unido. El estándar resultante se llama **USB (Bus Serial Universal)** y se le está implementando ampliamente en las computadoras personales. Este bus se describe con detalle en (Anderson, 1997; y Tan, 1997).

Algunos de los objetivos de las compañías que originalmente concibieron el USB e iniciaron el proyecto son los siguientes:

1. Los usuarios no deben tener que ajustar interruptores ni puenteadores en las tarjetas o dispositivos.
2. Los usuarios no deben tener que abrir la caja para instalar dispositivos de E/S nuevos.
3. Sólo debe haber un tipo de cable, que sirva para conectar todos los dispositivos.
4. Los dispositivos de E/S deberán obtener su energía del cable.
5. Se deberán poder conectar hasta 127 dispositivos a una sola computadora.
6. El sistema deberá apoyar dispositivos de tiempo real (por ejemplo, sonido, teléfono).
7. Los dispositivos deberán poder instalarse mientras la computadora está funcionando.
8. No se deberá requerir un rearranque después de instalar un nuevo dispositivo.
9. El nuevo bus y sus dispositivos de E/S deberán ser de bajo costo.

El USB cumple con todos estos objetivos; se le diseñó para dispositivos de baja velocidad como teclados, ratones, cámaras de foto fija, digitalizadores de imágenes, teléfonos digitales, etc. El ancho de banda total del USB es de 1.5 MB/s, que es suficiente para una cantidad sustancial de tales dispositivos. Se escogió este límite bajo para mantener bajos los costos.

Un sistema USB consiste en un **eje raíz** que se conecta al bus principal (vea la figura 3-50). Este eje tiene zócalos para cables que se pueden conectar a dispositivos de E/S o a ejes de expansión, a fin de tener más zócalos, de modo que la topología de un sistema USB es un árbol con su raíz en el eje raíz, dentro de la computadora. Los cables tienen diferentes conectores en el extremo del eje y en el extremo del dispositivo, para evitar que las personas conecten accidentalmente un zócalo de eje a otro.

El cable consta de cuatro hilos: dos para datos, uno para alimentación (+ 5 volts) y uno para tierra. El sistema de señalización transmite un 0 como una transición de voltaje y un 1 como la ausencia de una transición de voltaje, de modo que las series largas de ceros generan una serie de pulsaciones regular.

Cuando se conecta un nuevo dispositivo de E/S, el eje raíz detecta este suceso e interrumpe el sistema operativo. Luego éste consulta al dispositivo para averiguar qué es y qué ancho de banda de USB necesita. Si el sistema operativo decide que hay suficiente ancho de banda para el dispositivo, asigna a éste una dirección única (1-127) y coloca esta dirección y otra información en los registros de configuración dentro del dispositivo. De este modo es posible añadir dispositivos nuevos sobre la marcha, sin que el usuario tenga que configurarlos y sin tener que instalar nuevas tarjetas ISA o PCI. Las tarjetas no inicializadas tienen la dirección 0, para poderlas direccionar. A fin de simplificar el cableado, muchos dispositivos USB contienen ejes integrados que aceptan dispositivos USB adicionales. Por ejemplo, un monitor podría tener dos zócalos de eje para dar cabida a los altavoces derecho e izquierdo.

Lógicamente, el sistema USB puede verse como un conjunto de conductos de bit que van del eje raíz a los dispositivos de E/S. Cada dispositivo puede dividir su conducto de bits

en, cuando más, 16 subconductos para diferentes tipos de datos (por ejemplo, audio y video). Dentro de cada conducto o subconducto, fluyen datos del eje raíz al dispositivo o en la otra dirección. No hay tráfico entre dos dispositivos de E/S.

Precisamente cada 1.00 ± 0.05 ms, el eje raíz transmite una nueva trama para mantener a todos los dispositivos sincronizados en el tiempo. Una trama se asocia a un conducto de bits y consiste en paquetes, el primero de los cuales va del eje raíz al dispositivo. Los paquetes subsecuentes de la trama también podrían enviarse en esta dirección, pero también podrían regresar del dispositivo al eje raíz. En la figura 3-54 se muestra una sucesión de cuatro tramas.

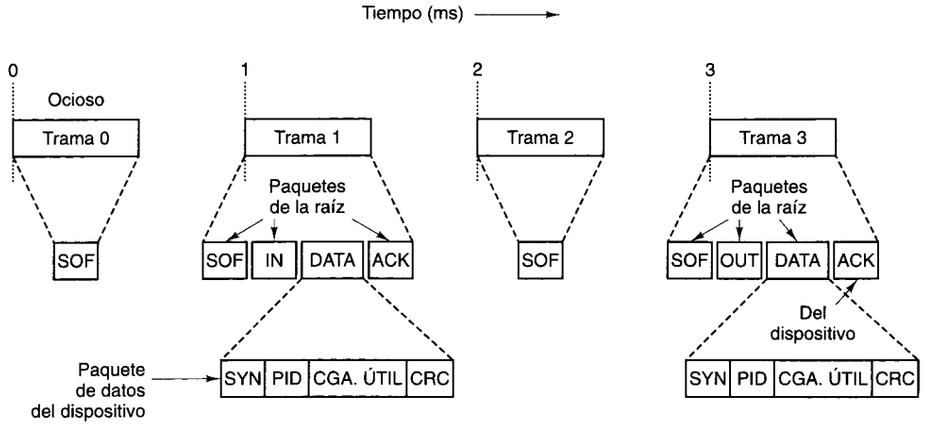


Figura 3-54. El eje raíz USB envía tramas cada 1.00 ms.

En la figura 3-54 no hay trabajo que realizar en las tramas 0 y 2, así que todo lo que se necesita es un paquete SOF (inicio de trama, *Start of Frame*). Este paquete siempre se difunde a todos los dispositivos. La trama 1 es un escrutinio, por ejemplo, una solicitud a un escáner para que devuelva los bits que ha encontrado en la imagen que está digitalizando. La trama 3 consiste en datos que se envían a algún dispositivo, digamos una impresora.

USB reconoce cuatro clases de tramas: de control, isocrónicas, de volumen y de interrupción. Las tramas de control sirven para configurar dispositivos, dar órdenes y preguntar por su estado. Las tramas isocrónicas son para dispositivos de tiempo real como micrófonos, altavoces y teléfonos que necesitan enviar o aceptar datos a intervalos de tiempo precisos. Estas tramas tienen un retraso altamente predecible pero no ofrecen retransmisión en caso de ocurrir errores. Las tramas de volumen son para transferencias grandes hacia o desde dispositivos que no requieren tiempo real, como las impresoras. Por último, las tramas de interrupción son necesarias porque USB no reconoce interrupciones. Por ejemplo, en lugar de hacer que un teclado cause una interrupción cada vez que se pulsa una tecla, el sistema operativo puede escrutarlo cada 50 ms para recolectar cualesquier digitaciones que estén pendientes.

Una trama consiste en uno o más paquetes, posiblemente algunos para cada dirección. Existen cuatro tipos de paquetes: de testigo, de datos, de saludo y especiales. Los paquetes de testigo van de la raíz a un dispositivo y sirven para control del sistema. Los paquetes SOF, IN y

OUT de la figura 3-54 son paquetes de testigo. El paquete SOF (inicio de trama) es el primero de cada trama y marca su principio. Si no hay trabajo que efectuar, el paquete SOF es el único de la trama. El paquete de testigo IN es un escrutinio, que pide al dispositivo devolver ciertos datos. Los campos del paquete IN indican cuál conducto de bits se está escrutando para que el dispositivo sepa qué datos debe devolver (si tiene múltiples flujos). El paquete de testigo OUT anuncia que a continuación vienen datos para el dispositivo. Un cuarto tipo de paquete de testigo, SETUP (que no se muestra en la figura), sirve para configuración.

Además del paquete de testigo, hay otros tres tipos. Éstos son los paquetes DATA (que sirven para transmitir hasta 64 bytes de información en cualquier dirección), los de saludo y los especiales. En la figura 3-54 se muestra el formato de un paquete de datos: consiste en un campo de sincronización de 8 bits, un tipo de paquete (PID) de 8 bits, la carga útil, y un **CRC** (**Código de Redundancia Cíclica**) para detectar errores. Se han definido tres clases de paquetes de saludo: ACK (el paquete de datos anterior se recibió correctamente), NAK (se detectó un error de CRC) y STALL (espera por favor; estoy ocupado).

Examinemos otra vez la figura 3-54. Cada 1.00 ms debe enviarse una trama del eje raíz, incluso si no hay trabajo. Las tramas 0 y 2 consisten en un solo paquete SOF, lo que indica que no hubo trabajo. La trama 1 es un escrutinio, de modo que principia con paquetes SOF e IN de la computadora al dispositivo de E/S, seguidos de un paquete DATA del dispositivo a la computadora. El paquete ACK le indica al dispositivo que los datos se recibieron correctamente. En caso de haber un error, se devolvería un NAK al dispositivo y el paquete se retransmitiría en el caso de datos de volumen (pero no en el de datos isocrónicos). La trama 3 tiene una estructura similar a la 1, excepto que ahora los datos fluyen de la computadora al dispositivo.

3.7 INTERFACES

Un sistema de computación de tamaño pequeño a mediano consiste en un chip de CPU, chips de memoria y algunos controladores de E/S, todos conectados por un bus. Ya estudiamos las memorias, las CPU y los buses con cierto detalle. Ha llegado el momento de examinar la última pieza del rompecabezas, los chips de E/S. Es a través de estos chips que la computadora se comunica con el mundo exterior.

3.7.1 Chips de E/S

Ya existen numerosos chips de E/S y constantemente se introducen nuevos. Entre los chips más comunes están los UART, los USART, los controladores de CRT, los controladores de disco y los PIO. Un **UART** (**receptor transmisor universal asincrónico**, *Universal Asynchronous Receiver Transmitter*) es un chip que puede leer un byte del bus de datos y enviarlo bit por bit a través de una línea serial hacia una terminal, o recibir datos de una terminal. Los UART por lo regular manejan varias velocidades; anchuras de caracteres de 5 a 8 bits; 1, 1.5 o 2 bits de paro; y paridad par, impar o ninguna, todo bajo control de un programa. Un **USART** (**receptor transmisor universal sincrónico asincrónico**, *Universal Synchronous Asynchronous Receiver Transmitter*) puede manejar transmisiones sincrónicas utilizando diversos protocolos, además de realizar todas las funciones del UART. Puesto que

ya examinamos los UART en el capítulo 2, estudiemos ahora la interfaz paralela como ejemplo de chip de E/S

Chips PIO

Un chip de **entrada/salida paralela** (**PIO**, *Parallel Input/Output*) representativo es el Intel 8255A, que se muestra en la figura 3-55. Este chip tiene 24 líneas de E/S que pueden conectarse con cualquier dispositivo compatible con TTL, por ejemplo, teclados, interruptores, lámparas o impresoras. En pocas palabras, el programa de la CPU puede escribir un 0 o un 1 en cualquier línea, o leer el estado de entrada de cualquier línea, lo que ofrece gran flexibilidad. Un sistema pequeño basado en CPU que usa un PIO a menudo puede sustituir a toda una tarjeta llena de chips SSI o MSI, sobre todo en los sistemas incorporados.

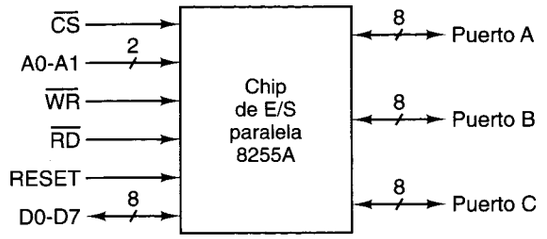


Figura 3-55. Chip PIO 8255A.

Aunque la CPU puede configurar el 8255A de muchas maneras cargando registros de situación dentro del chip, nos concentraremos en algunos de los modos de operación más sencillos. La forma más simple de usar el 8255A es como tres puertos de 8 bits independientes, A, B y C. Cada puerto tiene asociado un registro latch de 8 bits. Para establecer las líneas de un puerto, la CPU sólo tiene que escribir un número de ocho bits en el registro correspondiente; el número de ocho bits aparecerá en las líneas de salida y permanecerá ahí hasta que se coloque un valor distinto en el registro. Si la CPU quiere usar un puerto para recibir entradas, sólo tiene que leer el registro correspondiente.

Otros modos de operación permiten sincronizarse con dispositivos externos. Por ejemplo, para enviar salidas a un dispositivo que no siempre está listo para aceptar datos, el 8255A puede presentar datos en un puerto de salida y esperar a que el dispositivo envíe de vuelta una pulsación para indicar que aceptó los datos y quiere más. La lógica necesaria para captar tales pulsaciones y proporcionarlas a la CPU viene incluida en el hardware del 8255A.

En el diagrama funcional del 8255A podemos ver que, además de 24 terminales para los tres puertos, el chip tiene ocho líneas que se conectan directamente con el bus de datos, una línea de selección de chip, líneas de lectura y escritura, dos líneas de dirección y una línea para restablecer el chip. Las dos líneas de dirección (A0 y A1) seleccionan uno de los cuatro registros internos, que corresponden a los puertos A, B, C y el registro de situación, que tiene bits que determinan cuáles puertos son para entrada y cuáles para salida, además de otras

funciones. Normalmente, las dos líneas de dirección están conectadas a los bits de orden bajo del bus de dirección.

3.7.2 Decodificación de direcciones

Hasta ahora hemos evitado deliberadamente explicar con claridad cómo se habilita la línea de selección de chip (CS) en los chips de memoria y de E/S que hemos examinado. Ha llegado el momento de ver con más detalle cómo se hace esto. Consideremos una sencilla computadora de 16 bits que consiste de una CPU, una EPROM de 2K × 8 bytes para el programa, una RAM de 2K × 8 bytes para los datos, y un PIO. Este pequeño sistema podría usarse como prototipo del cerebro de un juguete barato o un aparato doméstico sencillo. Ya en producción, la EPROM podría sustituirse por una ROM.

El PIO puede seleccionarse de una de dos maneras: como verdadero dispositivo de E/S o como parte de la memoria. Si optamos por usarla como dispositivo de E/S, deberemos seleccionarla usando una línea de bus explícita que indique que se está haciendo referencia a un dispositivo de E/S, no a la memoria. Si usamos la otra estrategia, **E/S con mapa en la memoria**, deberemos asignar 4 bytes del espacio de memoria para los tres puertos y el registro de control. La decisión es un tanto arbitraria. Escogeremos E/S con mapa en la memoria porque ilustra algunos aspectos interesantes de las interfaces de E/S.

La EPROM necesita 2K de espacio de direcciones, la RAM también necesita 2K de espacio de direcciones, y el PIO necesita 4 bytes. Puesto que en nuestro ejemplo el espacio de direcciones es de 64K, deberemos decidir dónde poner los tres dispositivos. Una posible opción se muestra en la figura 3-56. La EPROM ocupa las direcciones hasta 2K, la RAM ocupa las direcciones de 32K a 34K, y el PIO ocupa los 4 bytes más altos del espacio de direcciones, 65532 a 65535. Desde el punto de vista del programador no importa cuáles direcciones se usen, pero para las interfaces sí importa. Si hubiéramos optado por direccionar el PIO a través del espacio de E/S, no necesitaría direcciones de memoria (pero sí necesitaría cuatro direcciones del espacio de E/S).

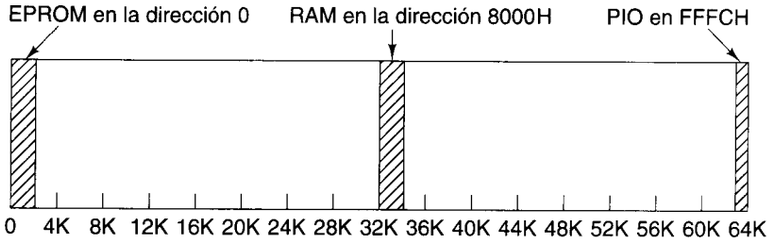


Figura 3-56. Ubicación de la EPROM, la RAM y el PIO en nuestro espacio de direcciones de 64K.

Con las asignaciones de direcciones de la figura 3-56, la EPROM deberá seleccionarse con cualquier dirección de memoria de 16 bits de la forma 00000xxxxxxxxxx (binario). En otras palabras, cualquier dirección de memoria cuyos 5 bits de orden alto sean todos cero

quedará en los 2K más bajos de la memoria, y por ende en la EPROM. Así, la selección de chip de la EPROM podría conectarse a un comparador de 5 bits, una de cuyas entradas estaría conectada permanentemente a 00000.

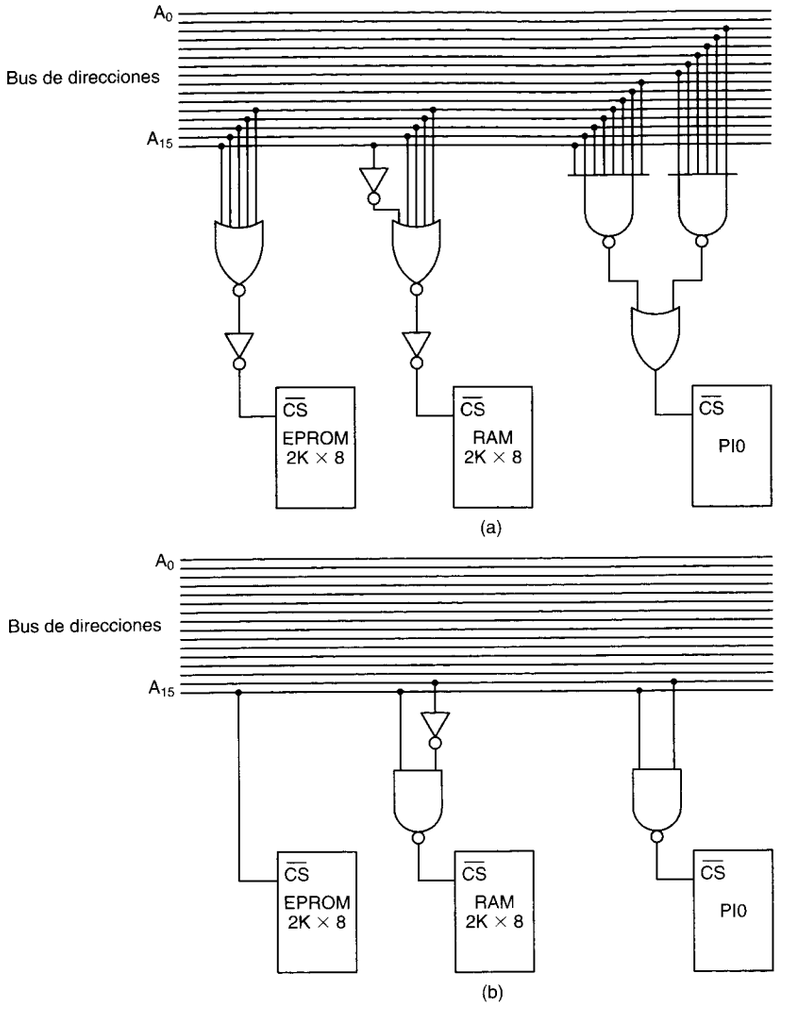


Figura 3-57. (a) Decodificación de dirección completa. (b) Decodificación de dirección parcial.

Una mejor forma de lograr el mismo efecto es usar una compuerta OR de cinco entradas, conectando éstas a las líneas de dirección A11 a A15. Sólo si las cinco líneas son 0 la salida será 0, lo que habilitará $\overline{CS}$. Lo malo es que no existen compuertas OR de cinco entradas en las

series SSI estándar. Lo más cercano que hay es una compuerta NOR de ocho entradas. Si conectamos a tierra tres entradas e invertimos la salida podremos producir la señal correcta, como se muestra en la figura 3-57(a). Los chips SSI son tan baratos que, excepto en circunstancias especiales, no importa si se usa uno de ellos de forma ineficiente. Por convención, las entradas que no se usan no se muestran en los diagramas de circuito.

Se puede usar el mismo principio para la RAM, sólo que en este caso la RAM deberá responder a direcciones binarias de la forma 10000xxxxxxxxxx. Por ello, se necesita un inversor adicional como se muestra en la figura. La decodificación de direcciones del PIO es un poco más complicada, porque se selecciona con las cuatro direcciones de la forma 11111111111111xx. En la figura se muestra un posible circuito que habilita $\overline{CS}$ sólo cuando la dirección correcta aparece en el bus de direcciones. Este circuito usa dos compuertas NAND de ocho entradas para alimentar una compuerta OR. La construcción de la lógica de decodificación de direcciones de la figura 3-57(a) utilizando SSI requiere seis chips: los cuatro chips de ocho entradas, una compuerta OR y un chip con tres inversores.

Sin embargo, si la computadora realmente consiste sólo en una CPU, dos chips de memoria y el PIO, podemos usar un truco para simplificar mucho la decodificación de direcciones. El truco se basa en el hecho de que todas las direcciones de la EPROM, y sólo esas direcciones, tienen un 0 en el bit de orden alto, A15. Por tanto, basta con conectar $\overline{CS}$ a A15 directamente, como se muestra en la figura 3-57(b).

A estas alturas la decisión de colocar la RAM en 8000H estará pareciendo menos arbitraria. La decodificación de RAM puede efectuarse observando que las únicas direcciones válidas de la forma 10xxxxxxxxxxxxxx están en la RAM, de modo que basta con dos bits de decodificación. Así mismo, cualquier dirección que comience con 11 deberá ser una dirección del PIO. La lógica de decodificación completa requiere ahora dos compuertas NAND y un inversor. Puesto que se puede crear un inversor a partir de una compuerta NAND con sólo unir las dos entradas, un solo chip NAND cuádruple es ahora más que suficiente.

La lógica de decodificación de direcciones de la figura 3-57(b) se llama **decodificación de direcciones parcial**, porque no se usan las direcciones completas. Esta lógica tiene la propiedad de que una lectura de las direcciones 0001000000000000, 0001100000000000 o 0010000000000000 dará el mismo resultado. De hecho, todas las direcciones de la mitad inferior del espacio de direcciones seleccionarán la EPROM. Dado que no se usan las direcciones adicionales, no hay problema, pero si se está diseñando una computadora que podría expandirse en el futuro (algo que no es muy probable en un juguete), debe evitarse la decodificación parcial porque ocupa mucho espacio de direcciones.

Otra técnica común para decodificar direcciones es emplear un decodificador, como el que se muestra en la figura 3-13. Si conectamos las tres entradas a las tres líneas de dirección de orden alto, obtendremos ocho salidas, que corresponden a direcciones en los primeros 8K, los segundos 8K, y así. En el caso de una computadora con ocho memorias RAM, cada una de 8K × 8, un chip de este tipo basta para toda la decodificación. En el caso de una computadora con ocho chips de memoria de 2K × 8 también es suficiente un solo decodificador, siempre que cada chip de memoria esté situado en un bloque de 8K distinto del espacio de direcciones. (Recuerde lo que dijimos antes, de que la posición de los chips de memoria y de E/S dentro del espacio de direcciones es importante.)

3.8 RESUMEN

Las computadoras se construyen con chips de circuitos integrados que contienen diminutos elementos de conmutación llamados compuertas. Las compuertas más comunes son AND, OR, NAND, NOR y NOT. Es posible construir directamente circuitos sencillos combinando compuertas individuales.

Otros circuitos más complejos son los multiplexores, desmultiplexores, codificadores, decodificadores, desplazadores y los ALU. Pueden programarse funciones booleanas arbitrarias empleando un PLA. Si se requieren muchas funciones booleanas, los PLA a menudo son más eficientes. Podemos utilizar las leyes del álgebra booleana para transformar circuitos de una forma a otra. En muchos casos es posible producir circuitos más económicos.

La aritmética de computadora se efectúa con sumadores. Se puede construir un sumador completo de un bit con dos medios sumadores. Un sumador para palabras de varios bits se puede construir conectando varios sumadores completos de modo que el acarreo de salida de cada uno se alimente a su vecino de la izquierda.

Los componentes de las memorias (estáticas) son latches y flip-flops, que pueden almacenar un bit de información. Éstos pueden combinarse linealmente para formar latches y flip-flops octales, o logarítmicamente para crear memorias a gran escala orientadas hacia las palabras. Los tipos de memoria disponibles son RAM, ROM, PROM, EPROM, EEPROM y flash. Las RAM estáticas no necesitan refrescarse; conservan sus valores en tanto hay alimentación eléctrica. Las RAM dinámicas, en cambio deben refrescarse periódicamente para compensar las fugas de los diminutos condensadores del chip.

Los componentes de un sistema de computación se conectan con buses. Muchas de las terminales de un chip de CPU típico, pero no todas, alimentan directamente una línea de bus. Las líneas de bus pueden dividirse en líneas de dirección, de datos y de control. Los buses sincrónicos se controlan con un reloj maestro. Los buses asincrónicos utilizan protocolos completos para sincronizar el esclavo con el iniciador.

El Pentium II es un ejemplo de CPU moderna. Los sistemas modernos que lo usan tienen un bus de memoria, un bus PCI, un bus ISA y un bus USB. El bus PCI puede transferir 64 bits a la vez con una rapidez de 66 MHz, lo que lo hace lo bastante rápido para casi todos los periféricos, pero no para la memoria.

Los interruptores, las lámparas, las impresoras y muchos otros dispositivos de E/S pueden conectarse a las computadoras utilizando chips de E/S paralela como el 8255A. Estos chips pueden configurarse de modo que sean parte del espacio de E/S o del espacio de memoria, según sea necesario. Pueden decodificarse total o parcialmente, dependiendo de la aplicación.

PROBLEMAS

1. Un lógico llega a un restaurante de comida rápida y dice: "Quiero una hamburguesa o una salchicha y papas fritas". Por desgracia, el cocinero no pasó del sexto grado y no sabe (ni le importa) si "y" tiene precedencia o no sobre "o". En lo que a él concierne, una interpretación es tan buena como otra. ¿Cuáles de los siguientes casos son interpretaciones válidas de la orden? (Cabe señalar que en español "o" es una "o exclusiva".)

   a. Sólo una hamburguesa.
   b. Sólo una salchicha.
   c. Sólo papas fritas.
   d. Una salchicha y papas fritas.
   e. Una hamburguesa y papas fritas.
   f. Una salchicha y una hamburguesa.
   g. Las tres cosas.
   h. Nada; el lógico se queda con hambre por pasarse de listo.

2. Un misionero perdido en el sur de California se detiene en una bifurcación del camino. Él sabe que dos pandillas de motociclistas habitan en la región, una que siempre dice la verdad y una que siempre miente. Él quiere saber cuál camino lleva a Disneyland. ¿Qué pregunta deberá hacer?

3. Existen cuatro funciones booleanas de una sola variable y 16 funciones de dos variables. ¿Cuántas funciones de tres variables hay? ¿De n variables?

4. Utilice una tabla de verdad para demostrar que $P = (P \text{ AND } Q) \text{ OR } (P \text{ AND NOT } Q)$.

5. Muestre cómo puede construirse la función AND con dos compuertas NAND.

6. Use la ley de DeMorgan para encontrar el complemento de $\overline{AB}$.

7. Utilizando el chip multiplexor de tres variables de la figura 3-12, implemente una función cuya salida sea la paridad de las entradas, es decir, la salida es 1 si y sólo si un número impar de entradas son 1.

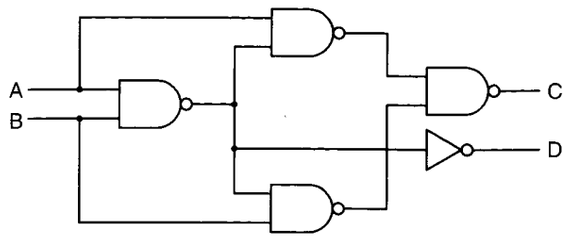
8. Póngase su gorro de pensar. El chip multiplexor de tres variables de la figura 3-12 en realidad puede calcular una función arbitraria de *cuatro* variables booleanas. Describa cómo y dé un ejemplo, dibuje el diagrama de lógica de la función que es 0 si la palabra en inglés para el número de fila de la tabla de verdad tiene un número impar de letras (por ejemplo, 0000 = cero = cuatro letras → 0; 0111 = siete = cinco letras → 1; 1101 = trece = ocho letras → 0). *Sugerencia*: Si llamamos a la cuarta variable de entrada D, las ocho líneas de entrada podrían conectarse a V_{cc}, tierra, D o $\overline{D}$.

9. Dibuje el diagrama de lógica de un demultiplexor de 2 bits, un circuito cuya única línea de entrada se envía a una de las cuatro líneas de salida dependiendo del estado de las dos líneas de control.

10. Dibuje el diagrama de lógica de un codificador de dos bits, con cuatro líneas de entrada, de las cuales exactamente una está alta en cualquier instante, y dos líneas de salida cuyo valor binario de dos bits indica cuál entrada está alta.

11. Redibuje el PLA de la figura 3-15 con suficiente detalle para mostrar cómo puede implementarse la función lógica de mayoría de la figura 3-3. En particular, no olvide mostrar cuáles conexiones están presentes en ambas matrices.

12. ¿Qué hace este circuito?



13. Un chip MSI común es un sumador de 4 bits. Cuatro de estos chips pueden conectarse para formar un sumador de 16 bits. ¿Cuántas terminales esperaría usted que tuviera el chip sumador de 4 bits? ¿Por qué?

14. Se puede construir un sumador de n bits conectando en serie n sumadores completos, con el acarreo de entrada a la etapa i, C_i, tomado de la salida de la etapa $i-1$. El acarreo de entrada de la etapa 0, C_0, es 0. Si cada etapa tarda T ns en producir su suma y acarreo, el acarreo de entrada de la etapa i no será válido sino hasta iT ns después de iniciada la suma. Si n es grande, el tiempo necesario para que el acarreo se propague hasta la etapa de orden alto podría ser inaceptablemente largo. Diseñe un sumador que funcione con mayor rapidez. *Sugerencia:* Cada C_i se puede expresar en términos de los bits del operando A_{i-1} y B_{i-1} así como el acarreo C_{i-1}. Con esta relación es posible expresar C_i en función de las entradas de las etapas 0 a $i-1$, y todos los acarreos pueden generarse simultáneamente.

15. Si todas las compuertas de la figura 3-19 tienen un retraso de propagación de 10 ns, y todos los demás retrasos son insignificantes, ¿en qué tiempo mínimo podemos estar seguros de que un circuito que usa este diseño tiene un bit de salida válido?

16. La ALU de la figura 3-20 puede realizar sumas de 8 bits en complemento a 2. ¿También puede efectuar restas en complemento a 2? Si esto es posible, explique cómo. Si no, modifíquela para que pueda efectuar restas.

17. A veces es útil que una ALU de 8 bits como la de la figura 3-20 genere la constante –1 como salida. Sugiera dos formas distintas de hacerlo. En cada caso, especifique los valores de las seis señales de control.

18. Una ALU de 16 bits se construye con 16 ALU de un bit, cada una de las cuales tiene un tiempo de suma de 10 ns. Si hay un retraso adicional de 1 ns por la propagación de una ALU a la siguiente, ¿cuánto tardará en aparecer el resultado de una suma de 16 bits?

19. ¿Cuál es el estado de reposo de las entradas S y R de un latch SR que se construye con dos compuertas NAND?

20. El circuito de la figura 3-26 es un flip-flop que se dispara en el flanco ascendente del reloj. Modifique este circuito para producir un flip-flop que se dispare en el flanco descendente del reloj.

21. Para ayudarse a hacer los pagos de su nueva computadora personal, usted está realizando trabajos de consultoría para compañías que se inician en la fabricación de chips SSI. Uno de sus clientes está pensando en sacar un chip que contenga cuatro flip-flops D, cada uno de los cuales produzca tanto Q como $\overline{Q}$, por solicitud de un cliente que podría ser importante. El diseño propuesto tiene las cuatro señales de reloj conectadas juntas, también por solicitud. No hay ni señal de preestablecer ni señal de borrar. Su tarea es dar una evaluación profesional del diseño.

22. La memoria 4×3 de la figura 3-29 usa 22 compuertas AND y tres compuertas OR. Si el circuito se fuera a expandir a 256×8, ¿cuántas compuertas de cada tipo se necesitarían?

23. A medida que aumenta más y más la capacidad de memoria en un solo chip, también se incrementa el número de líneas necesarias para direccionarla. En muchos casos no es conveniente tener un gran número de líneas de dirección en un chip. Proponga una forma de direccionar 2^n palabras de memoria utilizando menos de n líneas.

24. Una computadora con un bus de datos de 32 bits emplea chips de memoria RAM dinámica $1M \times 1$. ¿Qué tamaño mínimo de memoria (en bytes) puede tener esta computadora?

25. Remitiéndonos al diagrama de temporización de la figura 3-37, suponga que frenara el reloj hasta tener un periodo de 40 ns en lugar de 25 ns como se muestra, pero que las restricciones de temporización no cambiaran. ¿Cuánto tiempo tendría la memoria para colocar los datos en el bus durante T_3 después de habilitarse $\overline{\text{MREQ}}$, en el peor de los casos?

26. Remítase otra vez a la figura 3-37 y suponga que el reloj sigue siendo de 40 MHz pero T_{AD} se aumentara a 16 ns. ¿Podrían seguir usándose chips de memoria de 40 ns?

27. En la figura 3-37(b) se especifica que T_{ML} es de por lo menos 6 ns. ¿Puede imaginar un chip en el que este tiempo sea negativo? En otras palabras, ¿la CPU podría habilitar $\overline{\text{MREQ}}$ antes de que la dirección se estabilice? ¿Por qué sí o por qué no?

28. Suponga que la transferencia de bloque de la figura 3-41 se efectuará por el bus de la figura 3-37. ¿Qué ancho de banda adicional se obtendría usando una transferencia de bloque en lugar de transferencias individuales en el caso de bloques largos? Suponga ahora que el bus tiene una anchura de 32 bits en lugar de 8 bits. Conteste otra vez la pregunta.

29. Denote los tiempos de transición de las líneas de dirección de la figura 3-38 como T_{A1} y T_{A2}, y los tiempos de transición de $\overline{\text{MREQ}}$ como T_{MREQ1} y T_{MREQ2}, etc. Escriba todas las desigualdades implícitas en el saludo completo.

30. Casi todos los buses de 32 bits permiten lecturas y escrituras de 16 bits. ¿Existe ambigüedad respecto a dónde deben colocarse los datos? Comente.

31. Muchas CPU tienen un tipo de ciclo de bus especial para acusar recibo de interrupciones. ¿Por qué?

32. Una PC/AT de 10 MHz necesita cuatro ciclos para leer una palabra. ¿Qué ancho de banda de bus consume la CPU?

33. Una CPU de 32 bits con líneas de dirección A2-A31 requiere que todas las referencias a la memoria estén alineadas; es decir, las palabras tienen que direccionarse en múltiplos de 4 bytes, y las medias palabras tienen que direccionarse en bytes pares. Los bytes pueden estar en cualquier lugar. ¿Cuántas combinaciones válidas hay para las lecturas de memoria, y cuántas líneas se necesitan para expresarlas? Dé dos respuestas y justifique cada una.

34. ¿Por qué es imposible que el Pentium II funcione en un bus PCI de 32 bits sin perder funcionalidad? Después de todo, otras computadoras con un bus de datos de 64 bits pueden efectuar transferencias de 32 bits, 16 bits e incluso 8 bits.

35. Suponga que una CPU tiene una caché nivel 1 y una caché nivel 2, con tiempos de acceso de 5 ns y 10 ns, respectivamente. El tiempo de acceso a la memoria principal es de 50 ns. Si el 20% de los accesos son aciertos en la caché nivel 1 y el 60% son aciertos en la caché nivel 2, calcule el tiempo de acceso promedio.

36. ¿Es probable que un sistema picoJava II incorporado pequeño incluya un chip 8255A?

37. Calcule el ancho de banda de bus necesario para exhibir una película VGA (640 × 480) en color real a 30 cuadros/s. Suponga que los datos deben pasar por el bus dos veces, una vez del CD-ROM a la memoria y una vez de la memoria a la pantalla.

38. ¿Cuál señal del Pentium II cree usted que alimente la línea FRAME# del bus PCI?

39. ¿Cuál de las señales de la figura 3-53 no es estrictamente necesaria para que funcione el protocolo de bus?

40. Las instrucciones de cierta computadora requieren dos ciclos de bus cada una, uno para traer la instrucción y uno para traer los datos. Cada ciclo de bus tarda 250 ns y cada instrucción tarda 500 ns (es decir, el tiempo de procesamiento interno es insignificante). La computadora también tiene un disco con 16 sectores de 512 bytes por pista. El tiempo de rotación del disco es de 8.092 ms. ¿A qué porcentaje de su velocidad normal se reduce la computadora durante una transferencia de DMA si cada una de estas transferencias tarda un ciclo de bus? Considere dos casos: transferencias de bus de 8 bits y transferencias de bus de 16 bits.

41. La carga útil máxima de un paquete de datos isocrónico en el bus USB es de 1023 bytes. Si suponemos que un dispositivo sólo puede enviar un paquete de datos por trama, ¿qué ancho de banda máximo puede tener un solo dispositivo isocrónico?

42. ¿Qué efecto tendría añadir una tercera línea de entrada a la compuerta AND que selecciona el PIO de la figura 3-57(b) si dicha línea nueva se conectara a A13?

43. Escriba un programa que simule el comportamiento de una matriz $m \times n$ de compuertas NAND de dos entradas. Este circuito, contenido en un chip, tiene j terminales de entrada y k terminales de salida. Los valores de j, k, m y n son parámetros de la simulación que se fijan en el momento de la compilación. El programa debe iniciar con la lectura de una "lista de alambrado" en la que cada alambre especifica una entrada y una salida. Una entrada es una de las j terminales de entrada o bien la salida de alguna compuerta NAND. Una salida es una de las k terminales de salida o bien una entrada de alguna compuerta NAND. Las entradas que no se usan están en 1 lógico. Después de leer la lista de alambrado, el programa deberá imprimir la salida de cada una de las 2^j posibles entradas. Los chips con matrices de compuertas como éste se usan ampliamente para colocar circuitos personalizados en un chip porque casi todo el trabajo (depositar la matriz de compuertas en el chip) es independiente del circuito por implementar. Sólo el alambrado es específico para cada diseño.

44. Escriba un programa que lea dos expresiones booleanas arbitrarias y determine si respresentan la misma función. El lenguaje de entrada deberá incluir letras individuales como variables booleanas, los operandos AND, OR y NOT, y paréntesis. Cada expresión deberá caber en una línea de entrada. El programa deberá calcular las tablas de verdad de ambas funciones y compararlas.

45. Escriba un programa que lea una colección de expresiones booleanas y calcule las matrices 24×50 y 50×6 necesarias para implementar esas expresiones con el PLA de la figura 3-15. El lenguaje de entrada deberá ser el mismo que en el problema anterior. Imprima las matrices en la impresora de líneas.

4

EL NIVEL DE MICROARQUITECTURA

El nivel que está arriba del de lógica digital es el de la microarquitectura. Su tarea es implementar el nivel ISA (arquitectura de conjunto de instrucciones) que está arriba de él, como se ilustra en la figura 1-2. El diseño del nivel de microarquitectura depende de la ISA que se está implementando, así como de los objetivos de costo y desempeño de la computadora. Muchas ISA modernas, en especial los diseños RISC, tienen instrucciones sencillas que por lo regular pueden ejecutarse en un solo ciclo de reloj. Las ISA más complejas, como la de Pentium II, podrían requerir muchos ciclos para ejecutar una sola instrucción. Dicha ejecución podría requerir encontrar los operandos en la memoria, leerlos y almacenar resultados de vuelta en la memoria. La secuencia de operaciones dentro de una sola instrucción con frecuencia obliga a adoptar un diferente enfoque de control que con las ISA sencillas.

4.1 UN EJEMPLO DE MICROARQUITECTURA

Lo ideal sería comenzar explicando los principios generales del diseño de microarquitecturas. Lo malo es que no existen principios generales; cada una es un caso especial. Por tanto, optaremos por analizar un ejemplo detallado. Como ejemplo de ISA hemos escogido un subconjunto de la Máquina Virtual Java, como prometimos en el capítulo 1. Este subconjunto contiene sólo instrucciones para manejar enteros, por lo que lo hemos llamado **IJVM** (*Integer Java Virtual Machine*). Examinaremos la JVM completa en el capítulo 5.

Comenzaremos por describir la microarquitectura sobre la cual implementaremos IJVM. La IJVM tiene algunas instrucciones relativamente complejas. Muchas arquitecturas de este

tipo se han implementado con microprogramación, como vimos en el capítulo 1. Aunque IJVM es pequeña, es un buen punto de partida para describir el control y la secuencia de las instrucciones.

Nuestra microarquitectura contendrá un microprograma (en ROM) cuya tarea es buscar, decodificar y ejecutar instrucciones IJVM. No podemos usar el intérprete de JVM de Sun en vez del microprograma porque necesitamos un microprograma diminuto que alimente de forma eficiente las compuertas individuales del hardware real. El intérprete de JVM de Sun, en cambio, se escribió en C para ser transportable, y no puede controlar el hardware con el nivel de detalle que necesitamos. Puesto que el hardware real empleado consiste únicamente en los componentes básicos que describimos en el capítulo 3, en teoría, después de entender plenamente este capítulo, el lector deberá poder salir y comprar una gran bolsa llena de transistores, y construir este subconjunto de la máquina JVM. Los estudiantes que completen con éxito esta tarea recibirán crédito extra (y un examen psiquiátrico completo).

Un modelo conveniente para el diseño de la microarquitectura es ver el diseño como un problema de programación, en el que cada instrucción en el nivel ISA es una función que el programa maestro debe invocar. En este modelo, el programa maestro es un ciclo infinito simple que determina cuál función se invocará, llama a la función, y vuelve a comenzar, casi como se muestra en la figura 2-3.

El microprograma tiene un conjunto de variables, llamadas **estado** de la computadora, al que todas las funciones tienen acceso. Cada función modifica al menos algunas de las variables que constituyen el estado. Por ejemplo, el Contador de Programa (PC) forma parte del estado; indica la localidad de memoria que contiene la siguiente función (o sea, instrucción de ISA) que se ejecutará. Durante la ejecución de cada instrucción, el PC se incrementa para apuntar a la siguiente instrucción por ejecutar.

Las instrucciones de IJVM son cortas y precisas. Cada instrucción tiene unos cuantos campos, por lo regular uno o dos, cada uno de los cuales tiene un propósito específico. El primer campo de cada instrucción es el **código de operación** (**opcode**) que identifica la instrucción como, por ejemplo, ADD (sumar) o BRANCH (ramificar) o algo más. Muchas instrucciones tienen un campo adicional que especifica el operando. Por ejemplo, las instrucciones que accesan una variable local necesitan un campo para indicar *cuál* variable.

Este modelo de ejecución, llamado **ciclo de búsqueda-ejecución**, es útil en lo abstracto y también podría ser la base para implementar ISA como IJVM que tienen instrucciones complejas. A continuación describiremos cómo funciona, qué aspecto tiene la microarquitectura, y cómo se controla con las microinstrucciones, cada una de las cuales controla la trayectoria de los datos durante un ciclo. La lista de todas las microinstrucciones constituye el microprograma, que presentaremos y analizaremos con detalle.

4.1.1 La trayectoria de datos

La **trayectoria de datos** es la parte de la CPU que contiene la ALU, sus entradas y sus salidas. La trayectoria de datos de nuestro ejemplo de microarquitectura se muestra en la figura 4-1. Si bien esta trayectoria de datos se optimó minuciosamente para interpretar programas en IJVM, es muy similar al que se usa en la mayor parte de las máquinas. Esta trayectoria contiene varios registros de 32 bits, a los que hemos asignado nombres simbólicos como PC, SP y MDR. Aunque algunos de estos nombres son conocidos, es importante entender que estos registros pueden accesarse sólo en el nivel de microarquitectura (por el microprograma). Reciben estos nombres porque casi siempre contienen un valor que corresponde a la variable del mismo nombre en la arquitectura del nivel ISA. Casi todos los registros

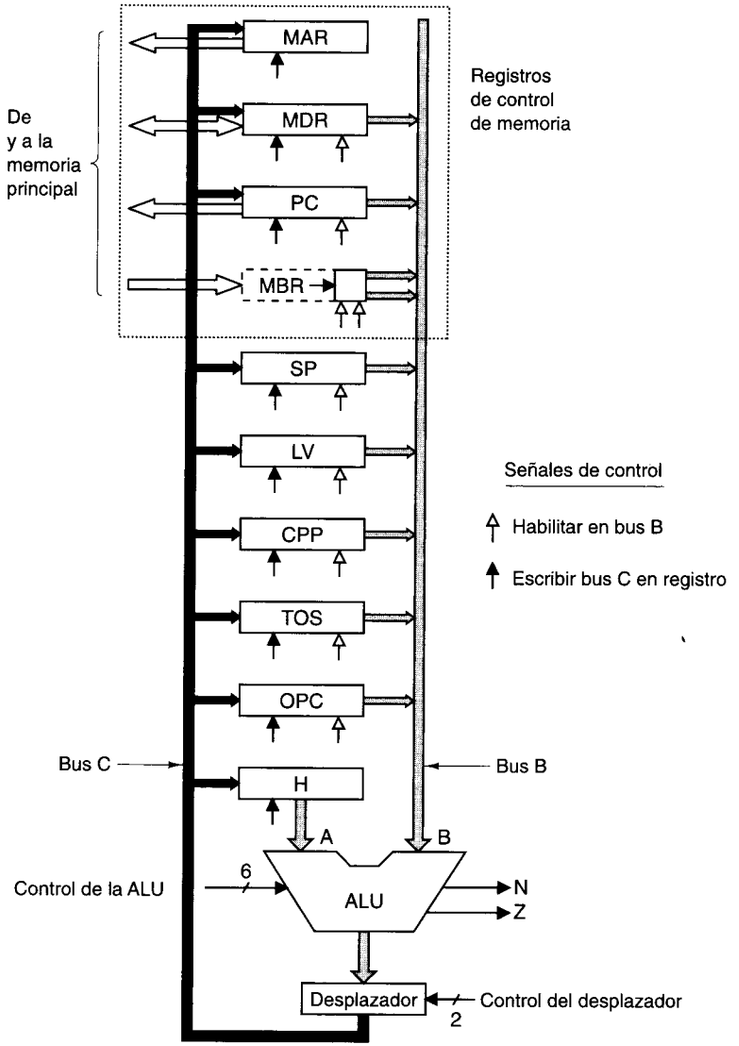


Figura 4-1. La trayectoria de datos de la microarquitectura que usaremos como ejemplo en este capítulo.

pueden colocar su contenido en el bus B. La salida de la ALU alimenta al desplazador y luego al bus C, cuyo valor puede escribirse en uno o más registros al mismo tiempo. No existe un bus A por el momento; lo añadiremos después.

La ALU es idéntica a la que se muestra en la figura 3-19 y la figura 3-20. Su función está determinada por seis líneas de control. La línea diagonal corta rotulada con "6" en la figura 4-1 indica que hay seis líneas de control de la ALU. Éstas son F_0 y F_1 para determinar el funcionamiento de la ALU, ENA y ENB para habilitar individualmente las salidas, INVA para invertir la entrada izquierda, e INC para forzar un acarreo de entrada en el bit de orden bajo, lo que en realidad suma 1 al resultado. Sin embargo, no todas las 64 combinaciones de las líneas de control de la ALU hacen algo útil.

Algunas de las combinaciones más interesantes se muestran en la figura 4-2. No todas estas funciones se necesitan en la IJVM, pero muchas de ellas serían útiles en la JVM completa. En muchos casos hay varias posibilidades para obtener el mismo resultado. En esta tabla, + significa suma aritmética y –significa resta aritmética, de modo que, por ejemplo, $-A$ indica el complemento a 2 de A.

F_0	F_1	ENA	ENB	INVA	INC	Función
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	1	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

Figura 4-2. Combinaciones útiles de señales de la ALU y la función que desempeñan.

La ALU de la figura 4-1 necesita dos entradas de datos: una entrada izquierda (A) y una entrada derecha (B). La entrada izquierda tiene conectado un registro de retención, H. La entrada derecha está conectada al bus B, que puede cargarse desde cualquiera de nueve orígenes, como indican las nueve flechas grises que lo tocan. Un diseño alternativo, con dos buses completos, tiene un conjunto diferente de ventajas y desventajas, mismas que veremos más adelante en este capítulo.

H se puede cargar inicialmente escogiendo una función de la ALU que se limite a pasar la entrada derecha (del bus B) hasta la salida de la ALU. Una función así sería sumar las entradas de la ALU, sólo que con ENA deshabilitada para que la entrada izquierda sea forzosamente cero. Sumar cero al valor del bus B produce ese mismo valor. Luego, este resultado puede pasarse por el desplazador sin modificación y guardarse en H.

Además de las funciones anteriores, se pueden usar otras dos líneas de control de forma independiente para controlar la salida de la ALU. SLL8 (desplazamiento lógico a la izquierda, *Shift Left Logical*) desplaza el contenido un byte a la izquierda, llenando los ocho bits menos significativos con ceros. SRA1 (desplazamiento aritmético a la derecha, *Shift Right Arithmetic*) desplaza el contenido un bit a la derecha, sin modificar el bit más significativo.

Es posible explícitamente leer y escribir el mismo registro en un ciclo. Por ejemplo, está permitido colocar SP en el bus B, deshabilitar la entrada izquierda de la ALU, habilitar la señal INC y guardar el resultado en SP con lo que se incrementa SP en 1 (vea la octava línea de la figura 4-2). ¿Cómo puede leerse y escribirse un registro en el mismo ciclo sin producir basura? La solución es que la lectura y la escritura se realizan realmente en diferentes momentos dentro del ciclo. Cuando un registro se selecciona como entrada derecha de la ALU, su valor se coloca en el bus B al principio del ciclo y se mantiene ahí continuamente durante todo el ciclo. Luego, la ALU efectúa su trabajo y produce un resultado que pasa al bus C a través del desplazador. Cerca del final del ciclo, cuando se sabe que las salidas de la ALU y del desplazador están estables, una señal de reloj activa el almacenamiento del bus C en uno más de los registros. Uno de estos registros bien podría ser el que proporcionó su entrada al bus B. La temporización precisa de la trayectoria de datos hace posible leer y escribir el mismo registro en un ciclo, como se describe en seguida.

Temporización de la trayectoria de datos

La temporización de estos sucesos se muestra en la figura 4-3. Aquí se produce una pulsación corta al principio de cada ciclo de reloj. La pulsación se puede sacar del reloj principal, como se muestra en la figura 3-21(c). En el flanco descendente de la pulsación, ya están preparados los bits que alimentarán todas las compuertas. Esto tarda un tiempo finito conocido, Δw. Luego se selecciona el registro que se necesita en el bus B y su valor se pasa a ese bus. Se requiere un tiempo Δx para que el valor se estabilice. Luego la ALU y el desplazador comienzan a operar con datos válidos. Después de un tiempo Δy adicional, las salidas de la ALU y el desplazador se han estabilizado. Después de otro intervalo Δz, los resultados se han propagado por el bus C hasta los registros, en los que pueden cargarse durante el flanco ascendente de la siguiente pulsación. La carga debe ser disparada por flanco y rápida, de modo que aun si algunos de los registros de entrada se modifican, los efectos no se harán sentir en el bus C sino hasta mucho después de haberse cargado todos los registros. También en el flanco ascendente de la pulsación, el registro que alimenta al bus B deja de hacerlo, en preparación del siguiente ciclo. MPC, MIR y la memoria se mencionan en la figura; exploraremos sus papeles en breve.

Es importante darse cuenta de que aunque no hay elementos de almacenamiento en la trayectoria de datos, hay un tiempo de propagación finito a través suyo. Modificar el valor

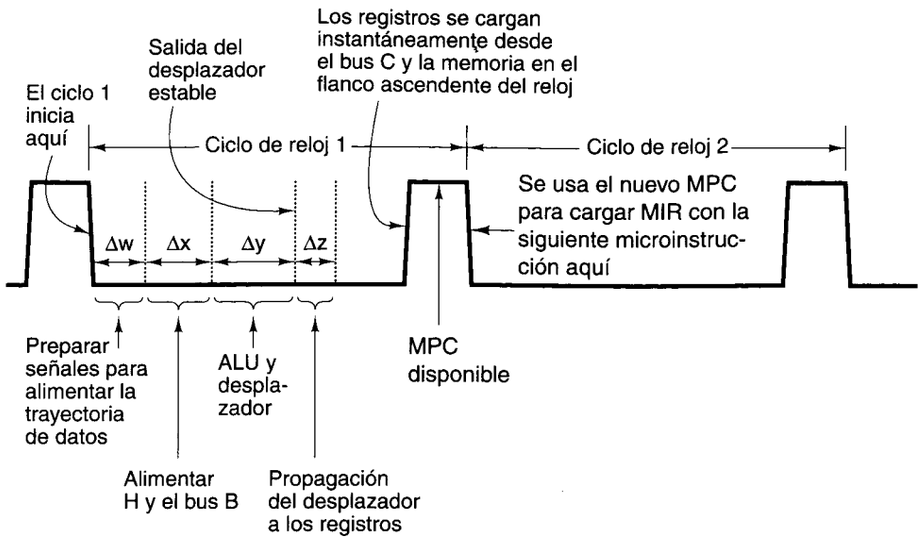


Figura 4-3. Diagrama de temporización de un ciclo de la trayectoria de datos.

que está en el bus B no hace que el bus C cambie sino hasta después de un tiempo finito (debido a los retrasos finitos en cada paso). Por consiguiente, aun si una operación de almacenamiento modifica uno de los registros de entrada, el valor ya estará guardado a salvo en el registro mucho antes de que el valor (ahora incorrecto) que se está colocando en el bus B (o en H) pueda llegar a la ALU.

Para que este diseño funcione se requiere una temporización rígida, un ciclo de reloj largo, un tiempo de propagación mínimo conocido a través de la ALU y un cargado rápido de los registros desde el bus C. Sin embargo, con una ingeniería cuidadosa se puede diseñar la trayectoria de datos de modo que funcione correctamente.

Una forma un tanto distinta de ver la trayectoria de datos es dividirlo conceptualmente en subciclos implícitos. El inicio del subciclo 1 se dispara por el flanco descendente del reloj. A continuación mostramos las actividades que ocurren durante los subciclos, junto con las longitudes de los subciclos (entre paréntesis).

1. Se preparan las señales de control (Δw).
2. Los registros se cargan en el bus B (Δx).
3. La ALU y el desplazador operan (Δy).
4. Los resultados se propagan por el bus C de regreso a los registros (Δz).

En el flanco ascendente del siguiente ciclo de reloj, los resultados se almacenan en los registros.

Dijimos que la mejor forma de ver los subciclos es como algo *implícito*. Con esto queremos decir que no hay pulsaciones de reloj ni otras señales implícitas que indiquen a la ALU cuándo debe operar o que digan a los resultados que entren en el bus C. En realidad, la ALU y

el desplazador están operando todo el tiempo. Sin embargo, sus entradas son basura hasta un tiempo $\Delta w + \Delta x$ después del flanco descendente del reloj. De igual manera, sus salidas son basura hasta que ha transcurrido $\Delta w + \Delta x + \Delta y$ después del flanco descendente del reloj. Las únicas señales explícitas que controlan la trayectoria de datos son el flanco descendente del reloj, que inicia el ciclo de la trayectoria de datos, y el flanco ascendente del reloj, que carga los registros a partir del bus C. Las otras fronteras de los subciclos están determinadas implícitamente por los tiempos de propagación inherentes a los circuitos participantes. Es responsabilidad de los ingenieros de diseño asegurarse de que el tiempo $\Delta w + \Delta x + \Delta y + \Delta z$ termine con tiempo de sobra antes del flanco ascendente del reloj, si se quiere que el cargado de los registros funcione siempre.

Operación de la memoria

Nuestra máquina tiene dos formas diferentes de comunicarse con la memoria: un puerto de memoria de 32 bits direccionable por palabra y un puerto de memoria de 8 bits direccionable por byte. El puerto de 32 bits está bajo el control de dos registros, MAR (**registro de dirección de memoria**, *Memory Address Register*) y MDR (**registro de datos de memoria**, *Memory Data Register*), como se muestra en la figura 4-1. El puerto de ocho bits está bajo el control de un registro, PC, que lee un byte y lo coloca en los ocho bits de orden bajo de MBR. Este puerto sólo puede leer datos de la memoria; no puede escribir datos en la memoria.

Cada uno de estos registros (y todos los demás registros de la figura 4-1) operan con una o dos **señales de control**. Una flecha blanca bajo un registro indica una señal de control que habilita la colocación de la salida del registro en el bus B. Puesto que MAR no está conectado con el bus B, no tiene una señal de habilitación. H tampoco tiene una porque siempre está habilitado, al ser la única posible entrada izquierda de la ALU.

Una flecha negra bajo un registro indica una señal de control que escribe (es decir, carga) el registro a partir del bus C. Puesto que MBR no se puede cargar a partir del bus C, no tiene una señal de escritura (aunque sí tiene otras dos señales de habilitación, que se describirán más adelante). Para iniciar una lectura o escritura de memoria, es preciso cargar los registros de memoria apropiados, y luego emitir una señal de lectura o escritura a la memoria (no se muestra en la figura 4-1).

MAR contiene direcciones de *palabra*, de modo que los valores 0, 1, 2, etc., se refieren a palabras consecutivas. PC contiene direcciones de *byte*, de modo que los valores 0, 1, 2, etc., se refieren a bytes consecutivos. Así, colocar un 2 en PC e iniciar una lectura de memoria hará que se lea el byte 2 de la memoria y se coloque en los 8 bits de orden bajo de MBR. Colocar un 2 en MAR e iniciar una lectura de memoria hará que se lean los bytes 8-11 (o sea, la palabra 2) de la memoria y se coloquen en MDR.

Esta diferencia en funcionalidad es necesaria porque MAR y PC se usarán para hacer referencia a dos partes diferentes de la memoria. La necesidad de esta distinción se aclarará posteriormente. Por ahora, basta decir que la combinación MAR/MDR se usa para leer y escribir palabras de datos en el nivel ISA, y la combinación PC/MBR sirve para leer el programa ejecutable en el nivel ISA, que consiste en un flujo de bytes. Todos los demás registros que contienen direcciones emplean direcciones de palabra, como MAR.

En la implementación física real, sólo hay una memoria y está organizada en bytes. Un sencillo truco permite a MAR contar en palabras (lo cual es necesario debido a la forma en que se definió la JVM) mientras la memoria física cuenta en bytes. Cuando MAR se coloca en el bus de direcciones, sus 32 bits no se corresponden directamente con las 32 líneas de dirección, 0-31. En vez de ello, el bit 0 de MAR se conecta con la línea 2 del bus de direcciones, el bit 1 de MAR se conecta con la línea 3 del bus de direcciones, y así. Los 2 bits superiores de MAR se desechan porque sólo se necesitan para las direcciones de palabra mayores que 2^{32}, que no son válidas en nuestra máquina de 4 GB. Con esta transformación, cuando MAR es 1, se coloca la dirección 4 en el bus; cuando MAR es 2, se coloca la dirección 8 en el bus, y así. Este truco se ilustra en la figura 4-4.

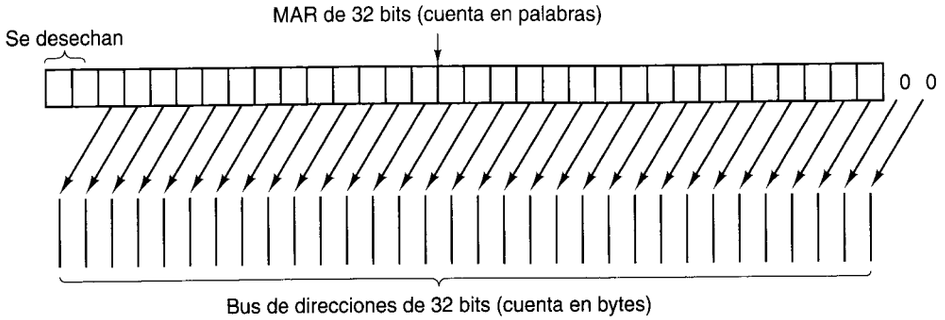


Figura 4-4. Correspondencia de los bits de MAR con el bus de direcciones.

Como ya se mencionó, los datos leídos de la memoria a través del puerto de memoria de ocho bits se devuelven en MBR, un registro de 8 bits. MBR puede copiarse en el bus B de una de dos formas: sin signo o con signo. Cuando se necesita un valor sin signo, la palabra de 32 bits que se coloca en el bus B contiene el valor de MBR en los 8 bits de orden bajo y ceros en los 24 bits superiores. Los valores sin signo son útiles como índices de tablas, o cuando es preciso armar un entero de 16 bits con dos bytes (sin signo) consecutivos del flujo de instrucciones.

La otra opción para convertir el MBR de 8 bits en una palabra de 32 bits es tratarlo como un valor con signo entre –128 y +127 y usar este valor para generar una palabra de 32 bits que tenga el mismo valor numérico. Esta conversión se efectúa copiando el bit de signo de MBR (el bit de la extrema izquierda) en las 24 posiciones de bit superiores del bus B, proceso que se llama **extensión del signo**. Cuando se escoge esta opción, los 24 bits superiores serán todos ceros o todos unos, dependiendo de si el bit de la izquierda del MBR es 0 o 1.

La decisión de convertir el MBR de 8 bits en un valor de 32 bits con o sin signo en el bus B depende de cuál de las dos líneas de control (flechas blancas debajo de MBR en la figura 4-1) está habilitada. Hay dos flechas porque se necesitan estas dos opciones. La capacidad de hacer que el MBR de 8 bits actúe como una fuente de 32 bits para el bus B se indica con el rectángulo punteado a la izquierda de MBR en la figura.

4.1.2 Microinstrucciones

Para controlar la trayectoria de datos de la figura 4-1 necesitamos 29 señales. Éstas se pueden dividir en cinco grupos funcionales, como se describe a continuación.

9 Señales para controlar la escritura de datos del bus C en registros.

9 Señales para controlar la habilitación de registros en el bus B para introducción en la ALU.

8 Señales para controlar las funciones de la ALU y el desplazador.

2 Señales (que no se muestran) para indicar lectura/escritura de memoria vía MAR/MDR.

1 Señal (que no se muestra) para indicar obtención de memoria vía PC/MBR.

Los valores de estas 29 señales de control especifican las operaciones durante un ciclo de la trayectoria de datos. Un ciclo consiste en colocar valores de los registros en el bus B, propagar las señales a través de la ALU y el desplazador, alimentarlas al bus C y por último escribir los resultados en el registro o registros apropiados. Además, si una señal de lectura de datos de memoria está habilitada, la operación de memoria se inicia al final del ciclo de la trayectoria de datos, una vez que se ha cargado MAR. Los datos de memoria están disponibles hasta el final del *siguiente* ciclo en MBR o MDR y se usa el ciclo *que sigue*. En otras palabras, una lectura de memoria por cualquiera de los puertos, iniciada al final del ciclo k, proporciona datos que no se usan en el ciclo $k + 1$, sino hasta el ciclo $k + 2$ o uno posterior.

Este comportamiento que aparentemente va contra la intuición se explica con la figura 4-3. Las señales de control de la memoria no se generan en el ciclo de reloj 1 sino hasta inmediatamente después de que MAR y PC se cargan en el flanco ascendente del reloj, hacia el final del ciclo de reloj 1. Supondremos que la memoria coloca sus resultados en los buses de memoria en el plazo de un ciclo, de modo que MBR y/o MDR se puedan cargar en el siguiente flanco ascendente del reloj, junto con los demás registros.

En otras palabras, cargamos MAR al final de un ciclo de la trayectoria de datos e iniciamos la lectura de memoria poco tiempo después. Por tanto, no podemos esperar realmente que los resultados de una operación de lectura estén en MDR al principio del siguiente ciclo, sobre todo si la pulsación de reloj es angosta. Si la memoria tarda un ciclo de reloj, simplemente no hay suficiente tiempo. Debe transcurrir un ciclo de la trayectoria de datos completa entre que se inicia una lectura de memoria y que se usa el resultado. Desde luego, se efectúan otras operaciones durante ese ciclo, pero no las que necesitan la palabra de la memoria.

El supuesto de que la memoria tarda un ciclo en operar equivale a suponer que la tasa de aciertos en caché es de 100%. Este supuesto nunca se cumple, pero la complejidad que implica tener un ciclo de memoria de longitud variable es mayor que con la que queremos lidiar aquí.

Puesto que MBR y MDR se cargan en el flanco ascendente del reloj, junto con todos los demás registros, podrían leerse durante ciclos en los que se está efectuando una nueva lectura de memoria. Esa lectura devuelve los valores antiguos, ya que todavía no ha habido tiempo de sobreescribirlos. No hay ambigüedad aquí; hasta que se carguen valores nuevos en MBR y MDR en el flanco ascendente del reloj, estos registros contendrán sus valores anteriores,

los cuales se podrán usar. Cabe señalar que es posible realizar lecturas una tras otra en dos ciclos consecutivos puesto que una lectura sólo tarda un ciclo. Además, ambas memorias podrían operar al mismo tiempo. Sin embargo, tratar de leer y escribir el mismo byte simultáneamente produce resultados no definidos.

Si bien podría ser deseable escribir la salida colocada en el bus C en más de un registro, nunca es conveniente habilitar más de un registro para el bus B al mismo tiempo. (De hecho, algunas implementaciones reales sufren daños físicos si se hace esto.) Si se agregan unos circuitos más, podemos reducir el número de bits necesarios para seleccionar una de las posibles fuentes que alimentan al bus B. Sólo hay nueve registros de entrada que pueden alimentar al bus B (si contamos por separado las versiones con signo y sin signo de MBR). Por tanto, podemos codificar la información del bus B en cuatro bits y usar un decodificador para generar las 16 señales de control, siete de las cuales no se necesitan. En un diseño comercial, los arquitectos estarían fuertemente tentados a deshacerse de uno de los registros para poder efectuar la tarea con sólo 3 bits. Como académicos, podemos darnos el lujo de desperdiciar un bit para contar con un diseño más limpio y sencillo.

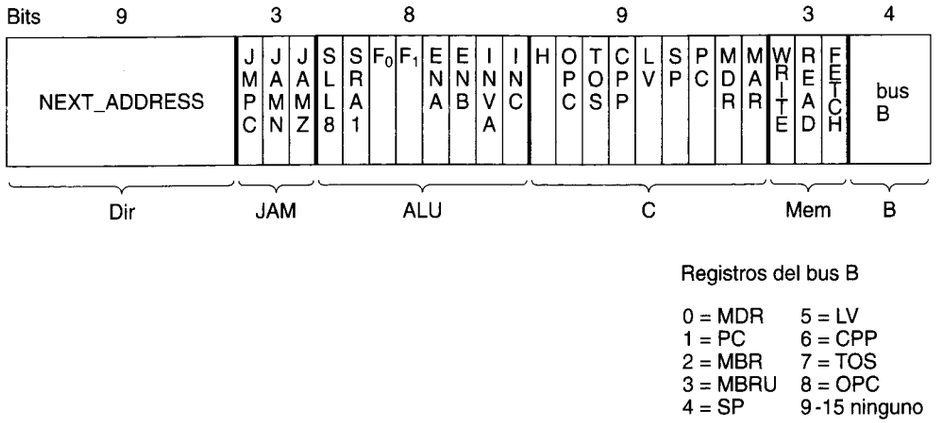


Figura 4-5. El formato de microinstrucciones para el Mic-1.

A estas alturas podemos controlar la trayectoria de datos con 9 + 4 + 8 + 2 + 1 = 24 señales, y por tanto 24 bits. Sin embargo, estos 24 bits sólo controlan la trayectoria de datos durante un ciclo. La segunda parte del control consiste en determinar qué debe hacerse en el siguiente ciclo. Para incluir esto en el diseño del controlador, crearemos un formato para describir las operaciones que se efectuarán usando los 24 bits de control más dos campos adicionales: el campo NEXT_ADDRESS y el campo JAM. A continuación explicaremos el contenido de cada uno de estos campos. La figura 4-5 muestra un posible formato, dividido en los seis grupos, y que contiene las 36 señales siguientes:

Dir – Contiene la dirección de una microinstrucción que podría ser la siguiente.
JAM – Determina cómo se selecciona la siguiente microinstrucción.
ALU – Funciones de la ALU y el desplazador.
C – Selecciona cuáles registros se escriben desde el bus C.
Mem – Funciones de memoria.
B – Selecciona la fuente del bus B; se codifica como se muestra.

El ordenamiento de los grupos es arbitrario en principio, aunque en realidad se escogió con mucho cuidado a fin de minimizar los cruces de líneas en la figura 4-6. Los cruces de línea en diagramas esquemáticos como el de la figura 4-6 a menudo corresponden a cruces de hilos en chips, que pueden causar problemas en los diseños bidimensionales y deben evitarse.

4.1.3 Control de microinstrucciones: el Mic-1

Hasta aquí hemos descrito cómo se controla la trayectoria de datos, pero todavía no hemos explicado cómo se decide cuál de las señales de control debe estar habilitada en cada ciclo. Esto lo determina un **secuenciador** que se encarga de recorrer la secuencia de operaciones necesarias para ejecutar una sola instrucción ISA.

El secuenciador debe producir dos tipos de información en cada ciclo:

1. El estado de cada señal de control en el sistema.
2. La dirección de la microinstrucción que debe ejecutarse a continuación.

La figura 4-6 es un diagrama de bloques detallado de la microarquitectura completa de nuestra máquina de ejemplo, que llamaremos **Mic-1**. A primera vista este diagrama podría parecer abrumador, pero vale la pena estudiarlo con detenimiento. Una vez que entienda plenamente todos y cada uno de los bloques y líneas de esta figura, habrá adelantado mucho en la comprensión del nivel de microarquitectura. El diagrama de bloques tiene dos partes: la trayectoria de datos, a la izquierda, que ya vimos con detalle, y la sección de control, a la derecha, que examinaremos a continuación.

El elemento más grande e importante de la porción de control de la máquina es una memoria llamada **almacén de control**. Es recomendable ver este "almacén" como una memoria que contiene todo el microprograma, aunque a veces se implementa como un conjunto de compuertas lógicas. En general, nos referiremos a él como almacén de control para evitar confundirlo con la memoria principal, a la que se accede a través de MBR y MDR. Sin embargo, en lo funcional, el almacén de control no es más que una memoria que contiene microinstrucciones en lugar de instrucciones ISA. En nuestra máquina de ejemplo, el almacén contiene 512 palabras, cada una de las cuales consiste en una microinstrucción de 36 bits como las que se ilustran en la figura 4-5. En realidad, no se necesitan todas estas palabras pero (por razones que se explicarán en breve) necesitamos direcciones para 512 palabras distintas.

Es importante notar que el almacén de control es muy diferente de la memoria principal: las instrucciones de la memoria principal tienen la propiedad de que se ejecutan en orden

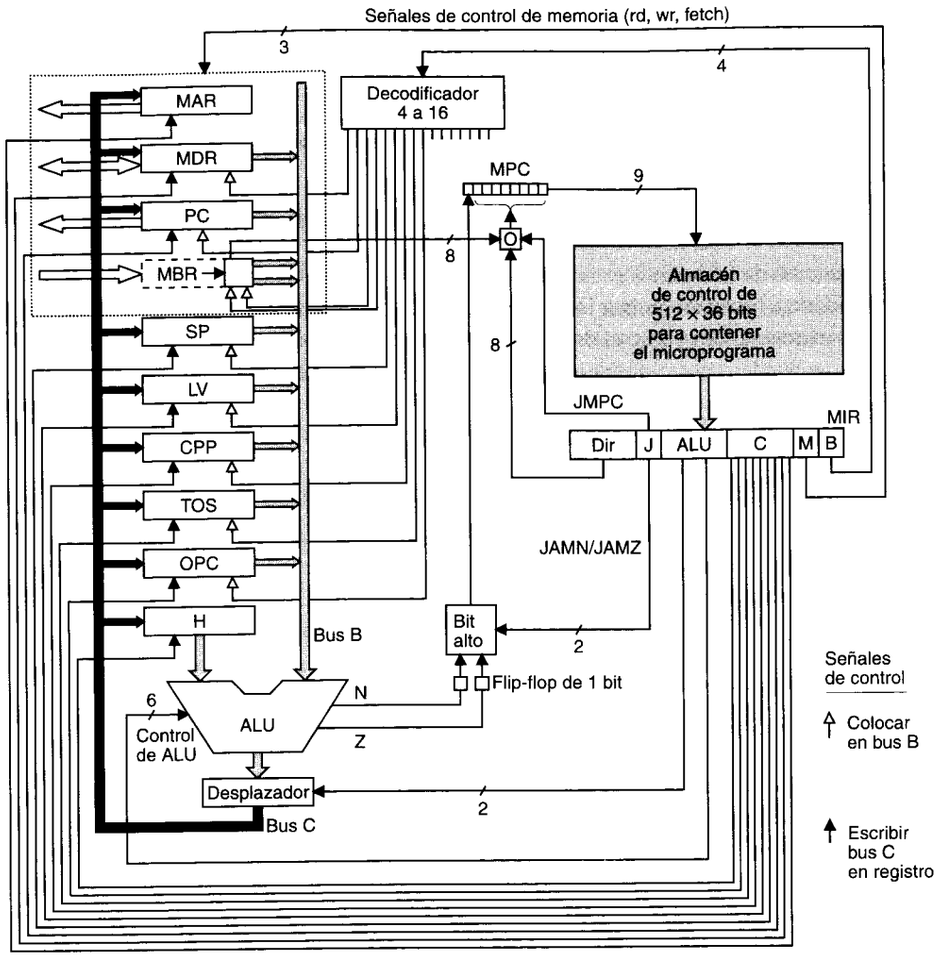


Figura 4-6. Diagrama de bloques completo de nuestra microarquitectura de ejemplo, Mic-1.

según su dirección (con excepción de las ramificaciones); las microinstrucciones no. El acto de incrementar el contador de programa en la figura 2-3 expresa el hecho de que la instrucción que se ejecutará por omisión después de la actual es la que sigue a la actual en la memoria. Los microprogramas necesitan mayor flexibilidad (porque las secuencias de microinstrucciones tienden a ser cortas), así que generalmente no tienen esta propiedad. En lugar de esto, cada microinstrucción especifica explícitamente su sucesora.

Puesto que el almacén de control es funcionalmente una memoria (sólo de lectura), necesita su propio registro de direcciones de memoria y su propio registro de datos de memoria. No necesita señales de leer y escribir, porque continuamente se está leyendo. Llamaremos al registro de dirección de memoria del almacén de control **MPC** (**contador de microprograma**, *MicroProgram Counter*). Este nombre es irónico porque las localidades que contiene no están ordenadas, explícitamente, de modo que el concepto de contar no es útil (pero, ¿quiénes somos nosotros para discutir con la tradición?). El registro de datos de memoria se llama **MIR** (**registro de microinstrucción**, *MicroInstruction Register*). Su función es almacenar la microinstrucción actual, cuyos bits alimentan las señales de control que operan la trayectoria de datos.

El registro MIR de la figura 4-6 contiene los mismos seis grupos de la figura 4-5. Los grupos Dir y J (por JAM) controlan la selección de la siguiente microinstrucción y se estudiarán más adelante. El grupo ALU contiene ocho bits que seleccionan la función de la ALU y controlan el desplazador. Los bits C hacen que registros individuales carguen la salida de la ALU desde el bus C. Los bits M controlan las operaciones de memoria.

Por último, los cuatro bits finales controlan el decodificador que determina qué se coloca en el bus B. En este caso hemos escogido un decodificador estándar 4 a 16, aunque sólo se requieren nueve posibilidades. En un diseño más afinado se usaría un decodificador de 4 a 9. La ventaja aquí es poder usar un circuito estándar tomado de una biblioteca de circuitos en lugar de diseñar uno a la medida. El uso del circuito estándar es más fácil y reduce la posibilidad de introducir errores. Crear circuitos propios ocupa menos área de chip pero tarda más y podríamos tener errores.

La operación de la figura 4-6 es como sigue. Al principio de cada ciclo de reloj (el flanco descendente del reloj de la figura 4-3), MIR se carga con la palabra del almacén de control a la que MPC apunta. El tiempo de carga de MIR se indica en la figura con Δw. Si pensamos en términos de subciclos, MIR se carga durante el primero.

Una vez que la microinstrucción está en MIR, las diversas señales se propagan a la trayectoria de datos. Se coloca un registro en el bus B, la ALU sabe qué instrucción debe ejecutar y ocurren muchas actividades más. Éste es el segundo subciclo. Después de un intervalo $\Delta w + \Delta x$ a partir del principio del ciclo, las entradas de la ALU se han estabilizado.

Después de transcurrir un tiempo Δy adicional, todo se ha calmado otra vez y las salidas de la ALU, N, Z y el desplazador están estables. Luego, los valores de N y Z se guardan en un par de flip-flops de un bit. Estos bits, al igual que todos los registros que se cargan a partir del bus C y la memoria, se guardan en el flanco ascendente del reloj, cerca del final del ciclo de la trayectoria de datos. La salida de la ALU no se almacena, sólo se alimenta al desplazador. La actividad de la ALU y el desplazador ocurre durante el subciclo 3.

Después de un intervalo adicional, Δz, la salida del desplazador ha llegado a los registros por el bus C. Luego los registros pueden cargarse cerca del final del ciclo (en el flanco ascendente de la pulsación de reloj de la figura 4-3). El subciclo 4 consiste en el cargado de los registros y los flip-flops N y Z; termina un poco después del flanco ascendente del reloj, cuando todos los resultados se han guardado y los resultados de las operaciones de memoria anteriores están disponibles, y se ha cargado MPC. Este proceso continúa hasta que alguien se aburre y apaga la máquina.

Al mismo tiempo que controla la trayectoria de datos, el microprograma tiene que determinar qué microinstrucción ejecutará a continuación, porque no se ejecutan en el orden en que aparecen en el almacén de control. El cálculo de la dirección de la siguiente microinstrucción se inicia una vez que MIR se ha cargado y estabilizado. Primero se copia el campo NEXT_ADDRESS de 9 bits en MPC. Mientras se está efectuando este copiado, se inspecciona el campo JAM. Si tiene el valor 000, no se hace nada más; una vez que concluye el copiado de NEXT_ADDRESS, MPC apunta a la siguiente microinstrucción.

Si uno o más de los bits de JAM son 1, hay que trabajar más. Si JAMN está activado, el flip-flop N de un bit se hace OR con el bit de orden alto de MPC. Del mismo modo, si JAMZ está establecido, el flip-flop Z de un bit realiza la operación OR. Si ambos están establecidos, se realiza la operación OR de ellos. Se necesitan los flip-flops N y Z porque después del flanco ascendente del reloj (mientras el reloj está alto), el B bus ya no está siendo controlado, y no puede suponerse que las salidas de la ALU sigan siendo correctas. Guardar las banderas de estado de la ALU en N y Z hace que los valores correctos se estabilicen y estén disponibles para el cálculo de MPC, sin importar qué esté ocurriendo en torno de la ALU.

En la figura 4-6 la lógica que efectúa este cálculo se denomina "Bit alto". La función booleana que calcula es

F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDRESS[8]

Observe que, en todos los casos, MPC puede adoptar sólo uno de dos posibles valores:

1. El valor de NEXT_ADDRESS.
2. El valor de NEXT_ADDRESS después de que se realizó el OR del bit de orden alto con 1.

No existen otras posibilidades. Si el bit de orden alto de NEXT_ADDRESS era 1, no tiene sentido usar JAMN o JAMZ.

Observe que cuando todos los bits de JAM son ceros, la dirección de la siguiente microinstrucción a ejecutar es simplemente el número de nueve bits que está en su campo NEXT_ADDRESS. Si JAMN o JAMZ son 1, hay dos posibles sucesores, NEXT_ADDRESS y el resultado de la operación OR de NEXT_ADDRESS con 0x100 (suponiendo que NEXT_ADDRESS ≤ 0xFF). (Cabe señalar que 0x indica que el número que sigue está en hexadecimal.) Este punto se ilustra en la figura 4-7. La microinstrucción en curso, en la posición 0x75, tiene NEXT_ADDRESS = 0x92 y JAMZ = 1. Por tanto, la siguiente dirección de la microinstrucción depende del bit Z almacenado en la operación de ALU previa. Si el bit Z es 0, la siguiente microinstrucción provendrá de 0x92. Si el bit Z es 1, la siguiente microinstrucción provendrá de 0x192.

El tercer bit del campo JAM es JMPC. Si está activo, se efectúa la operación OR bit por bit de los ocho bits de MBR con los 8 bits de orden bajo del campo NEXT_ADDRESS de la microinstrucción en curso. El resultado se envía a MPC. El cuadro rotulado "O" en la figura 4-6 efectúa un OR de MBR con NEXT_ADDRESS si JMPC es 1, pero se limita a pasar NEXT_ADDRESS directamente a MPC si JMPC es 0. Cuando JMPC es 1, los ocho bits de orden bajo de NEXT_ADDRESS normalmente son cero. El bit de orden alto puede ser 0 o 1, de modo que el valor de NEXT_ADDRESS que se usa con JMPC normalmente es 0x000 o 0x100. La razón para usar a veces 0x000 y a veces 0x100 se explicará más adelante.

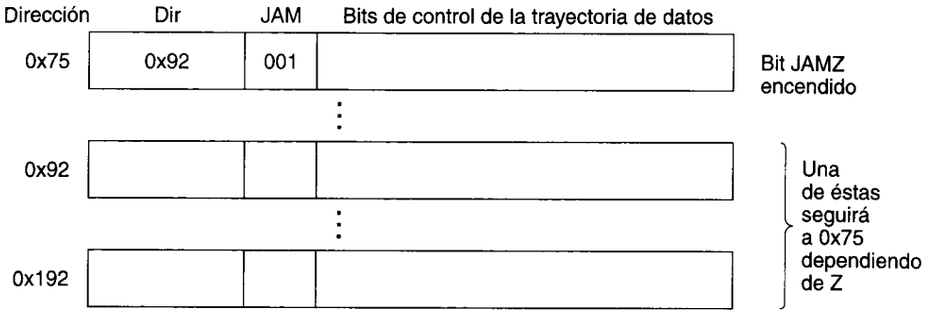


Figura 4-7. Una microinstrucción con JAMZ puesto en 1 tiene dos posibles sucesoras.

La capacidad para calcular el OR de MBR con NEXT_ADDRESS y almacenar el resultado en MPC hace posible una implementación eficiente de una ramificación (salto) de múltiples vías. Observe que es posible especificar cualquiera de 256 direcciones, determinada exclusivamente por los bits presentes en MBR. En un uso típico, MBR contiene un código de operación, de modo que el uso de JMPC hará que se seleccione para ejecutarse en seguida una microinstrucción única para cada posible código de operación. Este método es útil para saltar directamente a la función que corresponde al código de operación encontrado.

Para la explicación que sigue es crucial entender la temporización de la máquina, así que tal vez valga la pena repetirla. Lo haremos en términos de subciclos, ya que esto es más fácil de visualizar, los sucesos de reloj reales son el flanco descendente, que inicia el ciclo, y el flanco ascendente, que carga los registros y los flip-flops N y Z. Remítase por favor a la figura 4-3.

Durante el subciclo 1, iniciado por el flanco descendente del reloj, MIR se carga con la dirección que actualmente está en MPC. Durante el subciclo 2, las señales de MIR se propagan hacia afuera y el bus B se carga a partir del registro seleccionado. Durante el subciclo 3, la ALU y el desplazador operan y producen un resultado estable. Durante el subciclo 4, el bus C, los buses de memoria y los valores de la ALU se estabilizan. En el flanco ascendente del reloj los registros se cargan a partir del bus C, los flip-flops N y Z se cargan, y MBR y MDR obtienen sus resultados de la operación de memoria que se inició al final del ciclo de trayectoria de datos anterior (si se inició alguna de esas operaciones). Tan pronto como está disponible MBR, MPC se carga en preparación para la siguiente microinstrucción. Así MPC obtiene su valor en algún momento durante la parte media del intervalo en el que el reloj está alto pero después de que MBR/MDR están listos. Esto podría ser disparado por nivel (en lugar de por flanco) o disparado por flanco con un retraso fijo después del flanco ascendente del reloj. Lo único que importa es que MPC no se cargue sino hasta que los registros de los que depende (MBR, N y Z) estén listos. Tan pronto como el reloj baja, MPC puede direccionar el almacén de control y se puede iniciar un ciclo nuevo.

Cabe señalar que cada ciclo es autosuficiente: especifica qué se coloca en el bus B, qué deben hacer la ALU y el desplazador, dónde debe almacenarse el valor del bus C y, por último, qué valor debe tener MPC a continuación.

Vale la pena un último comentario acerca de la figura 4-6. Hemos estado tratando a MPC como un registro normal, con 9 bits de capacidad, que se carga mientras el reloj está alto. En realidad, no hay necesidad de tener un registro ahí. Todas sus entradas se pueden almacenar directamente hasta el almacén de control. En tanto estén presentes en el almacén de control en el flanco descendente del reloj cuando MIR se selecciona y lee, será suficiente. No hay necesidad de almacenarlas realmente en MPC. Por esta razón, MPC bien podría implementarse como un **registro virtual**, que no es más que un punto de reunión de señales, más parecido a un panel de conexiones electrónico que a un registro real. Hacer que MPC sea un registro virtual simplifica la temporización: ahora sólo ocurren sucesos en los flancos descendentes y ascendentes del reloj, y en ningún otro momento. Pero si para usted es más fácil pensar en MPC como un registro real, también es un punto de vista válido.

4.2 UN EJEMPLO DE ISA: IJVM

Continuemos nuestro ejemplo presentando el nivel ISA de la máquina para ser interpretada por el microprograma que se ejecuta en la microarquitectura de la figura 4-6 (IJVM). Por comodidad, algunas veces nos referiremos a la arquitectura del conjunto de instrucciones (ISA) como la **macroarquitectura**, a fin de contrastarla con la microarquitectura. Sin embargo, antes de describir IJVM haremos una breve digresión para motivar la descripción.

4.2.1 Pilas

Prácticamente todos los lenguajes de programación soportan el concepto de procedimientos (métodos) que tienen variables locales. Estas variables pueden accesarse desde el interior del procedimiento pero dejan de ser accesibles una vez que el procedimiento ha regresado. Así, surge la pregunta: "¿en qué lugar de la memoria deben guardarse estas variables?"

La solución más sencilla, asignar a cada variable una dirección de memoria absoluta, no funciona. El problema es que un procedimiento puede llamarse a sí mismo. Estudiaremos estos procedimientos recursivos en el capítulo 5. Por el momento, basta decir que si un procedimiento se activa (es decir, se llama) dos veces, es imposible almacenar sus variables en posiciones de memoria absolutas porque la segunda invocación interferirá la primera.

En vez de ello, se usa una estrategia distinta. Se reserva un área de memoria, llamada **pila**, para variables, pero las variables individuales no reciben direcciones absolutas dentro de la pila. En vez de ello se ajusta un registro, digamos LV (*Local Variable*) de modo que apunte a la base de las variables locales del procedimiento en curso. En la figura 4-8(a) se ha invocado un procedimiento *A*, que tiene las variables locales *a1*, *a2* y *a3*, así que se ha reservado memoria para sus variables locales a partir de la posición a la que LV apunta. Otro registro, SP (*Stack Pointer*, apuntador de la pila), apunta a la palabra más alta de las variables locales de *A*. Si LV es 100 y las palabras son de 4 bytes, entonces SP será 108. Se hace referencia a las variables especificando a qué distancia están de LV. La estructura de datos entre LV y SP (que incluye las dos palabras a las que se apunta) es el **marco de variables locales** de *A*.

Consideremos ahora qué sucede si *A* llama a otro procedimiento, *B*. ¿Dónde deberán almacenarse las cuatro variables locales de *B* (*b1, b2, b3, b4*)? Respuesta: en la pila, encima de las de *A*, como se muestra en la figura 4-8(b). Observe que la llamada de procedimiento

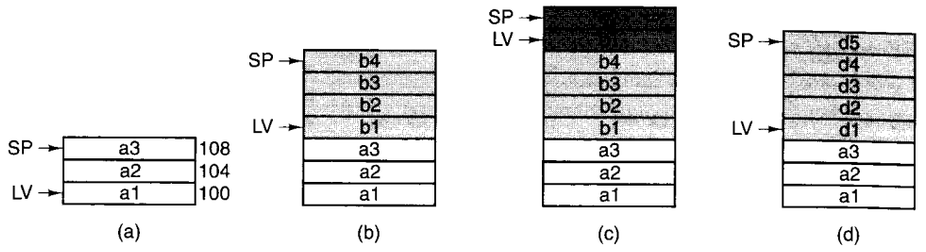


Figura 4-8. Uso de una pila para almacenar variables locales. (a) Mientras *A* está activo. (b) Después de que *A* llama a *B*. (c) Después de que *B* llama a *C*. (d) Después de que *C* y *B* regresan y *A* llama a *D*.

ajustó LV de modo que apunte a las variables locales de *B* en lugar de a las de *A*. Podemos hacer referencia a las variables locales de *B* dando su distancia respecto a LV. De igual manera, si *B* llama a *C*, LV y SP se ajustan nuevamente para asignar espacio a las dos variables locales de *C*, como se ilustra en la fig. 4-8(c).

Cuando *C* regresa, *B* se activa nuevamente, y la pila se vuelve a ajustar como en la figura 4-8(b) de modo que ahora LV apunte otra vez a las variables locales de *B*. De igual modo, cuando *B* regresa, volvemos a la situación de la figura 4-8(a). En todas las condiciones, LV apunta a la base del marco de pila del procedimiento que está activo, y SP apunta al tope del marco de pila.

Supongamos ahora que *A* llama a *D*, que tiene cinco variables locales. Llegamos a la situación de la figura 4-8(d), en la que las variables locales de *D* usan la misma memoria que usaron las de *B*, y también parte de las de *C*. Con esta organización de memoria, sólo se asigna espacio a los procedimientos que están activos. Cuando un procedimiento regresa, se libera la memoria que usaban sus variables locales.

Las pilas tienen otro uso, además de contener variables locales: pueden servir para retener operandos durante el cálculo de una expresión aritmética. Cuando se usa de este modo, la pila se llama **pila de operandos**. Suponga, por ejemplo, que antes de llamar a *B*, *A* tiene que efectuar el cálculo

```
a1 = a2 + a3;
```

Una forma de efectuar esta suma es meter *a2* en la pila, como se muestra en la figura 4-9(a). Aquí SP se incrementó tantos bytes como hay en una palabra, digamos 4, y el primer operando se almacenó en la dirección a la que ahora apunta SP. Luego se mete *a3* en la pila, como se muestra en la figura 4-9(b). Como acotación acerca de la notación, usaremos el tipo de letra Helvética para todos los fragmentos de programa como el anterior. También usaremos este tipo de letra para los códigos de operación de lenguaje ensamblador y los registros de la máquina, pero en el texto corrido las variables y procedimientos de los programas se darán en *cursivas*. La diferencia es que los nombres de variables y procedimientos son elegidos por el usuario; los códigos de operación y los nombres de registros están fijos.

Ahora puede efectuarse el cálculo ejecutando una instrucción que obtiene dos palabras de la pila, las suma y mete el resultado otra vez a la pila, como se muestra en la figura 4-9(c). Por último, saca la variable que está en la localidad más alta de la pila y se almacena en la variable local *a1*, como se ilustra en la figura 4-9(d).

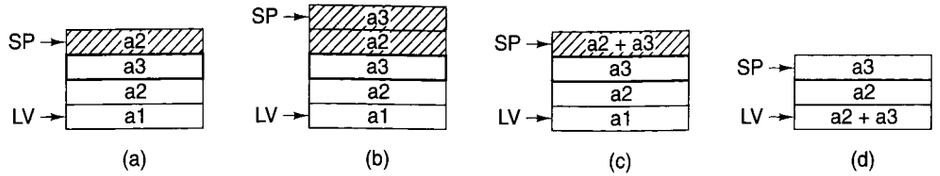


Figura 4-9. Uso de una pila de operandos para realizar un cálculo aritmético.

Los marcos de variables locales y las pilas de operandos se pueden entremezclar. Por ejemplo, al calcular una expresión como $x^2 + f(x)$, parte de la expresión (digamos x^2) podría estar en la pila de operandos cuando se invoca una función f. El resultado de la función se deja en la pila, encima de x^2, y la siguiente instrucción los suma.

Vale la pena señalar que si bien todas las máquinas usan una pila para almacenar variables locales, no todas usan una pila de operandos como ésta para efectuar operaciones aritméticas. De hecho, pocas lo hacen, pero JVM e IJVM trabajan así, y es por ello que hemos presentado esta introducción a las operaciones de pila. Las estudiaremos con mayor detalle en el capítulo 5.

4.2.2 El modelo de memoria IJVM

Ya estamos listos para examinar la arquitectura de la IJVM. Básicamente, consiste en una memoria que se puede ver de dos maneras: como una formación de 4,294,967,296 bytes (4 GB) o como una formación de 1,073,741,824 palabras, cada una de 4 bytes. A diferencia de la mayor parte de las ISA, la Máquina Virtual Java no permite que el nivel ISA vea directamente las direcciones, pero hay varias direcciones implícitas que sirven de base para un apuntador. Las instrucciones de la IJVM sólo pueden acceder a la memoria sumando índices a tales apuntadores. En cualquier momento están definidas las siguientes áreas de memoria:

1. *La reserva de constantes.* Un programa IJVM no puede escribir en esta área que consta de constantes, cadenas y apuntadores a otras áreas de memoria a las que se hace referencia. Se carga cuando el programa se almacena en la memoria y no se modifica posteriormente. Existe un registro implícito, CPP, que contiene la dirección de la primera palabra de la reserva de constantes.

2. *El marco de variables locales.* Para cada invocación de un método, se asigna un área para almacenar variables durante la vigencia de la invocación. Esta área se llama **marco de variables locales**. Al principio de este marco residen los parámetros (también llamados argumentos) con que se invocó el método. El marco de variables locales no incluye la pila de operandos, que es aparte. Sin embargo, por razones de eficiencia, nuestra implementación coloca la pila de operandos inmediatamente arriba del marco de variables locales. Existe un registro implícito que contiene la dirección de la primera posición del marco de variables locales. Llamaremos a este registro LV. Los parámetros pasan en el momento de invocarse el método y se almacenan al principio del marco de variables locales.

3. *La pila de operandos.* Se garantiza que el marco de pila no excederá cierto tamaño, calculado con anticipación por el compilador de Java. El espacio de la pila de operandos se asigna directamente arriba del marco de variables locales, como se ilustra en la figura 4-10. En nuestra implementación es conveniente pensar en la pila de operandos como parte del marco de variables locales. En todo caso, existe un registro implícito que contiene la dirección de la palabra superior de la pila. Observe que, a diferencia de CPP y LV, este apuntador, SP, cambia durante la ejecución del método a medida que se meten operandos en la pila o se sacan de ella.

4. *El área de métodos.* Por último, hay una región de la memoria que contiene el programa y que se llama área de "texto" en el caso de un proceso UNIX. Existe un registro implícito que contiene la dirección de la instrucción que se traerá a continuación. Este apuntador es el Contador de Programa o PC. A diferencia de las demás regiones de memoria, el área de métodos se trata como una formación de bytes.

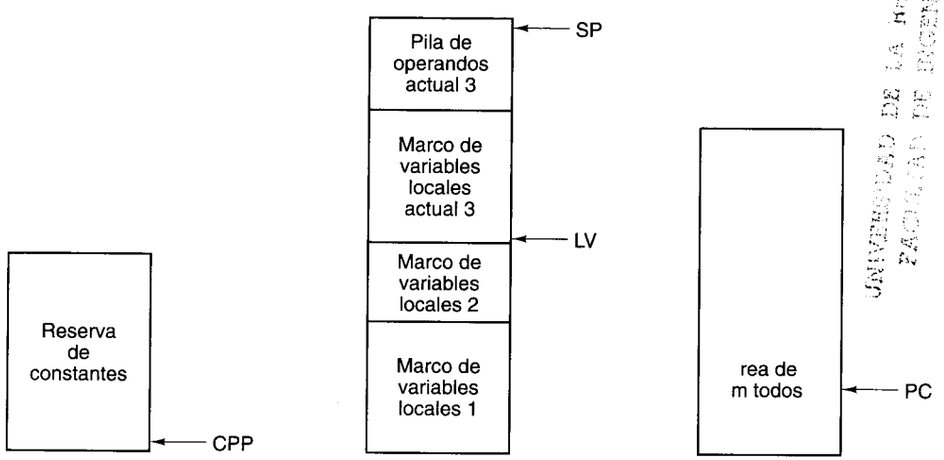


Figura 4-10. Las distintas partes de la memoria IJVM.

Debemos dejar en claro algo acerca de los apuntadores. Los registros CPP, LV y SP son apuntadores a *palabras*, no *bytes*, y las distancias se miden en palabras. Para el subconjunto entero que hemos escogido, todas las referencias a elementos de la reserva de constantes, el marco de variables locales y la pila son palabras, y todas las distancias empleadas como índices dentro de estos marcos se dan en palabras. Por ejemplo, LV, LV + 1 y LV + 2 se refieren a las primeras tres palabras del marco de variables locales. En contraste, LV, LV + 4 y LV + 8 se refieren a palabras situadas a intervalos de cuatro palabras (16 bytes).

En contraste, PC contiene una dirección de byte, y una resta o suma a PC altera la dirección en cierto número de bytes, no de palabras. El direccionamiento de PC es distinto

del de los demás registros, y esto se pone de manifiesto al haber un puerto de memoria especial en Mic-1 para PC. Recuerde que el puerto sólo tiene un byte de anchura. Incrementar PC en 1 e iniciar una lectura hace que se traiga el siguiente *byte*. Incrementar SP en 1 e iniciar una lectura hace que se traiga la siguiente *palabra*.

4.2.3 El conjunto de instrucciones IJVM

El conjunto de instrucciones de IJVM se muestra en la figura 4-11. Cada instrucción consiste en un código de operación y a veces un operando, como una distancia en la memoria o una constante. La primera columna da la codificación hexadecimal de la instrucción. La segunda da su representación mnemotécnica en lenguaje ensamblador. La tercera da una breve descripción de su efecto.

Hex	Mnemónico	Significado
0x10	BIPUSH *byte*	Almacenar byte en la pila
0x59	DUP	Copiar la palabra en el tope de la pila y almacenarla en la pila
0xA7	GOTO *distancia*	Ramificación incondicional
0x60	IADD	Sacar dos palabras de la pila; meter su suma
0x7E	IAND	Sacar dos palabras de la pila; meter su AND booleano
0x99	IFEQ *distancia*	Sacar palabra de la pila y saltar si es cero
0x9B	IFLT *distancia*	Sacar palabra de la pila y saltar si es menor que cero
0x9F	IF_ICMPEQ *distancia*	Sacar dos palabras de la pila; saltar si son iguales
0x84	IINC *númvar const*	Sumar una constante a una variable local
0x15	ILOAD *númvar*	Almacenar variable local en la pila
0xB6	INVOKEVIRTUAL *despl*	Invocar un método
0x80	IOR	Sacar dos palabras de la pila; almacenar su OR booleano
0xAC	IRETURN	Regresar de método con valor entero
0x36	ISTORE *númvar*	Sacar palabra de la pila y guardar en variable local
0x64	ISUB	Sacar dos palabras de la pila; almacenar su diferencia
0x13	LDC_W *índice*	Meter en la pila constante de la reserva de constantes
0x00	NOP	No hacer nada
0x57	POP	Quitar palabra del tope de la pila
0x5F	SWAP	Intercambiar las dos palabras en el tope de la pila
0xC4	WIDE	Prefijo de instrucción; la siguiente instrucción tiene un índice de 16 bits

Figura 4-11. El conjunto de instrucciones de IJVM. Los operandos *byte*, *const* y *númvar* son de 1 byte. Los operandos *despl*, *índice* y *distancia* son de 2 bytes.

Se proporcionan instrucciones para meter en la pila una palabra procedente de varios orígenes. Éstos incluyen la reserva de constantes (LDC_W), el marco de variables locales (ILOAD) y la instrucción misma (BIPUSH). También se puede sacar una variable de la pila y guardarse en el marco de variables locales (ISTORE). Pueden efectuarse dos operaciones aritméticas (IADD e ISUB) así como dos operaciones lógicas (booleanas) (IAND y IOR) utilizando como operandos las dos palabras que están en el tope de la pila. En todas las operaciones aritméticas y lógicas se obtienen dos palabras de la pila y el resultado se almacena en la pila. Se proporcionan cuatro instrucciones de ramificación, una incondicional (GOTO) y tres condicionales (IFEQ, IFLT e IF_ICMPEQ). Todas las instrucciones de ramificación, si se siguen, ajustan el valor de PC según la distancia (de 16 bits con signo) que sigue al código de operación en la instrucción. Esta distancia se suma a la dirección del código de operación. También hay instrucciones IJVM para intercambiar las dos palabras que están en el tope de la pila (SWAP), duplicar la palabra que está en el tope (DUP) y sacarla (POP).

Algunas instrucciones tienen varios formatos, que incluyen una forma corta para las de uso común. Hemos incluido en IJVM dos de los diversos mecanismos que la JVM usa para implementar esto. En un caso hemos pasado por alto la forma corta en favor de la más general. En otro caso mostramos cómo se puede usar la instrucción prefijo WIDE para modificar la instrucción subsecuente.

Por último, hay una instrucción (INVOKEVIRTUAL) para invocar otro método, y otra instrucción (IRETURN) para salir de un método y devolver el control al método que lo invocó. Debido a la complejidad del mecanismo, hemos simplificado un poco la definición, y esto hizo posible crear un mecanismo sencillo para hacer una llamada y regresar. La restricción es que, a diferencia de Java, sólo permitimos a un método invocar un método que exista dentro de su propio objeto. Esta restricción obstaculiza severamente la orientación a objetos pero nos permite presentar un mecanismo mucho más sencillo, al evitar la necesidad de localizar el método dinámicamente. (Si no está familiarizado con la programación orientada a objetos, puede hacer caso omiso de este comentario sin problema. Lo que hemos hecho es convertir a Java en un lenguaje no orientado a objetos, como C o Pascal.) En todas las computadoras excepto JVM, la dirección de procedimiento al que se llamará la determina directamente la instrucción CALL, así que nuestro enfoque en realidad es el caso normal, no la excepción.

El mecanismo para invocar un método es el siguiente. Primero, el invocador mete en la pila una referencia (apuntador) al objeto que se llamará. (Esta referencia no se necesita en una IJVM porque no se puede especificar otro objeto, pero se conserva para mantener la consistencia con JVM.) En la figura 4-12(a) esta referencia se indica con OBJREF. Luego, el invocador mete los parámetros del método en la pila, en este ejemplo, *Parámetro 1*, *Parámetro 2* y *Parámetro 3*. Por último, se ejecuta INVOKEVIRTUAL.

La instrucción INVOKEVIRTUAL incluye un desplazamiento que indica la posición en la reserva de constantes que contiene la dirección, dentro del área de métodos, donde inicia el método que se está invocando. Sin embargo, si bien el código del método reside en la posición a la que apunta este apuntador, los primeros cuatro bytes en el área de métodos contienen datos especiales. Los primeros dos bytes se interpretan como un entero de 16 bits que indica el número de parámetros del método (los parámetros se almacenaron previamente en la pila). En lo referente a esta cifra, OBJREF cuenta como parámetro: el parámetro 0. Este entero de 16 bits, junto con el valor de SP, da la ubicación de OBJREF. Observe que LV

apunta a OBJREF y no al primer parámetro real. La decisión de a dónde apunta LV es un tanto arbitraria.

Los siguientes dos bytes del área de método se interpretan como otro entero de 16 bits que indica el tamaño del área de variables locales para el método que se está invocando. Esto es necesario porque se establecerá una nueva pila para el método, que comenzará inmediatamente arriba del marco de variables locales. Por último, el quinto byte del área de métodos contiene el primer código de operación a ejecutar.

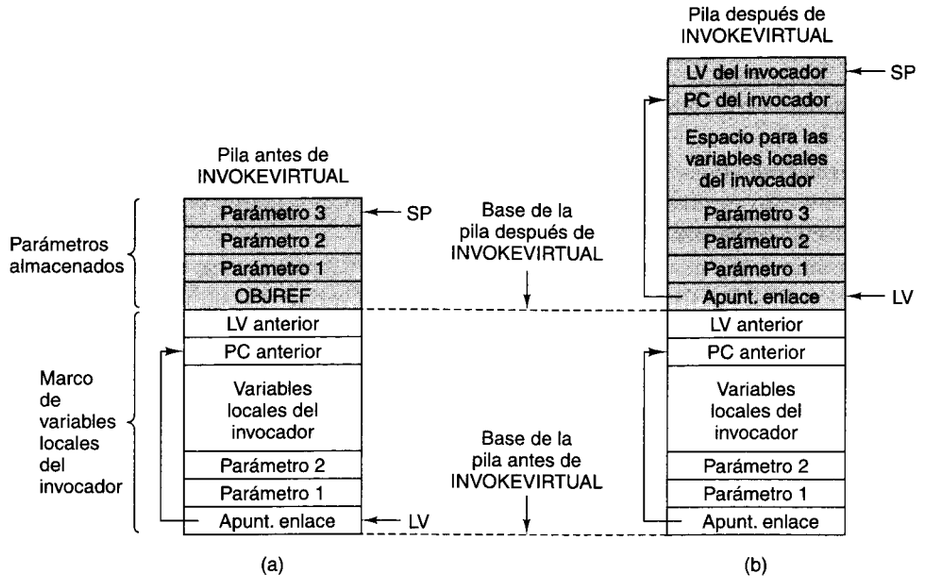


Figura 4-12. (a) Memoria antes de ejecutar INVOKEVIRTUAL. (b) Después de ejecutarlo.

La secuencia real que ocurre con INVOKEVIRTUAL es la que sigue y se muestra en la figura 4-12. Los dos bytes de índice sin signo que siguen al código de operación sirven para construir un índice para la tabla de la reserva de constantes (el primer byte es el byte de orden alto). La instrucción calcula la dirección base del nuevo marco de variables locales restando el número de parámetros al apuntador de pila y ajustando LV de modo que apunte a OBJREF. En este lugar, sobreescribiendo OBJREF, la implementación almacena la dirección de la posición donde se almacenará el PC antiguo. Esta dirección se calcula sumando el tamaño del marco de variables locales (parámetros + variables locales) a la dirección contenida en LV. Inmediatamente arriba de la dirección donde se guardará el PC viejo está la dirección donde se guardará el LV viejo. Inmediatamente arriba de esa dirección está el principio de la pila del procedimiento recién llamado. Se ajusta SP de modo que apunte al viejo LV, que es la dirección inmediatamente abajo de la primera posición vacía de la pila. Recuerde que SP siempre apunta a la palabra que está en el tope de la pila. Si la pila está vacía, SP apuntará a

la primera posición debajo del final de la pila porque nuestras pilas crecen hacia arriba, hacia direcciones más altas. En nuestras figuras las pilas siempre crecen hacia arriba, hacia la dirección más alta en la parte superior de la página.

La última operación que se requiere para ejecutar INVOKEVIRTUAL es hacer que PC apunte al quinto byte del espacio de código de métodos.

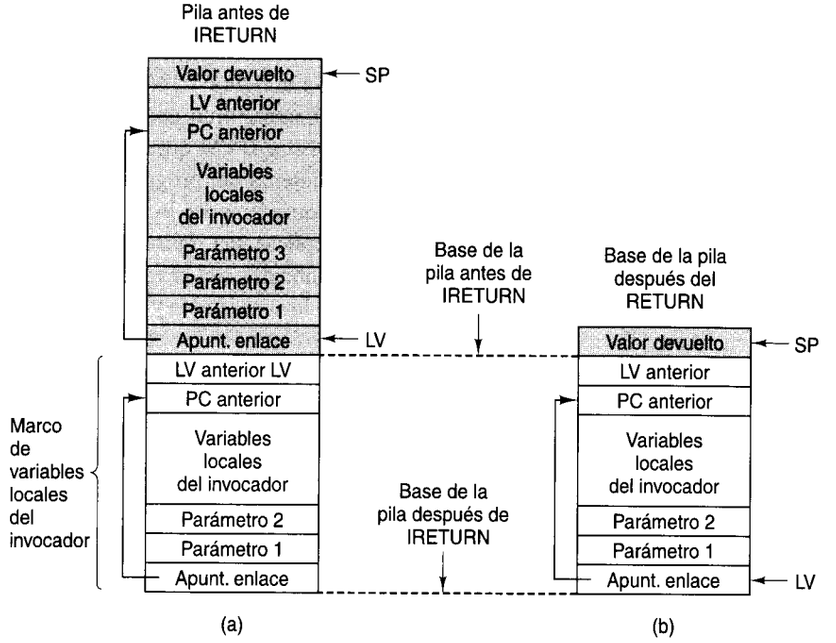


Figura 4-13. (a) Memoria antes de ejecutar IRETURN. (b) Después de ejecutarlo.

La instrucción IRETURN invierte las operaciones de la instrucción INVOKEVIRTUAL, como se muestra en la figura 4-13. Libera el espacio utilizado por el método que regresa y también restaura la pila a su estado anterior, excepto que (1) la palabra OBJREF (que ya se sobreescribió) y todos los parámetros se sacaron de la pila, y (2) el valor devuelto se colocó en el tope de la pila, en la posición que antes ocupaba OBJREF. Para restaurar el estado antiguo, la instrucción IRETURN debe restaurar los valores antiguos de PC y LV. Esto lo hace accediendo al apuntador de enlace (que es la palabra identificada por el apuntador LV actual). Recuerde que en esta posición, donde originalmente estaba almacenado OBJREF, la instrucción INVOKEVIRTUAL guardó la dirección que contiene el antiguo PC. Esta palabra y la que está arriba de ella se recuperan para restaurar PC y LV, respectivamente, a sus antiguos valores. El valor devuelto, que se guarda en el tope de la pila del método que termina, se copia en la posición en que se había guardado originalmente OBJREF, y SP se restaura de modo que apunte a esta posición. Así, el control se devuelve a la instrucción que sigue inmediatamente a la instrucción INVOKEVIRTUAL.

Hasta aquí, nuestra máquina no tiene instrucciones de entrada/salida. Y no las vamos a agregar, pues las necesita tanto como las necesita la Máquina Virtual Java, en cuya especificación oficial ni siquiera se menciona E/S. La teoría es que una máquina que no efectúa entrada ni salida es "segura". (La lectura y escritura se efectúan en JVM mediante llamadas a métodos especiales, que realizan la E/S.)

4.2.4 Compilación de Java a IJVM

Veamos ahora la relación que hay entre Java y la IJVM. En la figura 4-14(a) mostramos un sencillo fragmento de código en Java. Cuando éste se alimente a un compilador Java, el compilador probablemente producirá el lenguaje ensamblador IJVM que se muestra en la figura 4-14(b). Los números de línea del 1 al 15 a la izquierda del programa en lenguaje ensamblador no forman parte de la salida del compilador, como tampoco lo hacen los comentarios (que comienzan con //); están ahí para ayudar a explicar una figura subsecuente. El ensamblador de Java traduciría entonces el programa ensamblado en el programa binario que se muestra en la figura 4-14(c). (En realidad, el compilador Java ensambla y genera el programa binario directamente.) Para este ejemplo hemos supuesto que *i* es la variable local 1, *j* es la variable local 2 y *k* es la variable local 3.

```
i - j + k;                1     ILOAD j          // i = j + k      0x15 0x02
if (i == 3)               2     ILOAD k                         0x15 0x03
    k = 0;                3     IADD                            0x60
else                      4     ISTORE i                        0x36 0x01
    j = j - 1;            5     ILOAD i          // if (i < 3)    0x15 0x01
                          6     BIPUSH 3                        0x10 0x03
                          7     IF_ICMPEQ L1                    0x9F 0x00 0x0D
                          8     ILOAD j          // j - j - 1     0x15 0x02
                          9     BIPUSH 1                        0x10 0x01
                         10     ISUB                            0x64
                         11     ISTORE j                        0x36 0x02
                         12     GOTO L2                         0xA7 0x00 0x07
                         13 L1: BIPUSH 0         // k = 0         0x10 0x00
                         14     ISTORE k                        0x36 0x03
                         15 L2:
```

(a) (b) (c)

Figura 4-14. (a) Fragmento en Java. (b) El lenguaje de ensamblador Java correspondiente. (c) El programa IJVM en hexadecimal.

El código compilado es sencillo. Primero *j* y *k* se meten en la pila, se suman y el resultado se guarda en *i*. Luego *i* y la constante 3 se meten en la pila y se comparan. Si son iguales, se sigue una ramificación a *L1*, donde *k* se pone en 0. Si son distintas, la comparación falla y se ejecuta el código que sigue a IF_ICMPEQ. Cuando el programa termina de hacer esto, salta a *L2*, donde se fusionan las partes then y else.

La pila de operandos para el programa IJVM de la figura 4-14(b) se muestra en la figura 4-15. Antes de que el código comience a ejecutarse, la pila está vacía, lo que se indica con la línea horizontal arriba del 0. Después del primer ILOAD, *j* está en la pila, como indica la *j* en un rectángulo arriba del 1 (que indica que ya se ejecutó la instrucción 1). Después del segundo ILOAD hay dos palabras en la pila, como se muestra arriba del 2. Después del IADD sólo queda una palabra en la pila, y contiene la suma *j* + *k*. Cuando se saca la palabra del tope de la pila y se guarda en *i*, la pila queda vacía, como se muestra arriba del 4.

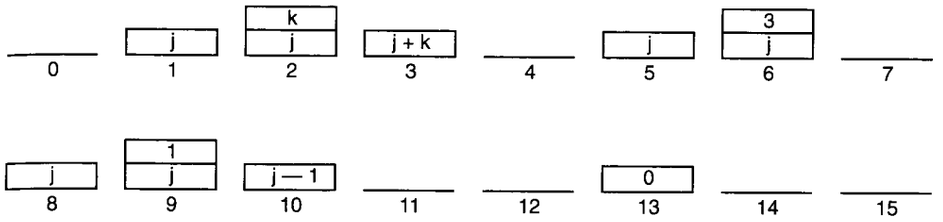


Figura 4-15. La pila después de cada instrucción de la figura 4-14(b).

La instrucción 5 (ILOAD) inicia el enunciado if metiendo *i* en la pila (en 5). A continuación viene la constante 3 (en 6). Después de la comparación, la pila queda vacía otra vez (7). La instrucción 8 es el principio de la parte else del fragmento de programa Java. La parte else continúa hasta la instrucción 12, donde se salta la parte then y llega al rótulo *L2*.

4.3 UN EJEMPLO DE IMPLEMENTACIÓN

Ahora que hemos especificado tanto la microarquitectura como la macroarquitectura con detalle, el aspecto que nos falta es la implementación. En otras palabras, ¿qué aspecto tiene un programa que se ejecuta en la primera y que interpreta la segunda, y cómo funciona? Antes de poder contestar estas preguntas, debemos considerar con detenimiento la notación que usaremos para describir la implementación.

4.3.1 Microinstrucciones y notación

En principio, podríamos describir el almacén de control en binario, 36 bits por palabra. Sin embargo, al igual que en los lenguajes de programación convencionales, es muy provechoso introducir una notación que comunique la esencia de los aspectos que necesitamos tratar y al mismo tiempo oculte los detalles que carecen de importancia o que se manejan mejor automáticamente. Es importante darse cuenta aquí de que el lenguaje que hemos escogido pretende ilustrar los conceptos más que facilitar diseños eficientes. Si esto último fuera nuestra meta, usaríamos una notación muy diferente para maximizar la flexibilidad de la que dispone el diseñador. Un aspecto en el que esta cuestión es importante es cómo escoger las

direcciones. Puesto que la memoria no está ordenada lógicamente, no existe una "siguiente instrucción" natural que esté implícita cuando especificamos una secuencia de operaciones. Gran parte de la potencialidad de esta organización de control proviene de la capacidad del diseñador (o del ensamblador) para seleccionar direcciones de forma eficiente. Por tanto, comenzaremos por introducir un lenguaje simbólico sencillo que describe cabalmente cada operación sin explicar plenamente cómo podrían haberse determinado todas las direcciones.

Nuestra notación especifica todas las actividades que ocurren en un solo ciclo de reloj en una sola línea. En teoría, podríamos usar un lenguaje de alto nivel para describir las operaciones. Sin embargo, el control ciclo por ciclo es muy importante porque ofrece la oportunidad de realizar varias operaciones de forma concurrente y es necesario poder analizar cada ciclo para entender y verificar las operaciones. Si la meta es una implementación rápida y eficiente (si los demás factores son iguales, rápido y eficiente siempre es mejor que lento e ineficiente), entonces cada ciclo cuenta. En una implementación real, el programa oculta muchos trucos sutiles, y se emplean secuencias u operaciones poco claras con objeto de ahorrar un solo ciclo. La recompensa por ahorrar ciclos es atractiva: si podemos reducir en dos ciclos una instrucción que tarda cuatro ciclos, se ejecutará en la mitad del tiempo, y esta aceleración ocurre cada vez que ejecutamos la instrucción.

Una posible estrategia sería simplemente hacer una lista de las señales que deben activarse en cada ciclo del reloj. Supongamos que en un ciclo queremos incrementar el valor de SP. También queremos iniciar una operación de lectura y queremos que la siguiente instrucción sea la que reside en la posición 122 del almacén de control. Podríamos escribir

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

donde WSP significa "escribir el registro SP". Esta notación es completa, pero difícil de entender. En vez de ello, combinaremos las operaciones de una forma natural e intuitiva para capturar el efecto de lo que está sucediendo:

```
SP = SP + 1; rd
```

Llamemos a nuestro microlenguaje ensamblador "MAL" (*Micro Assembly Language*) (que es como usted se va a sentir si escribe demasiado código en él). MAL está hecho a propósito para que refleje las características de la microarquitectura. Durante cada ciclo se puede escribir cualquiera de los registros, aunque normalmente sólo se escribe uno. Sólo se puede colocar el contenido de un registro en el lado B de la ALU. En el lado A las opciones son +1, 0, –1 y el registro H. Así, podemos usar un sencillo enunciado de asignación, como en Java, para indicar la operación a ejecutar. Por ejemplo, si queremos copiar algo de SP a MDR, podemos decir

```
MDR = SP
```

Si queremos indicar el uso de las funciones de la ALU aparte de dejar pasar simplemente el contenido del bus B, podemos escribir, por ejemplo,

```
MDR = H + SP
```


que suma el contenido del registro H a SP y escribe el resultado en MDR. El operador + es conmutativo (lo que implica que el orden de los operandos no importa), así que el enunciado anterior también puede escribirse como

```
MDR = SP + H
```

y generar la misma microinstrucción de 36 bits, aunque en términos estrictos H deba ser el operando izquierdo de la ALU.

Debemos tener cuidado de usar sólo operaciones permitidas. Las operaciones permitidas más importantes se muestran en la figura 4-16, donde SOURCE (origen) puede ser cualquiera de MDR, PC, MBR, MBRU, SP, LV, CPP, TOS u OPC (MBRU se refiere a la versión sin signo de MBR). Todos estos registros pueden actuar como fuentes de la ALU en el bus B. De igual manera, DEST puede ser MAR, MDR, PC, SP, LV, CPP, TOS, OPC o H, los cuales son posibles destinos de la salida de la ALU en el bus C. Este formato es engañoso porque muchos enunciados aparentemente razonables no están permitidos. Por ejemplo,

```
MDR = SP + MDR
```

parece perfectamente razonable, pero no es posible ejecutarlo en la trayectoria de datos de la figura 4-6 en un solo ciclo. Esta restricción se debe a que en una suma (fuera del incremento o decremento) uno de los operandos debe ser el registro H. De igual modo,

```
H = H – MDR
```

podría ser útil, pero también es imposible porque el único origen posible de un sustraendo (el valor que se resta) es el registro H. Corresponde al ensamblador rechazar enunciados que parezcan válidos pero que en realidad están prohibidos.

Extenderemos la notación para permitir múltiples asignaciones empleando varios signos de igual. Por ejemplo, si queremos incrementar SP y guardar el nuevo valor en SP y también en MDR, podemos escribir

```
SP = MDR = SP + 1
```

Para indicar lecturas y escrituras de palabras de datos de 4 bytes en la memoria simplemente incluiremos rd (leer) y wr (escribir) en la microinstrucción. La obtención de un byte a través del puerto de un byte se indica con fetch (buscar). Pueden ocurrir asignaciones y operaciones de memoria en el mismo ciclo. Esto se indica escribiéndolas en la misma línea.

Para evitar confusiones, repetiremos que Mic-1 tiene dos formas de acceder a la memoria. Las lecturas y escrituras de palabras de datos de 4 bytes usan MAR/MDR y se indican en las microinstrucciones con rd y wr, respectivamente. Las lecturas de códigos de operación de un byte del flujo de instrucciones usan PC/MBR y se indican con fetch en las microinstrucciones. Ambos tipos de operaciones de memoria pueden efectuarse simultáneamente.

Sin embargo, el mismo registro no puede recibir un valor de la memoria y de la trayectoria de datos en el mismo ciclo. Considere el código

```
MAR = SP; rd
MDR = H
```

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = – H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = – 1

Figura 4-16. Todas las operaciones permitidas. Cualquiera de estas operaciones se puede extender añadiéndole "<<8" para desplazar el resultado un byte a la izquierda. Por ejemplo, una operación común es H = MBR << 8.

El efecto de la primera microinstrucción es asignar a MDR un valor de la memoria al final de la segunda microinstrucción. Sin embargo, la segunda microinstrucción también asigna un valor a MDR al mismo tiempo. Estas dos asignaciones están en conflicto y no se permiten porque el resultado no está definido.

Recuerde que cada microinstrucción debe proporcionar explícitamente la dirección de la siguiente microinstrucción que se ejecutará. No obstante, a menudo ocurre que sólo hay una microinstrucción que puede invocar cierta microinstrucción: la que está en la línea inmediatamente anterior a ésta. Para facilitar la tarea del microprogramador, el microensamblador normalmente asigna una dirección a cada microinstrucción (no necesariamente consecutivas en el almacén de control) y llena el campo NEXT_ADDRESS de modo que microinstrucciones escritas en líneas consecutivas se ejecuten una tras otra.

Sin embargo, hay ocasiones en que el microprogramador necesita efectuar un salto, sea incondicional o condicional. La notación para los saltos incondicionales es fácil:

goto *etiqueta*

y se puede incluir en cualquier microinstrucción para nombrar explícitamente a su sucesora. Por ejemplo, casi todas las secuencias de microinstrucciones terminan con un regreso a la primera instrucción del ciclo principal, de modo que la última instrucción de una secuencia semejante por lo regular incluye

goto Main1

Cabe señalar que la trayectoria de datos está disponible para operaciones normales aun durante una microinstrucción que contiene un goto. Después de todo, cada microinstrucción

contiene un campo NEXT_ADDRESS. Lo único que goto hace es ordenar al microensamblador que coloque ahí un valor específico en lugar de la dirección donde el microensamblador decidió colocar la microinstrucción de la siguiente línea. En principio, toda línea debería tener un enunciado goto sólo cuando es útil para el microprogramador; si la dirección objetivo es la siguiente línea, puede omitirse.

Necesitamos una notación distinta para los saltos condicionales. Recuerde que JAMN y JAMZ usan los bits N y Z. A veces es necesario probar un registro para ver si es cero, por ejemplo. Una forma de hacerlo sería pasar el contenido del registro por la ALU y almacenarlo de vuelta en el mismo registro. El enunciado

TOS = TOS

parece extraño, aunque hace lo que buscamos (establecer el flip-flop Z con base en TOS). Sin embargo, con objeto de que los microprogramas sean más elegantes, extenderemos MAL añadiendo dos registros imaginarios, N y Z a los cuales podemos hacer asignaciones. Por ejemplo,

Z = TOS

pasa el contenido de TOS por la ALU y establece los valores de los flip-flops Z y N, pero no almacena un valor en un registro. Lo que hacemos realmente al usar Z o N como destino es decirle al microensamblador que ponga en 0 todos los bits del campo C de la figura 4-5. La trayectoria de datos ejecuta un ciclo normal, con todas las operaciones normales permitidas, pero no se escribe en ningún registro. Observe que no importa si el destino es N o Z; la microinstrucción generada por el microensamblador es idéntica. A los programadores que escojan intencionalmente el destino "incorrecto" se les debería obligar a trabajar con una IBM PC original de 4.77 MHz durante una semana como castigo.

La sintaxis para indicarle al microensamblador que establezca el bit JAMZ es

si (Z) goto L1; else goto L2

Puesto que el hardware exige que estas dos direcciones sean idénticas en sus 8 bits de orden bajo, corresponde al microensamblador asignarles tales direcciones. Por otra parte, dado que *L2* puede estar en cualquier lugar dentro de las 256 primeras palabras del almacén de control, el microensamblador tiene total libertad para encontrar un par disponible.

Normalmente, estos dos enunciados se combinarán; por ejemplo,

Z = TOS; if (Z) goto L1; else goto L2

El efecto de este enunciado es que MAL genera una microinstrucción en la que TOS se hace pasar por la ALU (pero no se almacena en ningún lado) para que su valor establezca el bit Z. Poco después de que Z se ha cargado con el valor del bit de condición de la ALU, se hace OR con el bit de orden alto de MPC, lo que obliga a traer la dirección de la siguiente microinstrucción de *L2* o bien de *L1* (que debe ser exactamente 256 más que *L2*). MPC estará estable y listo para usarse en la obtención de la siguiente microinstrucción.

Por último, necesitamos una notación para usar el bit JMPC. La que usaremos es

goto (MBR OR *valor*)

Esta sintaxis dice al microensamblador que use *valor* para NEXT_ADDRESS y ajuste el bit JMPC de modo que el resultado de MBR OR NEXT_ADDRESS se coloque en MPC. Si *valor* es 0, como sucede normalmente, basta con escribir

```
goto (MBR)
```

Observe que sólo los 8 bits de orden bajo de MBR están conectados permanentemente con MPC (vea la figura 4-6), por lo que en este caso no surge la cuestión de la extensión de signo (es decir, MBR versus MBRU). Observe también que el MBR que está disponible al final del ciclo actual es el mismo que se usó. Una obtención iniciada en *esta* microinstrucción llega demasiado tarde para afectar la selección de la siguiente microinstrucción.

4.3.2 Implementación de IJVM utilizando el Mic-1

Por fin hemos llegado a un punto en el que podemos armar todas las piezas. La figura 4-17 es el microprograma que se ejecuta en Mic-1 e interpreta IJVM. Es un programa sorprendentemente corto: 112 microinstrucciones en total. Se dan tres columnas para cada microinstrucción: el rótulo simbólico, el microcódigo real y un comentario. Cabe señalar que microinstrucciones consecutivas no necesariamente se encuentran en direcciones consecutivas en el almacén de control, como ya mencionamos.

A estas alturas los nombres que escogimos para casi todos los registros de la figura 4-1 deberán tener un significado obvio: CPP (*Constant Pool Pointer*), LV (*Local Variables*) y SP (*Stack Pointer*) sirven para almacenar los apuntadores a la reserva de constantes, a las variables locales y al tope de la pila, respectivamente, mientras que PC (*Program Counter*) contiene la dirección del siguiente byte que se obtendrá del flujo de instrucciones. MBR es un registro de un byte que contiene los bytes del flujo de instrucciones que llegan de la memoria para interpretarse. TOS (*Top Of Stack*) y OPC (*OPCode*) son registros extra. Su uso se describe a continuación.

En ciertos momentos, se garantiza que cada uno de estos registros contiene cierto valor, pero todos pueden usarse como registros temporales si es necesario. Al principio y al final de cada instrucción, TOS contiene el valor que está en la posición de memoria a la que SP apunta, la palabra que está en el tope de la pila. Este valor es redundante porque siempre se puede leer de la memoria, pero tenerlo en un registro a menudo ahorra una referencia a la memoria. En el caso de unas cuantas instrucciones, mantener TOS implica *más* operaciones de memoria. Por ejemplo, la instrucción POP desecha la palabra que está en el tope de la pila, y por tanto debe traer de la memoria la nueva palabra que ahora está en el tope y colocarla en TOS.

El registro OPC es un registro temporal; no tiene un uso preasignado. Se usa, por ejemplo, para guardar la dirección del código de operación de una operación de ramificación mientras PC se incrementa para acceder a parámetros. OPC también se usa como registro temporal en las instrucciones de ramificación condicional de IJVM.

Al igual que todos los intérpretes, el microprograma de la figura 4-17 tiene un ciclo principal que busca, decodifica y ejecuta instrucciones del programa que se está interpretando, en este caso, instrucciones IJVM. Su ciclo principal comienza en la línea rotulada Main1, con la invariante de que PC ya se cargó previamente con la dirección de una posición de memoria que contiene un código de operación. Además, ese código de operación ya se colocó en MBR. Cabe señalar que esto implica que cuando regresemos a este punto deberemos

asegurarnos de que PC se haya actualizado de modo que apunte al siguiente código de operación a interpretar y que el byte del código de operación mismo ya se haya traído y se haya colocado en MBR.

Esta sucesión inicial de instrucciones se ejecuta al principio de cada instrucción, por lo que es importante que sea lo más corta posible. Gracias a un diseño muy cuidadoso del hardware y software del Mic-1 hemos logrado reducir el ciclo principal a una sola microinstrucción. Una vez que la máquina ha iniciado, cada vez que se ejecuta esta microinstrucción el código de operación IJVM que se va a ejecutar ya está presente en MBR. Lo que la microinstrucción hace es ramificar al microcódigo que ejecuta esta instrucción IJVM y también comenzar a traer el byte que sigue al código de operación, que podría ser un byte de operando o el siguiente código de operación.

Ahora podemos revelar la verdadera razón por la que cada microinstrucción nombra explícitamente a su sucesora, en lugar de dejar que se ejecuten en sucesión. Todas las direcciones del almacén de control que corresponden a códigos de operación deben reservarse para la primera palabra del intérprete de la instrucción correspondiente. Así, en la figura 4-11 vemos que el código que interpreta POP inicia en 0x57 y el código que interpreta DUP inicia en 0x59. (Cómo MAL supo que debía poner POP en 0x57 es uno de los misterios del universo; cabe suponer que en algún lugar hay un archivo que lo especifica.)

Lo malo es que el código de POP tiene una longitud de tres microinstrucciones, así que si lo colocáramos en palabras consecutivas interferiría el principio de DUP. Puesto que todas las direcciones del almacén de control que corresponden a códigos de operación están reservadas de hecho, las microinstrucciones que siguen a la primera de cada secuencia deben colocarse en los huecos que hay entre las direcciones reservadas. Por esta razón hay una ramificación constante, y tener una microrramificación explícita (una microinstrucción que ramifica) cada cierto número de instrucción para saltar de un hueco a otro implicaría un gran desperdicio.

Para ver cómo funciona el intérprete, supongamos, por ejemplo, que MBR contiene el valor 0x60, es decir, el código de operación de IADD (vea la figura 4-11). En el ciclo principal, que abarca una sola microinstrucción, realizamos tres cosas:

1. Incrementamos PC, de modo que ahora contiene la dirección del primer byte después del código de operación.
2. Iniciamos la búsqueda del siguiente byte para colocarlo en MBR. Este byte se necesitará tarde o temprano, sea como operando para la instrucción IJVM en curso o como siguiente código de operación (como en el caso de la instrucción IADD, que no tiene bytes de operandos).
3. Realizar una ramificación multivías a la dirección contenida en MBR al principio de Main1. Esta dirección es igual al valor numérico del código de operación que se está ejecutando. La microinstrucción anterior la colocó ahí. Tome nota de que el valor que se está obteniendo en esta microinstrucción no desempeña ningún papel en la ramificación multivías.

La obtención del siguiente byte se inicia aquí para que esté disponible cuando inicie la tercera microinstrucción. Podría necesitarse entonces o no, pero se usará tarde o temprano, por lo que iniciar la obtención ahora en nada podría perjudicar.

Mnemónico	Operaciones	Comentarios
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene código de operación; obtener siguiente byte; despachar
nop1	goto Main1	No hacer nada
iadd1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
iadd2	H = TOS	H = tope de la pila
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Sumar dos palabras del tope; escribir en tope de pila
isub1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
isub2	H = TOS	H = tope de la pila
isub3	MDR = TOS = MDR – H; wr; goto Main1	Restar; escribir en tope de pila
iand1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
iand2	H = TOS	H = tope de la pila
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Hacer AND; escribir en tope de pila
ior1	MAR = SP = SP + 1; rd	Leer 2a. palabra de la pila
ior2	H = TOS	H = tope de la pila
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Hacer OR; escribir en tope de pila
dup1	MAR = SP = SP + 1	Incrementar SP y copiar en MAR
dup2	MDR = TOS; wr; goto Main1	Escribir nueva palabra de pila
pop1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
pop2		Esperar que se lea nuevo TOS de la memoria
pop3	TOS = MDR; goto Main1	Copiar nueva palabra en TOS
swap1	MAR = SP – 1; rd	Asignar SP-1 a MAR; leer 2a. palabra de pila
swap2	MAR = SP	Asignar palabra en tope a MAR
swap3	H = MDR; wr	Guardar TOS en H; escribir 2a. pal. en tope de pila
swap4	MDR = TOS	Copiar TOS viejo en MDR
swap5	MAR = SP – 1; wr	Asignar SP-1 a MAR; escribir 2a. pal. en pila
swap6	TOS = H; goto Main1	Actualizar TOS
bipush1	SP = MAR = SP + 1	MBR = byte que se meterá en la pila
bipush2	PC = PC + 1; fetch	Incrementar PC, traer siguiente cód. op.
bipush3	MDR = TOS = MDR; wr; goto Main1	Extender signo de constante y meter en pila
iload1	H = LV	MBR contiene índice; copiar LV en H
iload2	MAR = MBRU + H; rd	MAR = dir. de variable local que se meterá
iload3	MAR = SP = SP + 1	SP apunta a nuevo tope de pila; preparar escrit.
iload4	PC = PC + 1; fetch; wr	Inc. PC; obt. sig. cód. op.; escribir en tope de pila
iload5	TOS = MDR; goto Main1	Actualizar TOS
istore1	H = LV	MBR contiene índice; copiar LV en H
istore2	MAR = MBRU + H	MAR = dir. de var. local en la que se guardará
istore3	MDR = TOS; wr	Copiar TOS en MDR; escribir palabra
istore4	SP = MAR = SP – 1; rd	Leer 2a. palabra de la pila
istore5	PC = PC + 1; fetch	Incrementar PC; traer siguiente cód. op.
istore6	TOS = MDR; goto Main1	Actualizar TOS
wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)	Ramif. multivías con bit alto encendido
wide_iload1	PC = PC + 1; fetch	MBR contiene 1er. byte índice; traer 2o.
wide_iload2	H = MBRU << 8	H = 1er. byte índice desplazado 8 bits a la izq.
wide_iload3	H = MBRU OR H	H = índice de 16 bits de variable local
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = dir. de variable local que se meterá
wide_istore1	PC = PC + 1; fetch	MBR contiene 1er. byte índice; traer 2o.
wide_istore2	H = MBRU << 8	H = 1er. byte índice desplazado 8 bits a la izq.
wide_istore3	H = MBRU OR H	H = índice de 16 bits de variable local
wide_istore4	MAR = LV + H; goto istore3	MAR = dir. de variable local en que se guardará
idc_w1	PC = PC + 1; fetch	MBR contiene 1er. byte índice; traer 2o.
idc_w2	H = MBRU << 8	H = 1er byte de índice << 8
idc_w3	H = MBRU OR H	H = índice de 16 bits en reserva de constantes
idc_w4	MAR = H + CPP; rd; goto iload3	MAR = dir. de la constante en la reserva

Figura 4-17. El microprograma del Mic-1 (parte 1 de 3).

Mnemónico	Operaciones	Comentarios
iinc1	H = LV	MBR contiene índice; copiar LV en H
iinc2	MAR = MBRU + H; rd	Copiar LV + índice en MAR; leer variable
iinc3	PC = PC + 1; fetch	Obtener constante
iinc4	H = MDR	Copiar variable en H
iinc5	PC = PC + 1; fetch	Obtener siguiente código de operación
iinc6	MDR = MBR + H; wr; goto Main1	Almacenar suma en MDR; actualizar variable
goto1	OPC = PC – 1	Guardar dirección de código de op.
goto2	PC = PC + 1; fetch	MBR = 1er. byte de distancia; traer 2o.
goto3	H = MBR << 8	Desplazar y guardar 1er. byte c/signo en H
goto4	H = MBRU OR H	H = distancia de salto de 16 bits
goto5	PC = OPC + H; fetch	Sumar distancia a OPC
goto6	goto Main1	Esperar que traigan siguiente cód. op.
ifft1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
ifft2	OPC = TOS	Guardar TOS en OPC temporalmente
ifft3	TOS = MDR	Poner nuevo tope de pila en TOS
ifft4	N = OPC; if (N) goto T; else goto F	Ramificar según bit N
ifeq1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
ifeq2	OPC = TOS	Guardar TOS en OPC temporalmente
ifeq3	TOS = MDR	Poner nuevo tope de pila en TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Ramificar según bit Z
if_icmpeq1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
if_icmpeq2	MAR = SP = SP – 1	Ajustar MAR para leer nuevo tope de pila
if_icmpeq3	H = MDR; rd	Copiar 2a. palabra de pila en H
if_icmpeq4	OPC = TOS	Guardar TOS en OPC temporalmente
if_icmpeq5	TOS = MDR	Poner nuevo tope de pila en TOS
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	Si 2 pal. en tope iguales, ir a T; si no, a F
T	OPC = PC – 1; fetch; goto goto2	Mismo que goto1; necesario p/dir. objetivo
F	PC = PC + 1	Saltar primer byte de distancia
F2	PC = PC + 1; fetch	PC ahora apunta a siguiente cód. op.
F3	goto Main1	Esperar que traigan código de operación
invokevirtual1	PC = PC + 1; fetch	MBR = byte índice 1; inc. PC, obt. 2o. byte
invokevirtual2	H = MBRU << 8	Desplazar y guardar primer byte en H
invokevirtual3	H = MBRU OR H	H = dist. de apunt. a método, a CPP
invokevirtual4	MAR = CPP + H; rd	Obt. apunt. al método del área de CPP
invokevirtual5	OPC = PC + 1	Guardar temporalmente PC de retorno en OPC
invokevirtual6	PC = MDR; fetch	PC apunta a nuevo método; obt. cuenta de parám.
invokevirtual7	PC = PC + 1; fetch	Traer 2o. byte de cuenta de parámetros
invokevirtual8	H = MBRU << 8	Desplazar y guardar primer byte en H
invokevirtual9	H = MBRU OR H	H = número de parámetros
invokevirtual10	PC = PC + 1; fetch	Traer primer byte de # locales
invokevirtual11	TOS = SP – H	TOS = dirección de OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = dirección de OBJREF (nuevo LV)
invokevirtual13	PC = PC + 1; fetch	Traer 2o. byte de # locales
invokevirtual14	H = MBRU << 8	Desplazar y guardar primer byte en H
invokevirtual15	H = MBRU OR H	H = # locales
invokevirtual16	MDR = SP + H + 1; wr	Sobreescribir OBJREF con apunt. enlace
invokevirtual17	MAR = SP = MDR;	Asignar a SP, MAR, pos. p/guardar viejo PC
invokevirtual18	MDR = OPC; wr	Guardar viejo PC arriba de variables locales
invokevirtual19	MAR = SP = SP + 1	SP apunta a pos. p/guardar viejo LV
invokevirtual20	MDR = LV; wr	Guardar viejo LV arriba de PC guardado
invokevirtual21	PC = PC + 1; fetch	Traer 1er. cód. op. del nuevo método
invokevirtual22	LV = TOS; goto Main1	Hacer que LV apunte a marco de var. loc.

Figura 4-17. El microprograma del Mic-1 (parte 2 de 3).

Mnemónico	Operaciones	Comentarios
ireturn1	MAR = SP = LV; rd	Restab. SP, MAR, p/obtener apunt. enlace
ireturn2		Esperar lectura
ireturn3	LV = MAR = MDR; rd	Asignar apunt. enlace a LV; obt. viejo PC
ireturn4	MAR = LV + 1	Ajustar MAR para leer viejo LV
ireturn5	PC = MDR; rd; fetch	Restaurar PC; traer siguiente cód. op.
ireturn6	MAR = SP	Ajustar MAR p/escribir TOS
ireturn7	LV = MDR	Restaurar LV
ireturn8	MDR = TOS; wr; goto Main1	Guardar valor de retorno en tope original

Figura 4-17. El microprograma del Mic-1 (parte 3 de 3).

Si sucede que el byte en MBR sólo contiene ceros, el código de una instrucción NOP, la siguiente microinstrucción es la rotulada nop1, que se obtiene de la posición 0. Puesto que esta instrucción no hace nada, simplemente salta de nuevo al inicio del ciclo principal, donde la secuencia se repite, sólo que ya se había obtenido un nuevo código de operación y se colocó en MBR.

Hacemos hincapié una vez más en que las microinstrucciones de la figura 4-17 no son consecutivas en la memoria y que Main1 no está en la dirección 0 del almacén de control (porque nop1 tiene que estar en la dirección 0). Corresponde al microensamblador colocar cada microinstrucción en una dirección apropiada y vincularlas en secuencias cortas empleando el campo NEXT_ADDRESS. Cada secuencia inicia en la dirección que corresponde al valor numérico del código de operación IJVM que interpreta (por ejemplo, POP inicia en 0x57), pero el resto de la secuencia puede estar en cualquier lugar del almacén de control, y no necesariamente en direcciones consecutivas.

Consideremos ahora la instrucción IADD de IJVM. La microinstrucción a la que se ramifica desde el ciclo principal es la rotulada iadd1. Esta instrucción inicia las tareas específicas de IADD:

1. TOS ya está presente, pero se debe obtener de la memoria la segunda palabra desde el tope de la pila hacia abajo.
2. TOS se debe sumar a la segunda palabra obtenida de la memoria.
3. El resultado, que se colocará en la pila, se debe almacenar en la memoria y también en el registro TOS.

Para obtener el operando almacenado en la memoria es necesario decrementar el apuntador de la pila y escribirlo en MAR. Observe que, por suerte, esta dirección también es la que se usará para la escritura subsecuente. Además, puesto que esta posición será el nuevo tope de la pila, hay que asignar este valor a SP. Por tanto, una sola operación puede determinar el nuevo valor de SP y MAR, decrementar SP, y escribirlo en ambos registros.

Estas cosas se logran en el primer ciclo, iadd1, y se inicia la operación de lectura. Además, MPC recibe el valor del campo NEXT_ADDRESS de iadd1, que es la dirección de iadd2, donde sea que esté. Luego se lee iadd2 del almacén de control. En el segundo ciclo, mientras esperamos que se lea el operando de la memoria, copiamos la palabra del tope de la pila, de TOS a H, donde estará disponible para la suma cuando la lectura termine.

Al principio del tercer ciclo, iadd3, MDR contiene el sumando que se obtuvo de la memoria. En este ciclo ese sumando se suma al contenido de H, y el resultado se guarda en MDR y también en TOS. De igual manera, se inicia una operación de escritura que almacena la nueva palabra del tope de la pila en la memoria. En este ciclo el goto tiene el efecto de asignar la dirección de Main1 a MPC, lo que nos regresa al punto de partida para ejecutar la siguiente instrucción.

Si el código de operación IJVM subsecuente, que ahora está contenido en MBR, es 0x64 (ISUB), ocurre otra vez una serie de sucesos casi exactamente igual. Una vez que se ejecuta Main1, el control se transfiere a la microinstrucción que está en 0x64 (isub1). Esta microinstrucción va seguida de isub2 e isub3, y luego de Main1 otra vez. La única diferencia entre esta secuencia y la anterior es que en isub3 el contenido de H se resta de MDR en lugar de sumársele.

La interpretación de IAND es casi idéntica a la de IADD e ISUB, excepto que se obtiene el AND bit por bit de las dos palabras del tope de la pila, en lugar de la suma o la diferencia. Algo similar ocurre con IOR.

Si el código de operación IJVM es DUP, POP o SWAP, es preciso ajustar la pila. La instrucción DUP simplemente duplica la palabra que está en el tope de la pila. Puesto que el valor de esta palabra ya está almacenado en TOS, la operación sólo requiere incrementar SP de modo que apunte a la nueva posición, y guardar TOS en esa posición. La instrucción POP es casi igual de sencilla, pues sólo decrementa SP para desechar la palabra que estaba en el tope de la pila. Sin embargo, para mantener en TOS la palabra que está en el tope de la pila ahora es necesario leer esa palabra de la memoria y escribirla en TOS. Por último, la instrucción SWAP implica intercambiar los valores de dos localidades de memoria: las dos palabras que están hasta arriba en la pila. Esto se facilita un poco por el hecho de que TOS ya contiene uno de esos valores, de modo que no se tiene que leer de la memoria. Examinaremos con mayor detalle esta instrucción más adelante.

La instrucción BIPUSH es un poco más complicada porque el código de operación va seguido de un solo byte, como se muestra en la figura 4-18. El byte se debe interpretar como un entero con signo. Este byte, que ya se obtuvo y se colocó en MBR en Main1, se debe extender con signo a 32 bits y meter en el tope de la pila. Por tanto, esta secuencia debe extender a 32 bits el signo del byte que está en MBR y copiarlo en MDR. Por último, SP se incrementa y se copia en MAR, lo que permite escribir el operando en el tope de la pila. Además, este operando debe copiarse en TOS. Cabe señalar que, antes de regresar al programa principal, se debe incrementar PC para que el siguiente código de operación esté disponible en Main1.

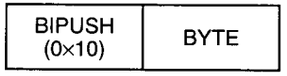


Figura 4-18. El formato de instrucción BIPUSH.

Consideremos ahora la instrucción ILOAD. Ésta también tiene un byte después del código de operación, como se muestra en la figura 4-19(a), pero este byte es un índice (sin signo)

que identifica la palabra del espacio de variables locales que debe almacenarse en la pila. Puesto que sólo hay un byte, sólo pueden distinguirse $2^8 = 256$ palabras, a saber, las primeras 256 palabras del espacio de variables locales. La instrucción ILOAD requiere tanto una lectura (para obtener la palabra) como una escritura (para almacenarla en el tope de la pila). Sin embargo, para determinar la dirección de lectura es preciso sumar al contenido de LV la distancia, contenida en MBR. Puesto que tanto MBR como LV sólo pueden accederse a través del bus B, primero se copia LV en H (en iload1), y luego se le suma MBR. El resultado de esta suma se copia en MAR y se inicia una lectura (en iload2).

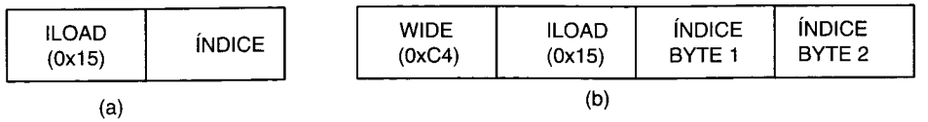


Figura 4-19. (a) ILOAD con un índice de un byte. (b) WIDE ILOAD con un índice de 2 bytes.

Aquí el uso de MBR como índice es un poco diferente que en BIPUSH, donde se extendía su signo. En el caso de un índice, la distancia siempre es positiva, por lo que la distancia en bytes debe interpretarse como un entero sin signo, a diferencia de en BIPUSH, donde se interpretaba como un entero de 8 bits con signo. La interfaz entre MBR y el bus B se diseñó con cuidado para hacer posibles ambas operaciones. En el caso de BIPUSH (entero de 8 bits con signo), la operación correcta es la extensión de signo, es decir, el copiado del bit de la extrema izquierda de MBR (que tiene un byte de longitud) en los 24 bits de orden alto del bus B. En el caso de ILOAD (entero de 8 bits sin signo), la operación correcta es el llenado con ceros. Aquí los 24 bits de orden alto del bus B simplemente se ponen en 0. Estas dos operaciones se distinguen por señales individuales que indican cuál operación debe efectuarse (vea la figura 4-6). En el microcódigo, esto se indica con MBR (extensión de signo, como en BIPUSH3) o MBRU (como en iload2).

Mientras se espera que la memoria proporcione el operando (en iload3), SP se incrementa para que indique dónde se guardará el resultado, el nuevo tope de pila. Este valor también se copia en MAR como preparación a escribir el operando en el tope de la pila. Se debe incrementar PC otra vez para traer el siguiente código de operación (en iload4). Por último, se copia MDR en TOS para que éste refleje el nuevo tope de la pila (en iload5).

ISTORE es la operación opuesta a ILOAD, es decir, se saca una palabra del tope de la pila y se almacena en la posición especificada por la suma de LV y el índice contenido en la instrucción. ISTORE emplea el mismo formato que ILOAD (figura 4-19(a)), excepto que con el código de operación 0x36 en lugar de 0x15. Esta instrucción es un poco diferente de lo que cabría esperar porque ya se conoce la palabra que está en el tope de la pila (está en TOS), así que puede almacenarse inmediatamente. Sin embargo, es preciso traer la palabra que será el nuevo tope de la pila, así que se requieren una lectura y una escritura, aunque pueden efectuarse en cualquier orden (o incluso en paralelo, si ello fuera posible).

Tanto ILOAD como ISTORE están restringidas en cuanto a que sólo accesan las primeras 256 variables locales. Si bien para la mayor parte de los programas este espacio de variables locales es más que suficiente, es desde luego necesario poder acceder a una variable en cualquier lugar que esté dentro del espacio de variables locales. Para lograr esto, IJVM usa el mismo mecanismo empleado en JVM: con código de operación especial WIDE, llamado **byte prefijo**, seguido del código de operación de ILOAD o ISTORE. Cuando ocurre esta secuencia, las definiciones de ILOAD e ISTORE se modifican, y un índice de 16 bits sigue al código de operación en lugar de un índice de 8 bits, como se muestra en la figura 4-19(b).

WIDE se decodifica de la forma acostumbrada y causa una ramificación a wide1, que maneja el código de operación WIDE. Aunque el código de operación para ensanchar ya está en MBR, wide1 trae el primer byte que sigue al código de operación porque la lógica del microprograma siempre espera que esté ahí. Luego se efectúa una segunda ramificación multivías, esta vez empleando el byte que sigue a WIDE para despachar. Sin embargo, dado que WIDE ILOAD requiere diferente microcódigo que ILOAD, y WIDE ISTORE requiere diferente microcódigo que ISTORE, etc., la segunda ramificación multivías no puede utilizar simplemente el código de operación como dirección objetivo, que es lo que hace Main1.

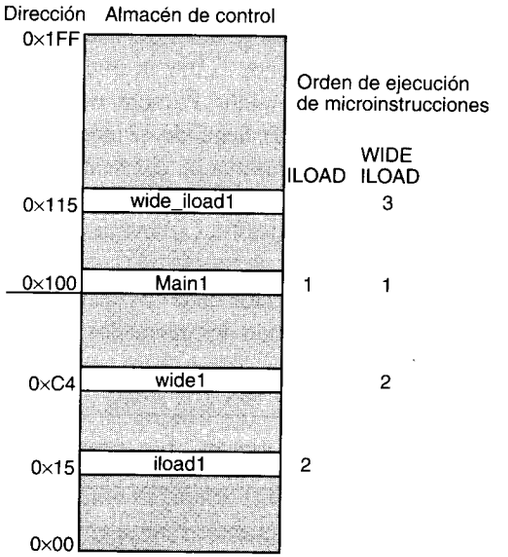


Figura 4-20. La secuencia de microinstrucciones inicial para ILOAD y WIDE ILOAD. Las direcciones son ejemplos.

En vez de ello, wide1 hace un OR de 0x100 con el código de operación mientras lo coloca en MPC. El resultado es que la interpretación de WIDE ILOAD inicia en 0x115 (en lugar de 0x15), la interpretación de WIDE ISTORE inicia en 0x136 (en lugar de 0x36), y así. De este modo, todos los códigos de operación WIDE inician en una dir 256 (o sea, 0x100) palabras más alta en el almacén de control que el código de operación normal correspondiente. La secuencia inicial de microinstrucciones para ILOAD y para WIDE ILOAD se muestra en la figura 4-20.

Una vez que se llega al código para implementar una WIDE ILOAD (0x115), el código difiere del ILOAD normal sólo en cuanto a que el índice debe construirse concatenando dos bytes de índice y no extendiendo el signo de un solo byte. La concatenación y la suma subsecuente deben efectuarse en etapas, copiando primero en H el primer byte del índice desplazado 8 bits a la izquierda. Puesto que el índice es un entero sin signo, MBR se extiende con ceros empleando MBRU. Ahora se suma el segundo byte del índice (la operación de suma es idéntica a la concatenación porque el byte de orden bajo de H ahora es cero, lo que garantiza que no habrá acarreo entre los bytes) y el resultado se vuelve a guardar en H. De aquí en adelante la operación puede proceder exactamente como si se tratara de un ILOAD estándar. En lugar de repetir las instrucciones finales de ILOAD (iload3 a iload5), simplemente saltamos de wide_iload4 a iload3. Observe, sin embargo, que es preciso incrementar PC dos veces durante la ejecución de la instrucción para que al final esté apuntando al siguiente código de operación. La instrucción ILOAD lo incrementa una vez; la secuencia WIDE_ILOAD también lo incrementa una vez.

La misma situación ocurre con WIDE_ISTORE: después de ejecutarse las primeras cuatro microinstrucciones (wide_istore1 a wide_istore4), la secuencia es la misma que en el caso de ISTORE después de las primeras dos instrucciones, así que wide_istore4 salta a istore3.

El siguiente ejemplo que consideraremos es una instrucción LDC_W. Este código de operación es diferente de ILOAD en dos sentidos. Primero, tiene una distancia de 16 bits sin signo (igual que la versión ancha de ILOAD). Segundo, se indexa respecto a CPP, no respecto a LV, ya que su función es leer de la reserva de constantes, no del marco de variables locales. (En realidad, hay una forma corta de LDC_W (LDC), pero no la incluimos en IJVM porque la forma larga incluye todas las posibles variaciones de la forma corta pero ocupa 3 bytes en lugar de 2.)

La instrucción IINC es la única instrucción de IJVM aparte de ISTORE que puede modificar una variable local. Lo hace incluyendo dos operandos, cada uno de un byte, como se muestra en la figura 4-21.

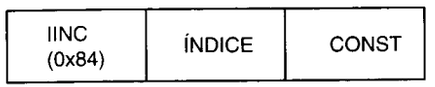


Figura 4-21. La instrucción IINC tiene dos campos de operando distintos.

La instrucción IINC usa ÍNDICE para especificar la distancia desde el principio del marco de variables locales. IINC lee esa variable, la incrementa en CONST, un valor contenido en la instrucción, y la vuelve a guardar en la misma localidad. Cabe señalar que esta instrucción puede incrementar negativamente, es decir, CONST es una constante de 8 bits con signo en el intervalo –128 a +127. La JVM completa incluye una versión ancha de IINC en la que cada operando ocupa 2 bytes.

Ahora llegamos a la primera instrucción de ramificación de IJVM, GOTO. La única función de esta instrucción es modificar el valor de PC, de modo que la siguiente instrucción IJVM que se ejecute sea la que está en la dirección que se calcula sumando la distancia de 16 bits (con signo) a la dirección del código de operación de ramificar. Una complicación aquí

es que la distancia es relativa al valor que PC tenía cuando comenzó a decodificarse la instrucción, no el valor que tiene una vez que se han traído los dos bytes que indican la distancia.

Para aclarar bien esto, en la figura 4-22(a) vemos la situación al inicio de Main1. El código de operación ya está en MBR, pero todavía no se ha incrementado PC. En la figura 4-22(b) vemos la situación al principio de goto1. A estas alturas ya se incrementó PC y el primer byte de la distancia se trajo y se colocó en MBR. Una microinstrucción después tenemos la situación de la figura 4-22(c), en la que el antiguo PC, que apunta al código de operación, se ha guardado en OPC. Este valor es necesario porque la distancia de la instrucción GOTO de IJVM es relativa a él, no al valor actual de PC. De hecho, ésta es la razón por la que necesitamos el registro OPC en primera instancia.

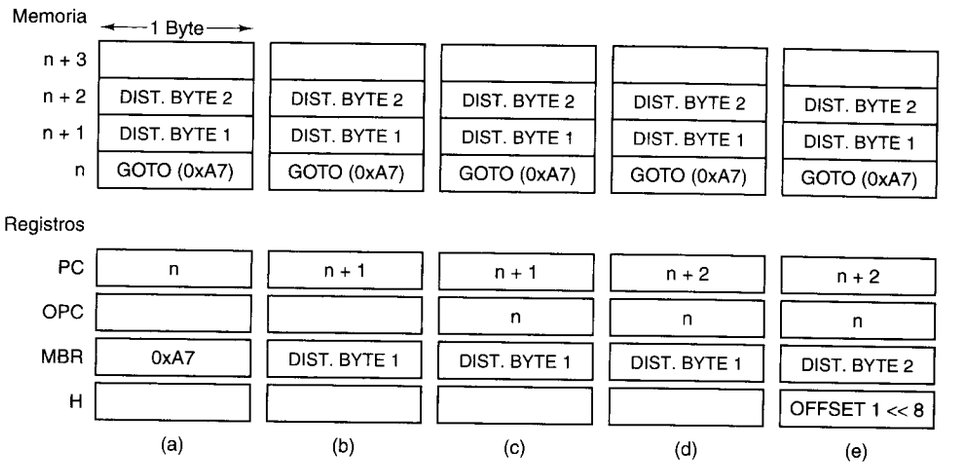


Figura 4-22. La situación al principio de diversas microinstrucciones. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

La microinstrucción en goto2 inicia la búsqueda del segundo byte de distancia, lo que da pie a la figura 4-22(d) al principio de goto3. Una vez que el primer byte de distancia se ha desplazado 8 bits a la izquierda y se ha copiado en H, llegamos a goto4 y a la figura 4-22(e). Ahora tenemos el primer byte de distancia desplazado a la izquierda en H, el segundo byte de distancia en MBR, y la base en OPC. Si construimos la distancia completa de 16 bits en H y la sumamos después a la base, obtenemos la nueva dirección que debe almacenarse en PC, en goto5. Tome nota de que en goto4 usamos MBRU en lugar de MBR porque no queremos extender el signo del segundo byte. De hecho, la distancia de 16 bits se construye haciendo el OR de las dos mitades. Por último, debemos traer el siguiente código de operación antes de regresar a Main1 porque ese código espera que el siguiente código de operación esté en MBR. El último ciclo, goto6, es necesario porque debemos traer los datos de la memoria con tiempo para que aparezcan en MBR durante Main1.

Las distancias empleadas en la instrucción goto de IJVM son valores de 16 bits con signo, con un mínimo de –32768 y un máximo de +32767. Esto implica que las ramificacio-

nes no son posibles en ninguna dirección a puntos más distantes que estos valores. Esta propiedad puede considerarse como un defecto o bien como una característica de IJVM (y también de JVM). Quienes la ven como defecto dirían que la definición de JVM no debería restringir su estilo de programación. Los que la ven como característica afirmarían que el trabajo de muchos programadores mejoraría drásticamente si tuvieran pesadillas acerca del temido mensaje del compilador

Programa demasiado grande y ramificado. Debe reescribirlo. Se aborta la compilación.

Lamentablemente (en nuestra opinión), este mensaje sólo aparece cuando una cláusula then o else excede 32 KB, que por lo regular equivale a unas 50 páginas de Java.

Consideremos ahora las tres instrucciones de ramificación condicional de IJVM: IFLT, IFEQ e IF_ICMPEQ. Las primeras dos sacan la palabra que está en el tope de la pila y toman la rama si la palabra es menor que cero o igual a cero, respectivamente. IF_ICMPEQ saca las dos palabras del tope de la pila y toma la rama si y sólo si son iguales. En los tres casos es necesario leer de la memoria una nueva palabra de tope de la pila para guardarla en TOS.

El control de estas tres instrucciones es similar: el o los operandos se colocan primero en registros, luego se lee el nuevo valor de tope de la pila y se coloca en TOS, y por último se efectúa la prueba y la ramificación. Consideremos primero IFLT. La palabra que vamos a probar ya está en TOS, pero como IFLT saca una palabra de la pila es necesario leer el nuevo tope de pila para guardarlo en TOS. Esta lectura se inicia en iflt1. En iflt2 la palabra que se va a probar se guarda en OPC por el momento a fin de poder colocar el nuevo valor en TOS en breve sin perder el valor actual. En iflt3 la nueva palabra de tope de pila ya está disponible en MDR, así que se copia en TOS. Por último, en iflt4 la palabra que se va a probar, que ahora está en OPC, se pasa por la ALU sin almacenarse y se fija y prueba el bit N. Esta microinstrucción también contiene una ramificación que escoge T si la prueba tuvo éxito, o F en caso contrario.

Si se tiene éxito, el resto de la operación es básicamente igual al principio de la instrucción GOTO, y la ejecución simplemente continúa a la mitad de la secuencia de GOTO, con goto2. Si no se tiene éxito se requiere una secuencia corta (F, F2 y F3) para saltarse el resto de la instrucción (la distancia) antes de regresar a Main1 para continuar con la siguiente instrucción.

El código de ifeq2 e ifeq3 sigue la misma lógica, sólo que usa el bit Z en lugar del bit N. En ambos casos, corresponde al ensamblador de MAL reconocer que las direcciones de T y F son especiales y asegurarse de que se coloquen en direcciones del almacén de control que difieran sólo en el bit de la extrema izquierda.

La lógica para IF_ICMPEQ es parecida a la de IFLT excepto que aquí es necesario leer también el segundo operando. Dicho operando se guarda en H en if_icmpeq3, donde se inicia la lectura de la palabra que será el nuevo tope de pila. En este caso también la palabra que era el tope de pila se guarda en OPC y la nueva se instala en TOS. Por último, la prueba que se efectúa en if_icmpeq6 es similar a la de ifeq4.

Consideremos ahora la implementación de INVOKEVIRTUAL y de RETURN, las instrucciones para efectuar una llamada de procedimiento y para regresar, como se describió en la sección 4.2.3. INVOKEVIRTUAL es una secuencia de 22 microinstrucciones, y es la instrucción IJVM más compleja que se implementó. Su funcionamiento se mostró en la figura

4.12. La instrucción utiliza su distancia de 16 bits para determinar la dirección del método que se invocará. En nuestra implementación, la distancia no es más que una distancia dentro de la Reserva de Constantes. Esta posición dentro de la Reserva de Constantes apunta al método que se invocará. Recuerde, empero, que los primeros 4 bytes de cada método *no* son instrucciones. Más bien, son dos apuntadores de 16 bits. El primero proporciona el número de palabras de parámetros (incluido OBJREF; vea la figura 4-12). El segundo proporciona el tamaño del área de variables locales en palabras. Estos campos se obtienen a través del puerto de 8 bits y se arman como si fueran dos distancias de 16 bits dentro de una instrucción.

Luego, la información de enlace necesaria para restablecer la máquina a su estado anterior —la dirección del inicio de la antigua área de variables locales y el antiguo PC— se guarda inmediatamente arriba del área de variables locales recién creada y debajo de la nueva pila. Por último, se obtiene el código de operación de la siguiente instrucción y PC se incrementa antes de regresar a Main para iniciar la siguiente instrucción.

IRETURN es una instrucción sencilla que no contiene operandos; simplemente usa la dirección almacenada en la primera palabra del área de variables locales para recuperar la información de enlace. Luego restaura SP, LV y PC a sus valores anteriores y copia el valor devuelto, del tope de la pila actual al tope de la pila original, como se muestra en la figura 4-13.

4.4 DISEÑO DEL NIVEL DE MICROARQUITECTURA

Como casi todo en computación, el diseño del nivel de microarquitectura está lleno de concesiones. Las computadoras tienen muchas características deseables, como rapidez, bajo costo, confiabilidad, facilidad de uso, bajo consumo de energía y tamaño físico. Sin embargo, el equilibrio que determina las decisiones más importantes que el diseñador de una CPU debe hacer es el de rapidez contra costo. En esta sección examinaremos este aspecto con detalle para ver qué puede sacrificarse para obtener qué, cuál es el máximo desempeño que puede alcanzarse, y cuál es su precio en términos de hardware y complejidad.

4.4.1 Rapidez *versus* costo

Si bien la creación de circuitos que operan a mayor velocidad ha sido el factor más importante para lograr computadoras más rápidas, ello rebasa el alcance de este texto. Los aumentos en rapidez logrados gracias a una mejor organización, aunque son menos espectaculares, también han sido impresionantes. Existen varias formas de medir la rapidez, pero dada una tecnología de circuitos y una ISA, hay tres estrategias básicas para aumentar la velocidad de ejecución:

1. Reducir el número de ciclos de reloj necesarios para ejecutar una instrucción.
2. Simplificar la organización para que el ciclo de reloj pueda ser más corto.
3. Traslapar la ejecución de instrucciones.

Las dos primeras son obvias, pero existe una sorprendente variedad de decisiones de diseño que pueden afectar drásticamente ya sea el número de ciclos de reloj, el periodo del reloj o, lo

que es más común, ambas cosas. En esta sección daremos un ejemplo de cómo la codificación y decodificación de una operación pueden afectar el ciclo de reloj.

El número de ciclos de reloj necesarios para ejecutar un conjunto de operaciones se denomina **longitud de trayectoria**. En ocasiones, la longitud de la trayectoria puede acortarse añadiendo hardware especializado. Por ejemplo, si agregamos un incrementador (que en lo conceptual es un sumador con un lado conectado permanentemente de modo que sume 1) a PC, ya no será necesario usar la ALU para incrementar PC, y en consecuencia se eliminan ciclos. El precio que se paga es un aumento en el hardware. Sin embargo, esta capacidad no ayuda tanto como cabría esperar. En casi todas las instrucciones los ciclos que se consumen incrementando PC son también ciclos durante los cuales se está efectuando una operación de lectura. De todos modos no sería posible ejecutar la siguiente instrucción antes porque depende de los datos que se están trayendo de la memoria.

Reducir el número de ciclos de instrucción necesarios para traer instrucciones requiere algo más que un circuito adicional para incrementar el contador de programa. Si queremos acelerar la obtención de instrucciones de forma apreciable es preciso aprovechar la tercera técnica: traslapar la ejecución de instrucciones. La separación de los circuitos que traen las instrucciones —el puerto de memoria de 8 bits y los registros MBR y PC— es más eficaz si se hace que la unidad sea funcionalmente independiente de la trayectoria de datos principal. Así, esos circuitos pueden traer el siguiente código de operación u operando por sí solos, tal vez operando asincrónicamente respecto al resto de la CPU y obteniendo una o más instrucciones por adelantado.

Una de las fases de la ejecución de muchas instrucciones que más tiempo consume es traer una distancia de dos bytes, extenderla de la forma apropiada, y acumularla en el registro H como preparación para una suma, por ejemplo, en una ramificación a PC ± n bytes. Una posible solución —hacer que el puerto de memoria tenga un ancho de 16 bits— complica considerablemente la operación, porque la memoria en realidad tiene una anchura de 32 bits. Los 16 bits requeridos podrían cruzar la frontera entre dos palabras, de modo que ni siquiera una sola lectura de 32 bits necesariamente traería los dos bytes que se necesitan.

Traslapar la ejecución de las instrucciones es por mucho el método más interesante y el que más oportunidades ofrece para aumentar drásticamente la rapidez. El simple traslapo de la búsqueda y la ejecución de las instrucciones es sorprendentemente eficaz. Sin embargo, las técnicas más sofisticadas van mucho más lejos, pues traslapan la ejecución de muchas instrucciones. De hecho, esta idea es fundamental para el diseño de computadoras modernas. A continuación analizaremos algunas técnicas básicas para traslapar la ejecución de instrucciones y delinearemos algunas de las más complejas.

La rapidez es sólo la mitad de la historia; la otra mitad es el costo. Además, hay varias formas de medir el costo, y definir éste con precisión es problemático. Algunas medidas son tan sencillas como un recuento del número de componentes. Esto era válido sobre todo en los tiempos en que los procesadores se construían con componentes discretos que se compraban y armaban. Hoy día todo el procesador existe en un solo chip, pero los chips grandes y complejos son más caros que los pequeños y sencillos. Podemos contar los componentes individuales, como transistores, compuertas o unidades funcionales, pero a menudo esa cuenta no es tan importante como el área que se requiere en el circuito integrado. Entre mayor sea el

área que se requiera para las funciones incluidas, más grande será el chip. Y el costo de fabricación del chip crece con mucha mayor rapidez que su área. Por esta razón, los diseñadores a menudo hablan del costo en términos de "terreno", es decir, el área requerida para un circuito (¿quizá medida en picohectáreas?).

Uno de los circuitos más minuciosamente estudiados en la historia es el sumador binario. Se han creado miles de diseños, y los más rápidos son mucho más veloces que los más lentos, aunque también son mucho más complejos. El diseñador de sistemas tiene que decidir si el aumento en la velocidad justifica ocupar más "terreno".

Los sumadores no son el único componente que ofrece muchas opciones. Casi todos los componentes del sistema se pueden diseñar de modo que operen con mayor o menor velocidad, pero con diferente costo. El reto del diseñador es identificar los componentes del sistema que pueden mejorarlo más si se hacen más rápidos. Resulta interesante que muchos componentes individuales se pueden sustituir por un componente mucho más rápido sin que se observe un efecto apreciable sobre la rapidez del sistema. En las secciones que siguen examinaremos algunas cuestiones de diseño y los equilibrios costo-rapidez que implican.

Uno de los factores clave para determinar con qué rapidez puede operar el reloj es la cantidad de trabajo que debe efectuarse en cada ciclo de reloj. Obviamente, cuanto más trabajo haya que realizar, más largo será el ciclo de reloj. Claro que la cosa no es tan sencilla, porque el hardware es muy bueno para hacer cosas en paralelo, así que es realmente la secuencia de operaciones que deben efectuarse *en serie* en un solo ciclo de reloj lo que determina la duración que debe tener el ciclo.

Un aspecto que puede controlarse es la cantidad de decodificación que hay que realizar. Recuerde, por ejemplo, que en la figura 4-6 vimos que si bien era posible introducir cualquiera de nueve registros en la ALU por medio del bus B, sólo requeríamos 4 bits de la palabra de la microinstrucción para especificar cuál registro había que seleccionar. Lo malo es que tales ahorros tienen un precio. El circuito decodificador aumenta el retraso en la ruta crítica. Esto implica que el registro que colocará sus datos en el bus B recibirá esa orden y después pondrá sus datos en el bus. Este efecto se propaga, pues la ALU recibe sus entradas un poco más tarde y produce sus resultados un poco más tarde. Por último, el resultado está disponible en el bus C para escribirse en los registros un poco más tarde. Puesto que este retraso en muchos casos es el factor que determina la longitud que debe tener el ciclo de reloj, podría implicar que el reloj no puede operar tan rápidamente como podría y que toda la computadora debe ser un poco más lenta. Así pues, hay un equilibrio entre rapidez y costo. Reducir el almacén de control en 5 bits por palabra tiene el costo de frenar el reloj. El ingeniero de diseño debe tener en cuenta los objetivos de diseño para decidir qué es lo que más conviene. En el caso de una implementación de alto desempeño, probablemente no es recomendable usar un decodificador; en una de bajo costo tal vez sería lo mejor.

4.4.2 Reducción de la longitud de la trayectoria de ejecución

El Mic-1 se diseñó tratando de hacerlo moderadamente sencillo y rápido, aunque hemos de admitir que existe una tensión enorme entre estas dos metas. En pocas palabras, las máquinas sencillas no son rápidas y las máquinas rápidas no son sencillas. Además, la CPU del Mic-1

emplea un mínimo de hardware: 10 registros, la sencilla ALU de la figura 3-19 repetida 32 veces, un desplazador, un decodificador, un almacén de control y un poco de adhesivo aquí y allá. Todo el sistema podría construirse con menos de 5000 transistores más lo que necesiten el almacén de control (ROM) y la memoria principal (RAM).

Habiendo visto cómo puede implementarse IJVM de forma directa en microcódigo con poco hardware, es momento de examinar otras implementaciones más rápidas. A continuación estudiaremos formas de reducir el número de microinstrucciones por cada instrucción ISA (es decir, reducir la longitud de la trayectoria de ejecución). Después consideraremos otras estrategias.

Fusión del ciclo del intérprete con el microcódigo

En el Mic-1, el ciclo principal consiste en una microinstrucción que debe ejecutarse al principio de cada instrucción IJVM. En algunos casos es posible traslaparla con la instrucción anterior. De hecho, esto ya se logró parcialmente. Observe que cuando se ejecuta Main1 el código de operación que se va a interpretar ya está en MBR. Está ahí porque el ciclo principal anterior la trajo (si la instrucción anterior no tenía operandos) o bien porque se trajo durante la ejecución de la instrucción anterior.

Este concepto de traslapar el principio de la instrucción puede llevarse más lejos, y en algunos casos el ciclo principal puede reducirse a la nada. Esto puede ocurrir como sigue. Consideremos las secuencias de microinstrucciones que terminan con un salto a Main1. En cada uno de estos puntos, la microinstrucción del ciclo principal puede anexarse al final de la sucesión (en lugar de estar al principio de la sucesión siguiente) de modo que la ramificación multivías esté repetida en muchos lugares (pero siempre con el mismo conjunto de objetivos). En algunos casos la microinstrucción Main1 se puede fusionar con microinstrucciones previas, ya que tales instrucciones no siempre se aprovechan plenamente.

En la figura 4-23 se muestra la sucesión dinámica de instrucciones para una instrucción POP. El ciclo principal ocurre antes y después de cada instrucción; en la figura mostramos sólo la ocurrencia que sigue a la instrucción POP. Observe que la ejecución de esta instrucción tarda cuatro ciclos: tres para las microinstrucciones específicas de POP y una para el ciclo principal.

Mnemónico	Operaciones	Comentarios
pop1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
pop2		Esperar lectura de nueva TOS de la memoria
pop3	TOS = MDR; goto Main 1	Copiar nueva palabra en TOS
Main	PC = PC + 1; fetch; goto (MBR)	MBR tiene cód. op.; obt. sig. byte. despachar

Figura 4-23. Nueva secuencia de microprograma para ejecutar POP.

En la figura 4-24 hemos reducido la sucesión a tres instrucciones fusionando las instrucciones del ciclo principal, aprovechando un ciclo de reloj en el que la ALU no se usa en pop2 para ahorrar un ciclo, y otra vez en Main1. Observe que el final de esta sucesión salta directamente al código específico de la instrucción subsecuente, de modo que sólo se requiere un total de tres ciclos. Este pequeño truco reduce el tiempo de ejecución de la siguiente microinstrucción en un ciclo, de modo que, por ejemplo, una IADD subsecuente baja de cuatro ciclos a tres, y esto equivale a acelerar el reloj de 250 MHz (microinstrucciones de 4 ns) a 333 MHz (microinstrucciones de 3 ns) gratis.

Mnemónico	Operaciones	Comentarios
pop1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
Main1.pop	PC = PC + 1; fetch	MBR contiene cód. op.; traer siguiente byte
pop3	TOS = MDR; goto (MBR)	Copiar nva. pal. en TOS; despachcar según cód. op.

Figura 4-24. Secuencia de microprograma mejorada para ejecutar POP.

La instrucción POP se presta mucho para este tratamiento porque tiene un ciclo muerto a la mitad que no usa la ALU. El ciclo principal, en cambio, sí usa la ALU. Por tanto, para reducir la longitud de las instrucciones en un ciclo es necesario encontrar un ciclo de la instrucción en el que la ALU no se esté usando. Tales ciclos muertos no son comunes, pero sí ocurren, así que vale la pena fusionar Main1 con el final de cada sucesión de microinstrucciones. Lo único que cuesta es un poco de espacio en el almacén de control. Así pues, ya tenemos nuestra primera técnica para reducir la longitud de la trayectoria:

Fusionar el ciclo del intérprete con el final de cada sucesión de microcódigo.

Una arquitectura de tres buses

¿Qué otra cosa podemos hacer para reducir la longitud de la trayectoria de ejecución? Otra solución fácil es que la ALU tenga dos buses de entrada completos, un bus A y un bus B, para un total de tres buses. Todos (o al menos casi todos) los registros deberán tener acceso a ambos buses de entrada. La ventaja de tener dos buses de entrada es que hace posible sumar cualesquier dos registros en un solo ciclo. Para ver el valor de esta característica, consideremos la implementación de ILOAD en Mic-1, que se muestra otra vez en la figura 4-25.

Mnemónico	Operaciones	Comentarios
iload1	H = LV	MBR contiene índice; copiar LV en H
iload2	MAR = MBRU + H; rd	MAR = dir. de variable local que se meterá
iload3	MAR = SP = SP + 1	SP apunta a nuevo tope de pila; preparar escrit.
iload4	PC = PC + 1; fetch; wr	Inc. PC; obt. sig. cód. op.; escribir en tope de pila
iload5	TOS = MDR; goto Main1	Actualizar TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR tiene cód. op.; obt. sig. byte; despachar

Figura 4-25. Código de Mic-1 para ejecutar ILOAD.

Aquí vemos que en iload1 se copia LV en H. La única razón por la que se hace esto es para sumarlo a MBRU en iload2. En nuestro diseño original de dos buses, no hay forma de

sumar dos registros arbitrarios, por lo que primero hay que copiar uno de ellos en H. Con nuestro nuevo diseño de tres buses podemos ahorrar un ciclo, como se muestra en la figura 4-26. Aquí añadimos el ciclo del intérprete a ILOAD, aunque hacerlo no hace la trayectoria de ejecución ni más corta ni más larga. De todos modos, el bus adicional ha reducido el tiempo de ejecución total de ILOAD de seis ciclos a cinco. Ahora tenemos nuestra segunda técnica para reducir la longitud de la trayectoria:

Cambiar el diseño, de dos buses a tres buses.

Mnemónico	Operaciones	Comentarios
iload1	MAR = MBRU + LV; rd	MAR = dir. de variable local que se meterá
iload2	MAR = SP = SP + 1	SP apunta a nuevo tope de pila; preparar escritura
iload3	PC = PC + 1; fetch; wr	Inc. PC; obt. sig. cód. op.; escribir en tope de pila
iload4	TOS = MDR	Actualizar TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR tiene cód. op.; obt. sig. byte; despachar

Figura 4-26. Código para ejecutar ILOAD con tres buses.

Una unidad de obtención de instrucciones

Las dos técnicas anteriores son recomendables, pero si queremos lograr una mejora drástica necesitamos algo mucho más radical. Retrocedamos y examinemos las partes comunes de todas las instrucciones: la obtención y decodificación de los campos de la instrucción. Observe que en cada instrucción podrían ocurrir las siguientes operaciones:

1. El PC se pasa por la ALU y se incrementa.
2. Se usa el PC para traer el siguiente byte del flujo de instrucciones.
3. Se leen operandos de la memoria.
4. Se escriben operandos en la memoria.
5. La ALU efectúa un cálculo y los resultados se guardan.

Si una instrucción tiene campos adicionales (para los operandos), cada campo debe traerse explícitamente, byte por byte, y armarse antes de que pueda usarse. Obtener y armar un campo ocupa a la ALU durante por lo menos un ciclo por byte para incrementar el PC, y luego otra vez para armar el índice o la distancia resultante. La ALU se usa casi en cada ciclo para efectuar diversas operaciones relacionadas con la obtención de la instrucción y el armado de los campos de la instrucción, además del trabajo "real" de la instrucción.

A fin de traslapar el ciclo principal, es necesario ahorrar a la ALU algunas de estas tareas. Esto podría hacerse introduciendo una segunda ALU, aunque no se necesita una ALU completa para una buena parte de las actividades. Observe que en muchos casos la ALU se usa simplemente como trayectoria para copiar un valor de un registro a otro. Estos ciclos podrían eliminarse introduciendo trayectorias de datos adicionales que no pasen por la ALU. Por ejemplo, podría ser benéfico crear una trayectoria de TOS a MDR, o de MDR a TOS, puesto que la palabra tope de la pila a menudo se copia entre estos dos registros.

En el Mic-1, podríamos aligerar mucho la carga de la ALU si creamos una unidad independiente que traiga y procese las instrucciones. Esta unidad, llamada **unidad de búsqueda de instrucciones** (**IFU**, *Instruction Fetch Unit*) puede incrementar PC de forma independiente y traer bytes del flujo de bytes antes de que se necesiten. Esta unidad sólo requiere un incrementador, un circuito mucho más sencillo que un sumador completo. Si llevamos esta idea más lejos, la IFU también podría ensamblar operandos de 8 y 16 bits de modo que estén listos para usarse de inmediato apenas se necesiten. Hay al menos dos formas de lograr esto:

1. La IFU puede interpretar cada código de operación, determinando cuántos campos adicionales hay que traer, y armarlos en un registro listos para que la unidad de ejecución principal los use.
2. La IFU puede aprovechar la naturaleza de flujo de las instrucciones, y proporcionar en todo momento los siguientes fragmentos de 8 y 16 bits, sea que tenga sentido hacerlo o no. Así, la unidad de ejecución principal puede pedir lo que necesite.

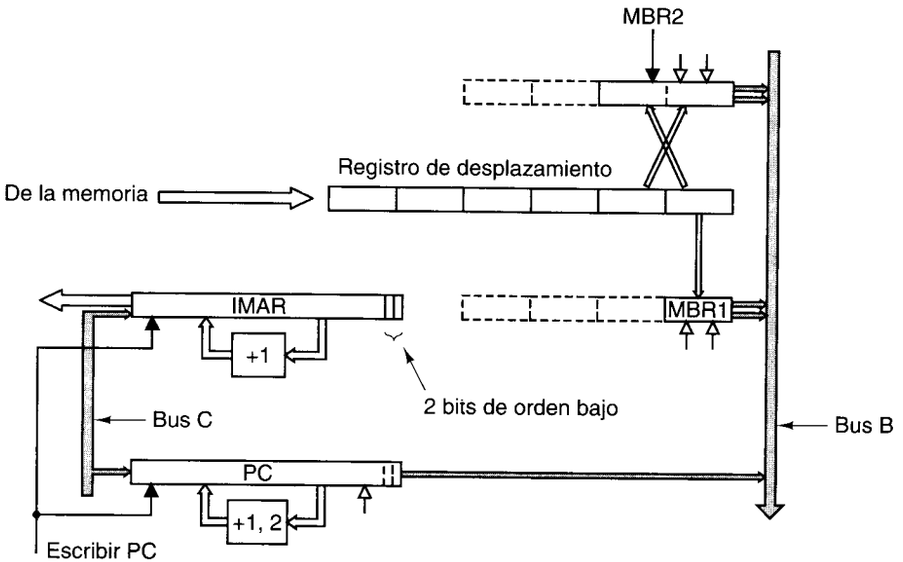


Figura 4-27. Unidad de obtención para el Mic-1.

En la figura 4-27 mostramos los rudimentos del segundo esquema. En lugar de un solo MBR de 8 bits, ahora hay dos MBR: el MBR1 de 8 bits y el MBR2 de 16 bits. La IFU sigue la pista al byte o bytes más recientes consumidos por la unidad de ejecución principal; también proporciona el siguiente byte en MBR1, igual que en el Mic-1, excepto que automáticamente detecta cuando se lee MBR1, prebusca el siguiente byte y lo carga en MBR1 de inmediato. Al igual que en el Mic-1, hay dos interfaces con el bus B: MBR1 y MBR1U. La primera se extiende con el signo a 32 bits; la segunda se extiende con ceros.

De forma similar, MBR2 ofrece la misma funcionalidad pero contiene los dos bytes siguientes, y también tiene dos interfaces con el bus B: MBR2 y MBR2U, que guardan los valores extendidos a 32 bits con el signo y con ceros, respectivamente.

La IFU se encarga de traer una serie de bytes. Para ello, usa un puerto de memoria convencional de 4 bytes, obtiene por adelantado palabras completas de 4 bytes y carga los bytes consecutivos en un registro de desplazamiento que los proporciona de uno en uno o dos a la vez, en el orden en que se obtuvieron de la memoria. La función del registro de desplazamiento es mantener una cola de bytes de la memoria para alimentarlos a MBR1 y MBR2.

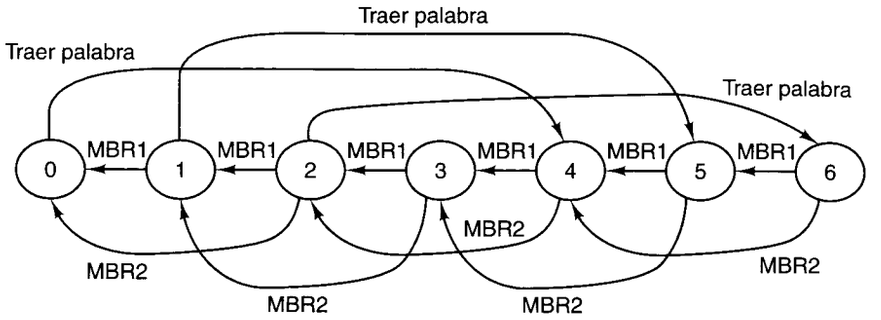
En todo momento MBR1 contiene el byte menos reciente de los que están en el registro de desplazamiento y MBR2 contiene los 2 bytes menos recientes (con el menos reciente a la izquierda) para formar un entero de 16 bits [vea la figura 4-19(b)]. Los dos bytes que están en MBR2 podrían provenir de diferentes palabras de la memoria, porque las instrucciones IJVM no se alinean según las fronteras de palabra en la memoria.

Cada vez que se lee MBR1, el registro de desplazamiento corre su contenido un byte a la derecha. Cada vez que se lee MBR2, el desplazamiento es de dos bytes a la derecha. Luego se vuelven a cargar MBR1 y MBR2 con el byte y el par de bytes menos recientes, respectivamente. Si existe suficiente espacio en el registro de desplazamiento para otra palabra entera, la IFU inicia un ciclo de memoria para leerla. Suponemos que cuando se lee cualquiera de los registros MBR, se ha vuelto a llenar para cuando inicia el siguiente ciclo, así que pueden leerse en ciclos consecutivos.

El diseño de la IFU se puede modelar con una **máquina de estados finitos** (**FSM**, *Finite State Machine*) como se muestra en la figura 4-28. Todas las FSM constan de dos partes: **estados**, que se indican con círculos, y **transiciones**, que se indican con flechas de un estado a otro. Cada estado representa una posible situación en que la FSM puede estar. Esta FSM en particular tiene siete estados, que corresponden a los siete estados del registro de desplazamiento de la figura 4-27. Los siete estados corresponden al número de bytes que están actualmente en el registro de desplazamiento, un número entre 0 y 6, inclusive.

Cada flecha representa un suceso que puede ocurrir. Aquí pueden ocurrir tres sucesos distintos. El primero es que se lea un byte de MBR1. Este suceso hace que el registro de desplazamiento se active y desplace su contenido un byte hacia la derecha, "expulsando" un byte por su extremo derecho y reduciendo su estado en 1. El segundo suceso es que se lean dos bytes de MBR2, lo que reduce el estado en 2. Ambas transiciones hacen que MBR1 y MBR2 se vuelvan a cargar. Cuando la FSM pasa a los estados 0, 1 o 2, se inicia una referencia a la memoria para leer una nueva palabra (suponiendo que la memoria no está ocupada leyendo una palabra). La llegada de la palabra adelanta el estado en 4.

Para que funcione correctamente, la IFU debe bloquearse cuando se le pide que haga algo que no puede hacer, como proporcionar el valor de MBR2 cuando sólo hay un byte en el registro de desplazamiento y la memoria sigue ocupada trayendo una nueva palabra. Además, sólo puede hacer una cosa a la vez, de modo que los sucesos deben colocarse en serie. Por último, cada vez que se modifique PC, habrá que actualizar la IFU. Estos detalles hacen que la IFU sea más complicada de lo que se muestra. No obstante, muchos dispositivos de hardware se construyen como máquinas de estados finitos.



Transiciones
MBR1: Ocurre cuando se lee MBR1
MBR2: Ocurre cuando se lee MBR2
Traer palabra: Ocurre cuando se lee una palabra de memoria y se colocan 4 bytes en el registro de desplazamiento

Figura 4-28. Máquina de estados finitos para implementar la IFU.

La IFU tiene su propio registro de dirección de memoria, llamado IMAR, que se usa para direccionar la memoria cuando es preciso traer una nueva palabra. Este registro tiene su propio incrementador dedicado, así que no se necesita la ALU principal para incrementarlo y así obtener la siguiente palabra. La IFU debe vigilar el bus C de modo que, cada vez que se cargue PC, el nuevo valor de PC también se copie en IMAR. Puesto que el nuevo valor que está en PC no necesariamente está en una frontera de palabra, la IFU tiene que traer la palabra necesaria y ajustar el registro de desplazamiento de manera acorde.

Con la IFU, la unidad de ejecución principal escribe en PC sólo cuando es necesario alterar la naturaleza secuencial del flujo de bytes de las instrucciones. Esto ocurre cuando una instrucción de ramificación condicional tiene éxito y también con INVOKEVIRTUAL y RETURN.

Puesto que el microprograma ya no incrementa explícitamente PC cuando se leen códigos de operación, la IFU debe mantener PC al día. Para ello, detecta cuándo se ha consumido un byte del flujo de instrucciones, es decir, cuándo se han leído MBR1 o MBR2 (o sus versiones sin signo). PC tiene asociado un incrementador propio, capaz de incrementar en 1 o en 2, dependiendo de cuántos bytes se hayan consumido. Así, el PC siempre contiene la dirección del primer byte que no se ha consumido. Al principio de cada instrucción MBR contiene la dirección del código de operación de esa instrucción.

Observe que hay dos incrementadores distintos y que tienen diferentes funciones. PC cuenta *bytes* y se incrementa en 1 o 2. IMAR cuenta *palabras* y se incrementa sólo en 1 (para obtener 4 bytes nuevos). Al igual que MAR, IMAR está conectado al bus de direcciones con sesgo (el bit 0 de IMAR conectado a la línea de dirección 2, etc.) para realizar una conversión implícita de direcciones de palabras en direcciones de bytes.

Como veremos en breve, no tener que incrementar PC en el ciclo principal representa un ahorro considerable, ya que la microinstrucción en la que PC se aumenta no hace mucho más

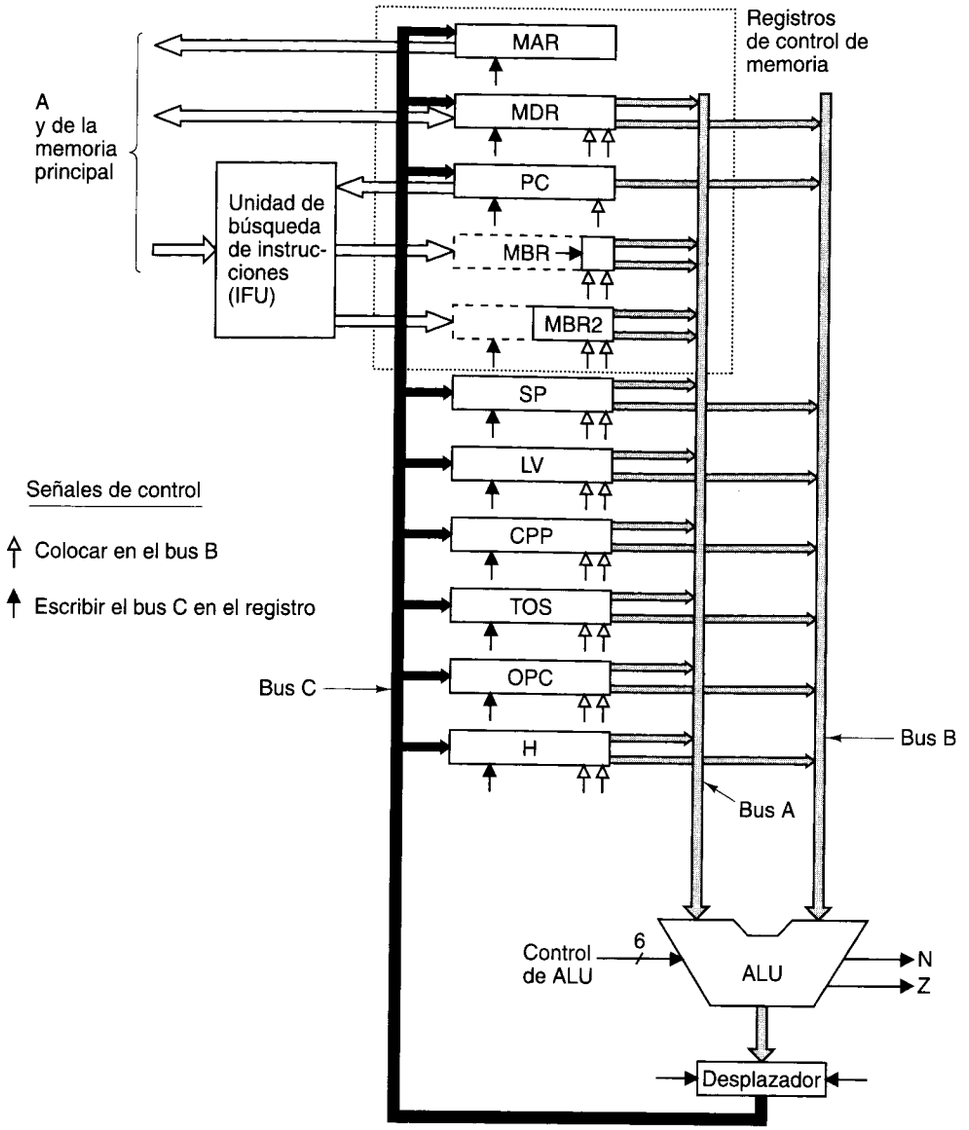


Figura 4-29. Trayectoria de datos del Mic-2.

que incrementar PC. Si podemos eliminar esta microinstrucción, se reducirá la trayectoria de ejecución. El precio aquí es más hardware a cambio de una máquina más rápida, así que nuestra tercera técnica para reducir la longitud de la trayectoria es

Hacer que una unidad especializada traiga las instrucciones de la memoria.

4.4.3 Un diseño con prebúsqueda: el Mic-2

La IFU puede reducir considerablemente la longitud de trayectoria de una instrucción típica. Primero, elimina el ciclo principal, puesto que el final de cada instrucción salta directamente a la siguiente instrucción. Segundo, evita ocupar la ALU en incrementar PC. Tercero, reduce la longitud de trayectoria cada vez que se calcula un índice o distancia de 16 bits, porque arma el valor de 16 bits y lo proporciona directamente a la ALU como valor de 32 bits, con lo que se evita la necesidad de armarlo en H. La figura 4-29 muestra el Mic-2, una versión mejorada del Mic-1 en la que se ha añadido la IFU de la figura 4-27. El microcódigo de la máquina mejorada se muestra en la figura 4-30.

Como ejemplo del funcionamiento del Mic-2, estudiemos IADD. Esta instrucción trae la segunda palabra de la pila y realiza la suma igual que antes, sólo que ahora no tiene que ir a Main1 cuando termina para incrementar PC y saltar a la siguiente microinstrucción. Cuando la IFU ve que se hizo referencia a MBR1 en iadd3, su registro de desplazamiento interno recorre todo a la derecha y vuelve a cargar MBR1 y MBR2; además, efectúa una transición a un estado uno más abajo que su estado actual. Si el nuevo estado es 2, la IFU comienza a leer una palabra de la memoria. Todo esto se hace en hardware. El microprograma no tiene que hacer nada. Es por ello que IADD puede reducirse de cuatro microinstrucciones a tres.

El Mic-2 mejora algunas instrucciones más que otras. LDC_W baja de nueve microinstrucciones a sólo tres, lo que reduce su tiempo de ejecución por un factor de 3. En cambio, SWAP sólo baja de ocho a seis microinstrucciones. En lo que respecta al desempeño global, lo que cuenta realmente es la ganancia que se obtiene en las instrucciones más comunes. Éstas incluyen ILOAD (antes 6, ahora 3), IADD (antes 4, ahora 3) y IF_ICMPEQ (antes 13, ahora 10 si se ramifica; antes 10, ahora 8 si no se ramifica). Para medir la mejora sería necesario escoger algunos programas de referencia y ejecutarlos, pero es evidente que las ganancias aquí son importantes.

4.4.4 Un diseño con filas de procesamiento: el Mic-3

Es obvio que el Mic-2 es mejor que el Mic-1: es más rápido y ocupa menos espacio en el almacén de control, aunque el costo de la IFU sin duda será mayor que lo que se ahorra en "terreno" al tener un almacén de control más pequeño. Por tanto, tenemos una máquina considerablemente más rápida por un precio un poco más alto. Veamos si podemos hacerla más rápida aún.

¿Qué tal si tratamos de reducir el tiempo de ciclo? En gran parte, el tiempo de ciclo está determinado por la tecnología subyacente. Cuanto más pequeños sean los transistores y menores sean las distancias físicas entre ellos, más rápidamente podrá operar el reloj. Para una tecnología dada, el tiempo requerido para efectuar una operación completa de trayectoria de datos está fijo (al menos desde nuestro punto de vista). No obstante, disponemos de cierta libertad que dentro de muy poco aprovecharemos al máximo.

Nuestra otra opción es introducir más paralelismo en la máquina. De momento, el Mic-2 es altamente secuencial: coloca registros en sus buses, espera que la ALU y el desplazador

Mnemónico	Operaciones	Comentarios
nop1	goto (MBR)	Saltar a la siguiente instrucción
iadd1	MAR = SP= SP – 1; rd	Leer 2a. palabra de la pila
iadd2	H = TOS	H = tope de la pila
iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)	Sumar dos palabras del tope; escribir en nvo. tope de la pila
isub1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
isub2	H = TOS	H = tope de la pila
isub3	MDR = TOS = MDR-H; wr; goto (MBR1)	Restar TOS del TOS-1 traído
iand1	MAR = SP = SP - 1; rd	Leer 2a. palabra de la pila
iand2	H = TOS	H = tope de la pila
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	Hacer AND del TOS-1 traído y TOS
ior1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
ior2	H = TOS	H = tope de la pila
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	Hacer OR del TOS-1 traído y TOS
dup1	MAR = SP = SP + 1	Incrementar SP; copiar en MAR
dup2	MDR = TOS; wr; goto (MBR1)	Escribir nueva palabra de la pila
pop1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
pop2		Esperar lectura
pop3	TOS = MDR; goto (MBR1)	Copiar nueva palabra en TOS
swap1	MAR = SP - 1; rd	Leer 2a. palabra de la pila; asignar SP a MAR
swap2	MAR = SP	Repararse para escribir nueva 2a. palabra
swap3	H = MDR; wr	Guardar nuevo TOS; escribir 2a. pal. en pila
swap4	MDR = TOS	Copiar TOS viejo en MDR
swap5	MAR = SP – 1; wr	Escribir TOS viejo en 2o. lugar de la pila
swap6	TOS = H; goto (MBR1)	Actualizar TOS
bipush1	SP = MAR = SP + 1	Reparar MAR p/escribir en nuevo tope de pila
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Actualizar pila en TOS y memoria
iload1	MAR = LV + MBR1U; rd	Pasar LV + índice a MAR; leer operando
iload2	MAR = SP = SP + 1	Incrementar SP; pasar nuevo SP a MAR
iload3	TOS = MDR; wr; goto (MBR1)	Actualizar pila en TOS y memoria
istore1	MAR = LV + MBR1U	Asignar LV + índice a MAR
istore2	MDR = TOS; wr	Copiar TOS para guardarlo
istore3	MAR = SP = SP – 1; rd	Decrementar SP; leer nuevo TOS
istore4		Esperar lectura
istore5	TOS = MDR; goto (MBR1)	Actualizar TOS
wide1	goto (MBR1 OR 0x100)	Sig. dir. es 0x100 OR código de operación
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Idéntico a iload1 pero usa índice de 2 bytes
wide_istore1	MAR = LV + MBR2U; goto istore2	Idéntico a iload1 pero usa índice de 2 bytes
idc_w1	MAR = CPP + MBR2U; rd; goto iload2	Igual que wide_iload1 pero indiza en CPP
iinc1	MAR = LV + MBR1U; rd	Asignar LV + índice a MAR para lectura
iinc2	H = MBR1	Asignar constante a H
iinc3	MDR = MDR + H; wr; goto (MBR1)	Incrementar constante y actualizar
goto1	H = PC – 1	Copiar PC en H
goto2	PC = H + MBR2	Sumar distancia y actualizar PC
goto3		Esperar que IFU traiga nuevo cód. op.
goto4	goto (MBR1)	Despachar a siguiente instrucción
iflt1	MAR = SP = SP – 1; rd	Leer segunda palabra de la pila
iflt2	OPC = TOS	Guardar TOS en OPC temporalmente
iflt3	TOS = MDR	Poner nuevo tope de pila en TOS
iflt4	N = OPC; if (N) goto T; else goto F	Ramificar según bit N

Figura 4-30. El microprograma del Mic-2 (parte 1 de 2).

Mnemónico	Operaciones	Comentarios
ifeq1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
ifeq2	OPC = TOS	Guardar TOS en OPC temporalmente
ifeq3	TOS = MDR	Poner nuevo tope de pila en TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Ramificar según bit Z
if_icmpeq1	MAR = SP = SP – 1; rd	Leer 2a. palabra de la pila
if_icmpeq2	MAR = SP = SP – 1	Ajustar MAR para leer nuevo tope de pila
if_icmpeq3	H = MDR; rd	Copiar 2a. palabra de la pila en H
if_icmpeq4	OPC = TOS	Guardar TOS en OPC temporalmente
if_icmpeq5	TOS = MDR	Poner nuevo tope de pila en TOS
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	Si 2 pal. en tope iguales, ir a T; si no, a F
T	H = PC – 1; goto goto2	Igual que goto1
F	H = MBR2	Tocar bytes de MBR2 que se desecharán
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Poner dir. de apunt. a método en MAR
invokevirtual2	OPC = PC	Guardar PC de retorno en OPC
invokevirtual3	PC = MDR	Hacer que PC apunte a 1er. byte de método
invokevirtual4	TOS = SP – MBR2U	TOS = dirección de OBJREF - 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = dirección de OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Sobreescribir OBJREF con apunt. enlace
invokevirtual7	MAR = SP = MDR	Asignar a SP, MAR, pos. p/guardar viejo PC
invokevirtual8	MDR = OPC; wr	Prepararse para guardar viejo PC
invokevirtual9	MAR = SP = SP + 1	Inc. SP p/que apunte a donde se guardará viejo LV
invokevirtual10	MDR = LV; wr	Guardar viejo LV
invokevirtual11	LV = TOS; goto (MBR1)	Hacer que LV apunte a parámetro 0
ireturn1	MAR = SP = LV; rd	Restab. SP, MAR, p/leer apunt. de enlace
ireturn2		Esperar apuntador de enlace
ireturn3	LV = MAR = MDR; rd	Asignar apunt. enlace a LV, MAR; leer viejo PC
ireturn4	MAR = LV + 1	Hacer que MAR apunte a viejo LV; leer viejo LV
ireturn5	PC + MDR; rd	Restaurar PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Restaurar LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Guardar valor de retorno en tope de pila original

Figura 4-30. El microprograma del Mic-2 (parte 2 de 2).

los procesen y luego escribe los resultados de vuelta en los registros. Fuera de la IFU, casi no hay paralelismo. La adición de paralelismo es una gran oportunidad.

Como dijimos antes, el ciclo de reloj está limitado por el tiempo que se requiere para que las señales se propaguen por la trayectoria de datos. La figura 4-3 muestra un desglose del retardo a través de los diversos componentes durante cada ciclo. El ciclo de la trayectoria de datos real tiene tres componentes principales:

1. El tiempo que toma colocar los registros seleccionados en los buses A y B.
2. El tiempo que tardan la ALU y el desplazador en realizar su trabajo.
3. El tiempo que tardan los resultados en regresar a los registros y guardarse.

En la figura 4-31 mostramos una nueva arquitectura de tres buses que incluye la IFU pero tiene tres latches (registros) adicionales, cada uno insertado en medio de un bus. Se

escribe en los latches en cada ciclo. De hecho, los latches dividen la trayectoria de datos en partes discretas que ahora pueden operar de forma independiente. Llamaremos a este modelo **Mic-3**, o modelo **con filas de procesamiento**.

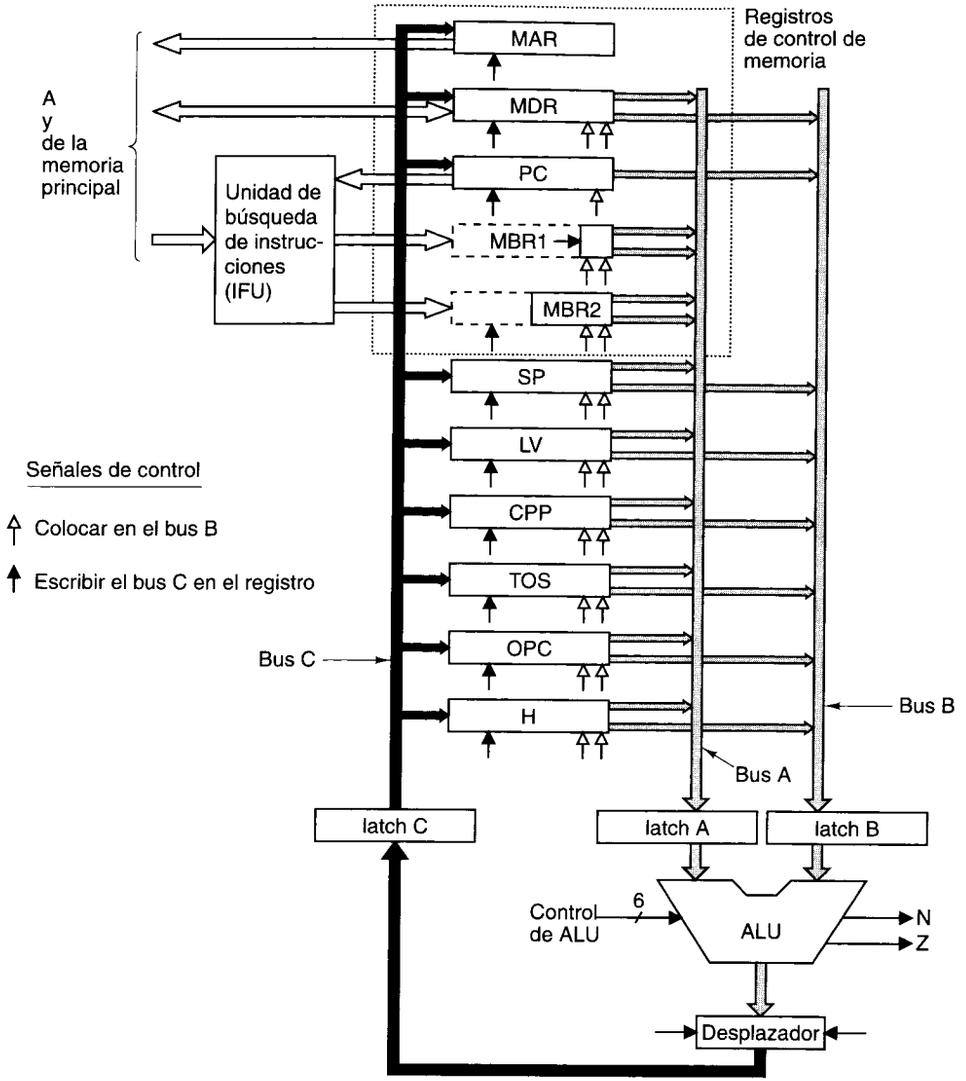


Figura 4-31. La trayectoria de datos de tres buses empleado en el Mic-3.

¿Cómo es posible que estos registros adicionales sirvan de algo? Ahora se requieren tres ciclos de reloj para usar la trayectoria de datos: uno para cargar los latches A y B, uno para

operar la ALU y el desplazador, además de cargar el latch C, y uno para almacenar el contenido del latch C de vuelta en los registros. ¿Acaso estamos locos? (*Sugerencia:* No.) La ventaja de insertar los latches es doble:

1. Podemos acelerar el reloj porque el retraso máximo ahora es más corto.
2. Podemos usar todas las partes de la trayectoria de datos durante cada ciclo.

Al descomponer la trayectoria de datos en tres partes, redujimos el retardo máximo, y el resultado es que la frecuencia del reloj puede ser más alta. Supongamos que al dividir el ciclo de la trayectoria de datos en tres intervalos de tiempo hacemos que cada uno tenga una duración de aproximadamente 1/3 de la original, así que podemos triplicar la velocidad del reloj. (Esto no es del todo realista porque también añadimos dos registros más a la trayectoria de datos, pero sirve como primera aproximación.)

Puesto que hemos estado suponiendo que todas las lecturas y escrituras de la memoria pueden satisfacerse con el contenido de la memoria caché de nivel 1, y esta caché está hecha con el mismo material que los registros, seguiremos suponiendo que una operación de memoria tarda un ciclo. En la práctica, empero, esto podría ser difícil de lograr.

El segundo punto tiene que ver con el rendimiento más que con la rapidez de una instrucción individual. En el Mic-2, durante la primera y la tercera partes de cada ciclo de reloj, la ALU está ociosa. Si dividimos la trayectoria de datos en tres partes, podremos usar la ALU en cada ciclo y obtener hasta tres veces más trabajo de la máquina.

Veamos ahora cómo funciona la trayectoria de datos del Mic-3. Antes de comenzar, necesitamos una notación para manejar los latches. La obvia es llamar a los latches A, B y C y tratarlos como registros, teniendo presentes las restricciones de la trayectoria de datos. La figura 4-32 muestra un ejemplo de secuencia de código, la implementación de SWAP para el Mic-2.

Mnemónico	Operaciones	Comentarios
swap1	MAR = SP – 1; rd	Leer 2a. palabra de la pila; asignar SP a MAR
swap2	MAR = SP	Prepararse para escribir nueva 2a. palabra
swap3	H = MDR; wr	Guardar nuevo TOS; escribir 2a. pal. en la pila
swap4	MDR = TOS	Copiar TOS viejo en MDR
swap5	MAR = SP – 1; wr	Escribir TOS viejo en 2o. lugar de la pila
swap6	TOS = H; goto (MBR1)	Actualizar TOS

Figura 4-32. El ódigo de SWAP en el Mic-2.

Reimplementemos ahora esta secuencia en el Mic-3. Recuerde que la operación de la trayectoria de datos ahora requiere tres ciclos: uno para cargar A y B, uno para efectuar la operación y cargar C, y uno para escribir los resultados de vuelta en los registros. Llamaremos a cada una de estas partes un **micropaso**.

La implementación de SWAP para el Mic-3 se muestra en la figura 4-33. En el ciclo 1 iniciamos en swap1 copiando SP en B. No importa lo que suceda en A porque para restar 1 a B ENA se invalida (vea la figura 4-2). Para simplificar, no mostraremos asignaciones que no

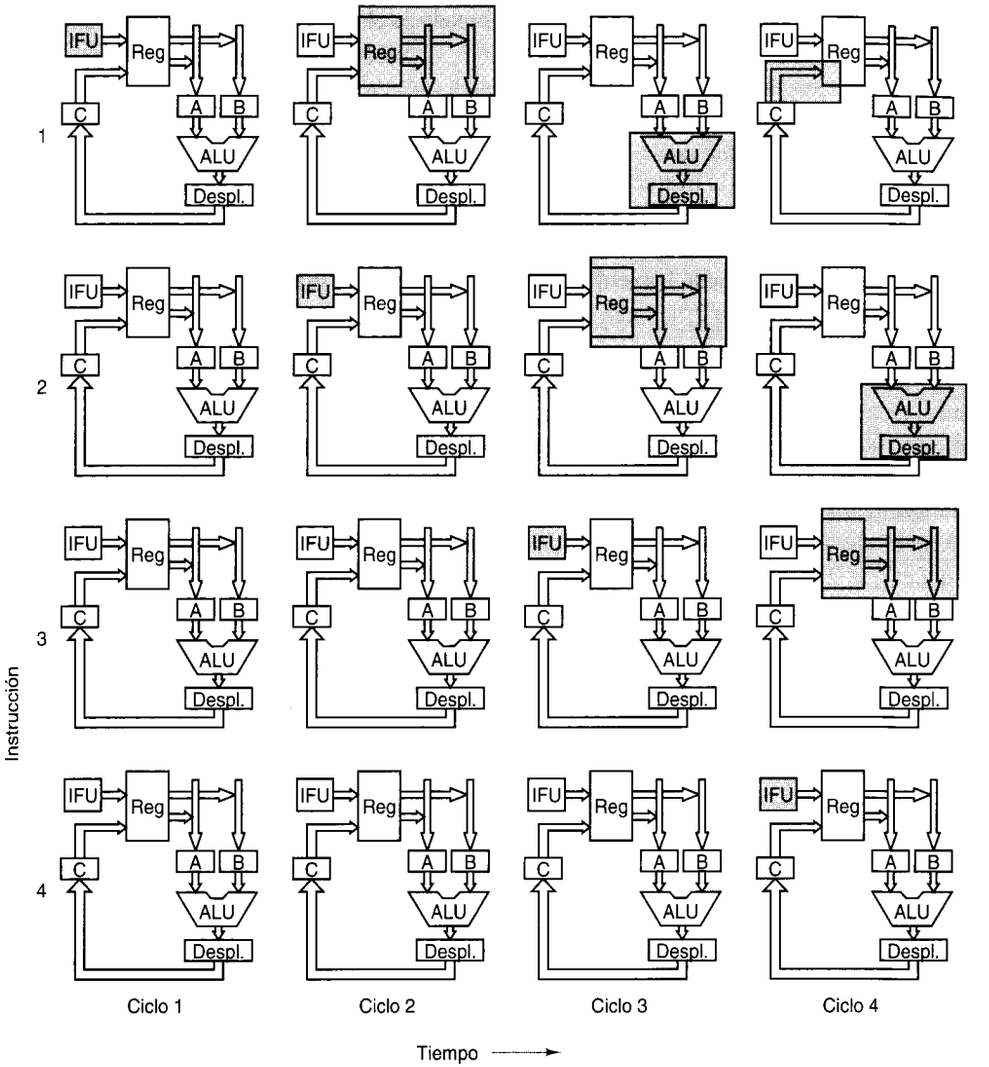
se usen. En el ciclo 2 realizamos la resta. En el ciclo 3 el resultado se almacena en MAR y se inicia la operación de lectura al final del ciclo 3 (una vez que se ha almacenado MAR). Puesto que las lecturas de memoria ahora tardan un ciclo, ésta no se completará sino hasta el final del ciclo 4, lo que se indica con la asignación a MDR en el ciclo 4. El valor en MDR no puede leerse antes del ciclo 5.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cic	MAR=SP–1;rd	MAR=SP	H=MDR;wr	MDR=TOS	MAR=SP–1wr	TOS=H;goto (MBR1)
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figura 4-33. Implementación de SWAP en el Mic-3.

Volvamos ahora al ciclo 2. Ya podemos comenzar a dividir swap2 en micropasos e iniciarlos. En el ciclo 2 podemos copiar SP en B, luego pasarlo por la ALU en el ciclo 3 y por último guardarlo en MAR en el ciclo 4. Hasta aquí todo va bien. Debe ser obvio que si podemos seguir a este ritmo, iniciando una nueva microinstrucción en cada ciclo, habremos triplicado la velocidad de la máquina. Esta ganancia se debe al hecho de que podemos emitir una nueva microinstrucción en cada ciclo de reloj, y el Mic-3 tiene tres veces más ciclos de reloj por segundo que el Mic-2. De hecho, hemos construido una CPU con conductos.

Lo malo es que nos topamos con un obstáculo en el ciclo 3. Nos gustaría comenzar a trabajar con swap3, pero lo primero que hace es pasar MDR por la ALU, y MDR no se habrá cargado desde la memoria antes del inicio del ciclo 5. La situación de que un micropaso no puede iniciarse porque está esperando un resultado que un micropaso anterior todavía no ha producido se llama **dependencia verdadera** o **dependencia RAW**. Es común llamar **peligros** a las dependencias. RAW significa "lectura después de escritura" (*Read After Write*) e indica que un micropaso quiere leer un registro que todavía no se escribe. La única acción sensata aquí es retrasar el inicio de swap3 hasta que MDR esté disponible, en el ciclo 5. Detenerse para esperar un valor necesario se denomina **parar** (*stalling*). Después, podremos seguir iniciando microinstrucciones en cada ciclo porque no hay más dependencias, aunque swap6 apenas lo logra, ya que lee H en el ciclo después de que swap3 lo escribe. Si swap5 hubiera tratado de leer H, habría parado durante un ciclo.



Figur a 4-34. Ilustración gráfica del funcionamiento de una fila de procesamiento.

Aunque el programa Mic-3 tarda más ciclos que el Mic-2, se ejecuta más rápidamente. Si llamamos al tiempo de ciclo de Mic-3 ΔT ns, entonces el Mic-3 requiere 11 ΔT ns para ejecutar SWAP. En cambio, el Mic-2 tarda 6 ciclos de 3 ΔT cada uno, para un total de 18 ΔT. Los conductos han hecho más rápida la máquina, aunque tuvimos que parar una vez para evitar una dependencia.

El uso de filas de procesamiento es una técnica fundamental en todas las CPU modernas, así que es importante entenderla bien. En la figura 4-34 vemos a la trayectoria de datos de la figura 4-31 ilustrada gráficamente como conducto o interconexión. La primera columna representa lo que sucede durante el ciclo 1, la segunda columna representa el ciclo 2, y así

(suponiendo que no hay paros). La región sombreada en el ciclo 1 para la instrucción 1 indica que la IFU está ocupada trayendo la instrucción 1. Un tic del reloj después, durante el ciclo 2, los registros que la instrucción 1 necesita se están cargando en los latches A y B mientras la IFU está ocupada trayendo la instrucción 2, lo que también se indica con los dos rectángulos sombreados en el ciclo 2.

Durante el ciclo 3 la instrucción 1 está usando la ALU y el desplazador para efectuar su operación, los latches A y B se están cargando para la instrucción 2 y se está trayendo la instrucción 3. Por último, durante el ciclo 4, se está trabajando con cuatro instrucciones al mismo tiempo. Se están guardando los resultados de la instrucción 1, se está efectuando el trabajo de ALU para la instrucción 2, se están cargando los latches A y B para la instrucción 3 y se está trayendo la instrucción 4.

Si hubiéramos mostrado el ciclo 5 y los subsecuentes, el patrón habría sido el mismo que en el ciclo 4: las cuatro partes de la trayectoria de datos que pueden operar de forma independiente estarían haciéndolo. Este diseño representa una fila de procesamiento de cuatro etapas: para traer instrucciones, accesar operandos, hacer operaciones de ALU y para la escritura de resultados en los registros. Esto es similar a la fila de procesamiento de la figura 2-4(a), excepto que no tiene la etapa de decodificación. El punto importante que debemos captar aquí es que si bien una sola instrucción tarda cuatro ciclos de reloj en ejecutarse, en cada uno de éstos se inicia una nueva instrucción y otra vieja termina.

Otra forma de ver la figura 4-34 es seguir cada instrucción horizontalmente. Para la instrucción 1, en el ciclo 1 la IFU está trabajando en ella. En el ciclo 2, sus registros se están colocando en los buses A y B. En el ciclo 3 la ALU y el desplazador están trabajando para ella. Por último, en el ciclo 4, sus resultados se están guardando de vuelta en los registros. Lo importante aquí es que se dispone de cuatro secciones del hardware, y durante cada ciclo una instrucción dada utiliza sólo una de ellas, con lo que las otras secciones quedan libres para instrucciones diferentes.

Una analogía útil de nuestro diseño con filas de procesamiento es una línea de ensamble de una fábrica que arma automóviles. Para abstraer los aspectos esenciales de este modelo, imagine que se golpea un gran gong cada minuto, y en ese instante todos los automóviles avanzan a la siguiente estación de la línea. En cada estación, los trabajadores realizan alguna operación en el automóvil que en ese momento está frente a ellos, como poner el volante o instalar los frenos. En cada golpe del gong (un ciclo) se inyecta un nuevo automóvil al principio de la línea de ensamble, y un automóvil terminado sale por el otro extremo. Así, aunque armar un automóvil podría tardar cientos de ciclos, en cada ciclo se termina un automóvil completo. La fábrica puede producir un automóvil por minuto, sin importar cuánto tarde realmente el armado de un automóvil. Ésta es la potencia de las filas de procesamiento, y aplica tanto a las CPU como a las fábricas de automóviles.

4.4.5 Una fila de procesamiento de siete etapas: el Mic-4

Un punto que hemos pasado por alto es el hecho de que cada microinstrucción selecciona su propia sucesora. Casi todas se limitan a seleccionar la que sigue en la secuencia, pero la última, digamos swap6, a menudo realiza una ramificación multivías, lo que frena la fila de

procesamiento porque es imposible efectuar prebúsqueda después de ella. Necesitamos una mejor forma de atacar este problema.

Nuestra siguiente (y última) microarquitectura es el Mic-4. Sus partes principales se ilustran en la figura 4-35, pero se han omitido muchos detalles por claridad. Al igual que el Mic-3, el Mic-4 tiene una IFU que prebusca palabras de la memoria y mantiene los diversos MBR.

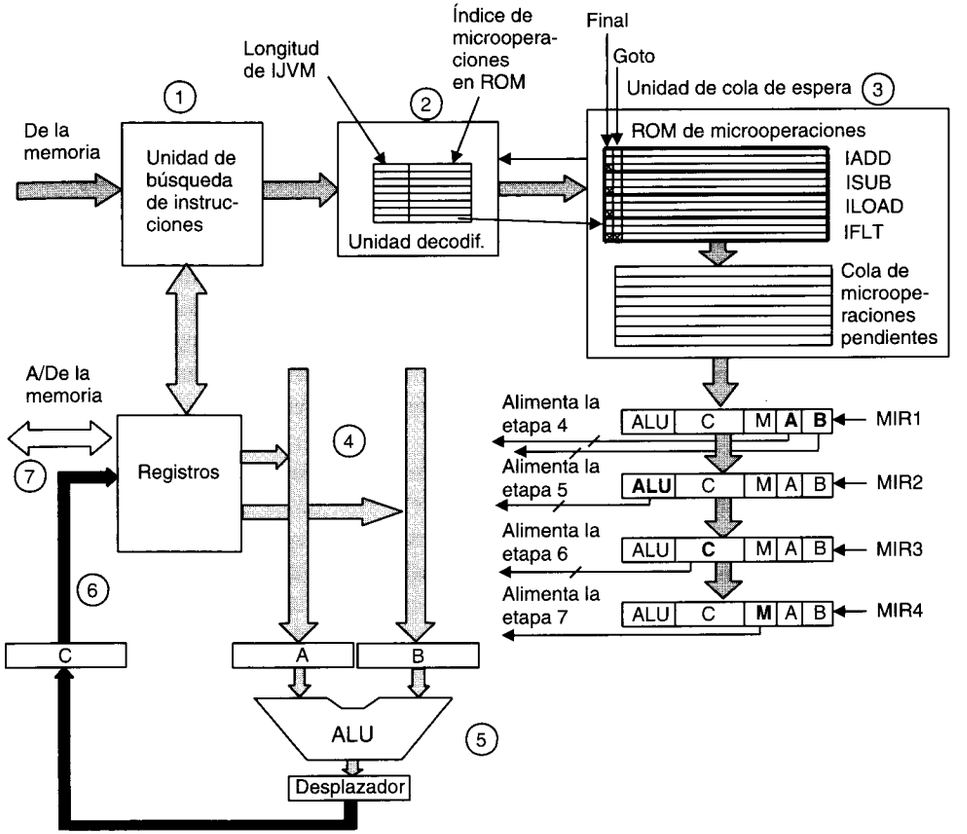


Figura 4-35. Principales componentes del Mic-4.

La IFU también alimenta el flujo de bytes entrante a un nuevo componente, la **unidad decodificadora**, que tiene una ROM interna indizada por el código de operación IJVM. Cada entrada (renglón) tiene dos partes: la longitud de esa instrucción IJVM y un índice para otra ROM, la ROM de microoperaciones. La longitud de la instrucción IJVM sirve para que la unidad decodificadora pueda analizar el flujo de bytes entrante y dividirlo en instrucciones, de modo que siempre sepa cuáles bytes son códigos de operación y cuáles son operandos. Si la longitud de la instrucción actual es 1 byte (por ejemplo, POP), la unidad decodificadora sabe que el siguiente byte es un código de operación. En cambio, si la longitud de la instrucción actual es de 2 bytes, la unidad decodificadora sabe que el siguiente byte es un operando,

seguido de inmediato por otro código de operación. Cuando se detecta el prefijo WIDE, el siguiente byte se transforma en un código de operación ancho especial; por ejemplo WIDE + ILOAD se convierte en WIDE_ILOAD.

La unidad decodificadora envía el índice para la ROM de microoperaciones, que encontró en su tabla, al siguiente componente, la **unidad de cola de espera**. Esta unidad contiene algo de lógica y dos tablas internas, una en ROM y una en RAM. La ROM contiene el microprograma, y cada instrucción IJVM tiene cierto número de entradas consecutivas llamadas **microoperaciones**. Las entradas deben estar en orden, por lo que no se permiten trucos como que wide_iload2 salte a iload2 como en Mic-2. Cada secuencia IJVM debe detallarse totalmente, duplicando secuencias en algunos casos.

Las microoperaciones son similares a las microinstrucciones de la figura 4-5 excepto que no existen los campos NEXT_ADDRESS ni JAM, y se necesita un nuevo campo codificado para especificar la entrada del bus A. También se incluyen dos nuevos bits: Final y Goto. El bit Final se enciende en la última microoperación de cada secuencia de microoperaciones IJVM para marcarla. El bit Goto se enciende para marcar las microoperaciones que son microrramificaciones condicionales. Su formato es diferente del de las microoperaciones normales, pues consiste en los bits JAM y un índice para la ROM de microoperaciones. Las microinstrucciones que antes hacían algo con la trayectoria de datos y también efectuaban una microrramificación condicional (como iflt4) ahora tienen que dividirse en dos microoperaciones.

La unidad de cola de espera funciona como sigue. Recibe de la unidad decodificadora un índice para la ROM de microoperaciones. Con ese índice, busca la microoperación en la ROM y la copia en una cola interna. Luego copia en la cola la siguiente microoperación, y también la que sigue a ésa. Sigue haciendo esto hasta que llega a una que tiene el bit Final encendido (en 1). Copia esa microoperación y se detiene. Suponiendo que no encontró una microoperación con el bit Goto encendido y todavía tenga mucho espacio en la cola, la unidad de cola de espera enviará una señal de acuse a la unidad decodificadora. Cuando la unidad decodificadora detecta este acuse, envía el índice de la siguiente instrucción IJVM a la unidad de cola de espera.

De este modo, la sucesión de instrucciones IJVM en la memoria se convierte en última instancia en una sucesión de microoperaciones en una cola. Estas microoperaciones alimentan a los MIR, que envían las señales que controlan la trayectoria de datos. Sin embargo, ahora hay otro factor que es preciso considerar: los campos de cada microoperación no están activos al mismo tiempo. Los campos A y B están activos durante el primer ciclo. El campo ALU está activo durante el segundo ciclo, el campo C está activo durante el tercer ciclo, y las operaciones de memoria, si se requieren, se efectúan en el cuarto ciclo.

Para que esto funcione correctamente, hemos introducido cuatro MIR independientes en la figura 4-35. Al principio de cada ciclo de reloj (el tiempo Δw en la figura 4-3), MIR3 se copia en MIR4, MIR2 se copia en MIR3, MIR1 se copia en MIR2 y MIR1 se carga con una nueva microoperación de la cola de microoperaciones. Luego cada MIR saca sus señales de control, pero sólo se usan algunas de ellas. Los campos A y B de MIR1 sirven para seleccionar los registros que alimentan los latches A y B, pero el campo ALU de MIR1 no se usa y no está conectado a ninguna otra cosa en la trayectoria de datos.

Un ciclo de reloj después, esta microoperación ha avanzado a MIR2 y los registros que seleccionó ya están a salvo en los latches A y B, esperando nuevas aventuras. Su campo ALU se usa ahora para alimentar la ALU. En el siguiente ciclo, el campo C escribirá los resultados de vuelta en los registros. Después, pasará a MIR4 e iniciará cualesquier operaciones de memoria que necesite utilizando el MAR que ya se cargó (y MDR, si se trata de una escritura).

Hay un último aspecto del Mic-4 que debemos tratar aquí: las microrramificaciones. Algunas instrucciones IJVM, como IFLT, necesitan ramificar condicionalmente con base en, digamos, el bit N. Cuando ocurre una microrramificación, la fila de procesamiento no puede continuar. Para resolver este problema, hemos añadido el bit Goto a cada microoperación. Cuando la unidad de cola de espera encuentra una microoperación con este bit encendido al copiarla en la cola, se da cuenta de que hay problemas en el futuro y no envía un acuse a la unidad decodificadora. El resultado es que la máquina se para en este punto hasta que se resuelve la microrramificación.

Es concebible que algunas instrucciones IJVM posteriores a la ramificación ya se hayan alimentado a la unidad decodificadora (pero no a la de cola de espera), ya que no devuelve una señal de acuse (o sea, de continuar) cuando encuentra una microoperación que tiene el bit Goto encendido. Se requieren hardware y mecanismos especiales para arreglar las cosas y volver al carril, pero este tema rebasa el alcance del presente libro. Cuando Edsger Dijkstra escribió su famosa carta "GOTO Statement Considered Harmful" ("El enunciado GOTO es perjudicial") (Dijkstra, 1968a), no sabía cuánta razón tenía.

Hemos llegado muy lejos desde el Mic-1. Éste era un sistema de hardware muy sencillo, con casi todo el control en software. El Mic-4 es un diseño con uso intensivo de filas de procesamiento, con siete etapas y hardware mucho más complejo. La fila de procesamiento se muestra de forma esquemática en la figura 4-36, donde los números en círculos corresponden a los de la figura 4-35. El Mic-4 prebusca automáticamente un flujo de bytes de la memoria, los decodifica como instrucciones IJVM, convierte éstas en una sucesión de microoperaciones mediante una ROM, y las encola para usarlas conforme se vayan necesitando. Las primeras tres etapas del conducto se pueden vincular con el reloj de la trayectoria de datos si se desea, pero no siempre habrá trabajo que efectuar. Por ejemplo, la IFU no puede alimentar un nuevo código de operación IJVM a la unidad decodificadora en cada ciclo porque las instrucciones IJVM tardan varios ciclos en ejecutarse y la cola se desbordaría rápidamente.

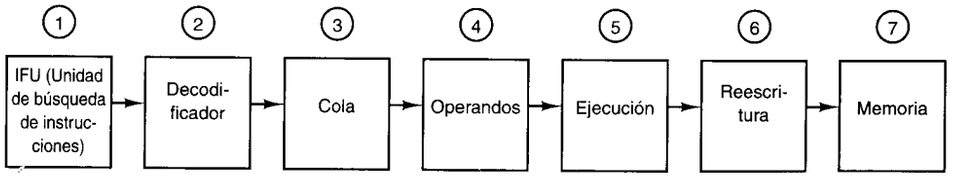


Figura 4-36. Fila de procesamiento del Mic-4.

En cada ciclo de reloj los MIR se desplazan hacia adelante y la microoperación que está al fondo de la cola se copia en MIR1 para iniciar su ejecución. Luego las señales de control de

los cuatro MIR se distribuyen por la trayectoria de datos y hacen que se efectúen ciertas acciones. Cada MIR controla una parte distinta de la trayectoria de datos y por tanto diferentes micropasos.

En este diseño tenemos una CPU con gran uso de filas de procesamiento que permite pasos individuales muy cortos y por tanto una frecuencia de reloj alta. Muchas CPU se diseñan casi de la misma manera, sobre todo las que tienen que implementar un conjunto de instrucciones más viejo (CISC). Por ejemplo, la implementación del Pentium II es conceptualmente similar al Mic-4 en algunos aspectos, como veremos más adelante en este capítulo.

4.5 MEJORAMIENTO DEL DESEMPEÑO

Todos los fabricantes de computadoras quieren que sus sistemas operen con la mayor rapidez posible. En esta sección examinaremos varias técnicas avanzadas que se están investigando con miras a mejorar el desempeño de los sistemas (primordialmente de la CPU y la memoria). Debido a la naturaleza tan competitiva de la industria de las computadoras, el retraso entre que surgen ideas nuevas para hacer más rápida una computadora y que se incorporan en productos es sorprendentemente corto. Por tanto, casi todas las ideas que veremos aquí ya se están usando en una amplia variedad de productos.

Las ideas que trataremos se pueden clasificar a grandes rasgos en dos categorías: mejoras en la implementación y mejoras en la arquitectura. Las mejoras en la implementación son formas de construir una nueva CPU o memoria para hacer que el sistema opere más rápidamente sin cambiar la arquitectura. Modificar la implementación sin alterar la arquitectura implica que los programas viejos podrán ejecutarse en la nueva máquina, lo cual es un atractivo que eleva las ventas. Una forma de mejorar la implementación es usar un reloj más rápido, pero no es el único camino. Las mejoras en el desempeño logradas del 80386 al 80486, al Pentium, al Pentium Pro, hasta el Pentium II se deben a mejores implementaciones, pues la arquitectura se ha mantenido básicamente sin cambios en todos ellos.

Algunos tipos de mejoras sólo pueden lograrse modificando la arquitectura. A veces tales cambios son incrementales, como añadir nuevas instrucciones o registros, de modo que los programas viejos puedan seguir funcionando en los modelos nuevos. En este caso, para obtener el desempeño óptimo es preciso modificar el software, o al menos recompilarlo con un compilador nuevo que aproveche las nuevas características.

No obstante, cada cierto número de décadas los diseñadores se dan cuenta de que la vieja arquitectura ha dejado de ser útil y que la única forma de progresar es comenzar otra vez desde el principio. La revolución RISC de la década de los ochenta fue uno de esos avances; otro se está gestando actualmente. Examinaremos un ejemplo (el Intel IA-64) en el capítulo 5.

En el resto de esta sección estudiaremos cuatro técnicas para mejorar el desempeño de la CPU. Comenzaremos con tres mejoras de la implementación bien establecidas y luego pasaremos a uno que necesita un poco de apoyo arquitectónico para trabajar óptimamente. Las técnicas en cuestión son memoria caché, predicción de ramas, ejecución no en orden con cambio de nombre de registros y ejecución especulativa.

4.5.1 Memoria caché

Uno de los desafíos del diseño de computadoras en toda su historia ha sido proporcionar un sistema de memoria capaz de alimentar operandos al procesador tan rápidamente como éste puede procesarlos. El considerable aumento en la velocidad de los procesadores a últimas fechas no ha ido acompañado de un aumento correspondiente en la velocidad de las memorias. En comparación con las CPU, las memorias se han estado volviendo más lentas desde hace décadas. Dada la enorme importancia de la memoria primaria, esta situación ha limitado severamente el desarrollo de sistemas de alto desempeño, y ha estimulado investigaciones sobre formas de resolver el problema de memorias que son mucho más lentas que las CPU, problema que, en términos relativos, empeora cada año.

Los procesadores modernos someten a los sistemas de memoria a demandas abrumadoras, tanto en términos de latencia (lo que tarda en proporcionar un operando) como de ancho de banda (la cantidad de datos que se proporciona por unidad de tiempo). Lamentablemente, estos dos aspectos de un sistema de memoria son en gran medida opuestos. Muchas técnicas para aumentar el ancho de banda lo logran únicamente aumentando la latencia. Por ejemplo, las técnicas de filas de procesamiento que usamos en el Mic-3 se pueden aplicar a un sistema de memoria, manejando de forma eficiente múltiples solicitudes de memoria que se traslapan. Lo malo es que, como sucede con el Mic-3, el resultado es una mayor latencia para operaciones de memoria individuales. A medida que aumenta la velocidad de reloj de los procesadores, se vuelve más difícil crear un sistema de memoria capaz de proporcionar operandos en uno o dos ciclos de reloj.

Una forma de atacar este problema es incluir cachés. Como vimos en la sección 2.2.5, una caché contiene las palabras de memoria que se usaron más recientemente en una memoria pequeña y rápida, con lo que se agiliza el acceso a ellas. Si un porcentaje suficientemente grande de las palabras de memoria que se necesitan están en la caché, la latencia de memoria efectiva se puede reducir enormemente.

Una de las técnicas más eficaces para mejorar tanto el ancho de banda como la latencia se basa en el empleo de múltiples cachés. Una técnica fundamental que resulta muy eficaz es introducir cachés individuales para instrucciones y para datos, lo que se conoce como **caché dividida**. Esto tiene varias ventajas. Primera, las operaciones de memoria se pueden iniciar de forma independiente en cada caché, lo que duplica efectivamente el ancho de banda del sistema de memoria. Ésta es la razón por la que conviene tener dos puertos de memoria distintos, como hicimos en el Mic-1: cada puerto tiene su propia caché. Cabe señalar que cada caché tiene acceso independiente a la memoria principal.

Hoy día muchos sistemas de memoria son más complicados, y una caché adicional, llamada **caché de nivel 2**, podría residir entre las cachés de datos e instrucciones y la memoria principal. De hecho, podría haber tres o más niveles de caché si se requieren sistemas de memoria más sofisticados. En la figura 4-37 vemos un sistema con tres niveles de caché. El chip de la CPU mismo contiene una pequeña caché de instrucciones y una pequeña caché de datos, por lo regular de 16 KB a 64 KB. Luego está la caché de nivel 2, que no está en el chip de la CPU pero podría estar incluida en su paquete, junto al chip y conectada a él con una trayectoria de alta velocidad. Esta caché suele estar unificada (es decir, contiene una mezcla

de datos e instrucciones). Un tamaño típico para la caché L2 es de 512 KB a 1 MB. La caché de tercer nivel está en la tarjeta del procesador y consiste en unos cuantos megabytes de SRAM, que es mucho más rápida que la memoria principal DRAM. Las cachés generalmente son inclusivas: todo el contenido de la caché de nivel 1 está en la caché de nivel 2, y todo el contenido de la caché de nivel 2 está en la caché de nivel 3.

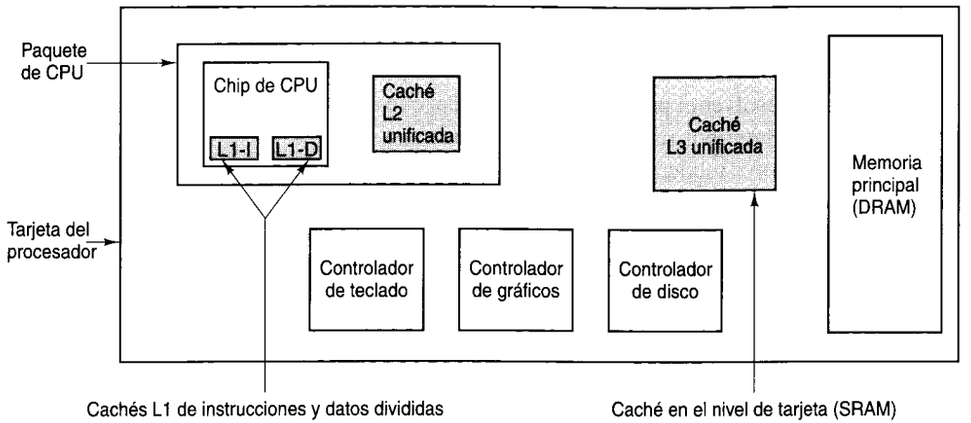


Figura 4-37. Sistema con tres niveles de caché.

Las cachés dependen de dos tipos de localidad de direcciones para lograr su objetivo. **Localidad espacial** es la observación de que es probable que en el futuro cercano se acceda a localidades de memoria con direcciones numéricamente similares a una localidad de memoria a la que se accedió recientemente. Las cachés aprovechan esta propiedad trayendo más datos de los que se solicitaron, con la expectativa de anticipar solicitudes futuras. Ocurre **localidad temporal** cuando se vuelve a acceder a localidades de memoria a las que se accedió recientemente. Esto puede ocurrir, por ejemplo, con localidades de memoria cercanas al tope de la pila, o instrucciones dentro de un ciclo. La localidad temporal se aprovecha primordialmente en los diseños de caché al tomar la decisión de qué debe desecharse cuando no se encuentra algo en ella (fallo de caché). Muchos algoritmos de reemplazo de caché aprovechan la localidad temporal desechando las entradas a las que no se ha accedido recientemente.

Todas las cachés usan el modelo que sigue. La memoria principal se divide en bloques de tamaño fijo llamados **líneas de caché**. Una línea de caché por lo regular consta de 4 a 64 bytes consecutivos. Las líneas se numeran consecutivamente a partir del 0, de modo que si el tamaño de línea es de 32 bytes la línea 0 comprende los bytes del 0 al 31, la línea 1 comprende los bytes del 32 al 63, etc. En cualquier instante dado, algunas líneas están en la caché. Cuando se hace referencia a la memoria, el circuito controlador de la caché verifica si la palabra a la que se hizo referencia está actualmente en la caché. Si es así, puede usar el valor que está ahí y evitarse un viaje a la memoria principal. Si la palabra no está ahí, se elimina una línea de la caché y la línea requerida se busca de la memoria o de una caché de nivel más

bajo para reemplazarla. Existen muchas variaciones de este esquema, pero todas se basan en la idea de mantener las líneas más usadas en la caché el mayor tiempo posible, a fin de maximizar el número de referencias a la memoria que se pueden satisfacer con la caché.

Cachés con mapeo directo

La caché más sencilla es la llamada **caché con mapeo directo**. En la figura 4-38(a) se muestra un ejemplo de caché con mapeo directo de un solo nivel. Esta caché contiene 2048 entradas. Cada entrada (renglón) de la caché puede contener exactamente una línea de caché de la memoria principal. Con un tamaño de línea de caché de 32 bytes (para este ejemplo), la caché puede contener 64 KB. Cada entrada de caché consta de tres partes:

1. El bit de validez indica si la entrada contiene o no datos válidos. Cuando el sistema se arranca (inicia), todas las entradas se marcan como no válidas.
2. El campo Etiqueta (*tag*) consiste en un valor único de 16 bits que identifica la línea de memoria de la cual provienen los datos.
3. El campo Datos contiene una copia de los datos de la memoria. Este campo contiene una línea de caché de 32 bytes.

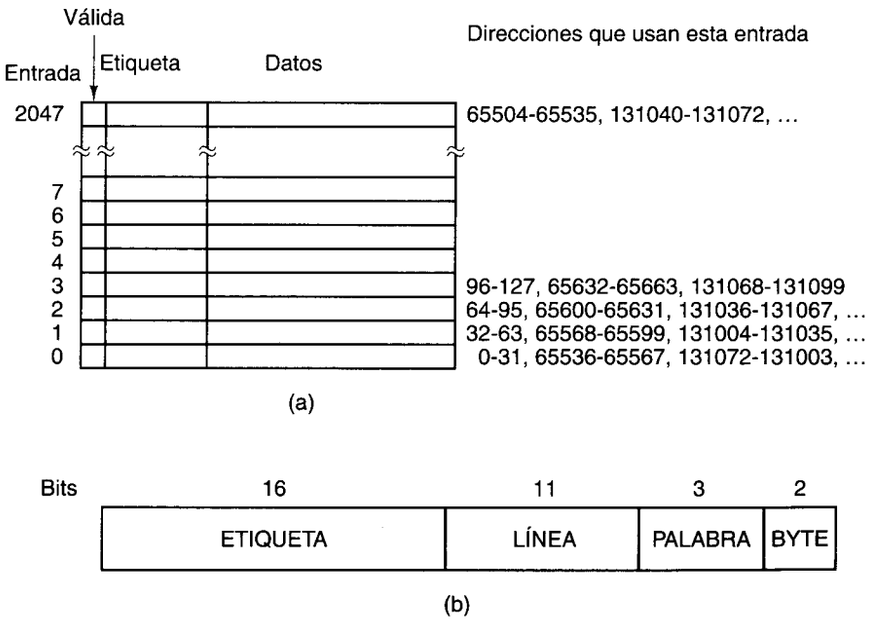


Figura 4-38. (a) Caché con mapeo directo. (b) Dirección virtual de 32 bits.

En una caché con mapeo directo, una palabra de memoria dada se puede almacenar en un y sólo un lugar dentro de la caché. Dada una dirección de memoria, sólo hay un lugar en el

que se puede buscar en la caché. Si no está ahí, entonces no está en la caché. Para almacenar y recuperar datos de la caché, la dirección se divide en cuatro componentes, como se muestra en la figura 4-38(b):

1. El campo ETIQUETA corresponde a los bits Etiqueta almacenados en una entrada de caché.
2. El campo LÍNEA indica cuál caché de entrada contiene los datos correspondientes, si están presentes.
3. El campo PALABRA indica a cuál palabra se hace referencia dentro de la línea.
4. El campo BYTE generalmente no se usa, pero si sólo se solicita un byte indica cuál byte dentro de la palabra se necesita. En el caso de una caché que sólo proporciona palabras de 32 bits, este campo siempre es 0.

Cuando la CPU produce una dirección de memoria, el hardware extrae los 11 bits LÍNEA de la dirección y los usa como índice dentro de la caché para encontrar una de las 2048 entradas. Si esa entrada es válida, se comparan el campo ETIQUETA de la dirección de memoria y el campo Etiqueta de la entrada de caché; si coinciden, la entrada de caché contiene la palabra que se solicitó, situación que se denomina **acierto de caché**. Si se acierta, la palabra que se desea leer se puede tomar de la caché, y ya no es necesario ir a la memoria. Sólo se extrae de la entrada de caché la palabra que se necesita; el resto de la entrada no se usa. Si la entrada de caché no es válida o si las etiquetas no coinciden, la entrada requerida no está presente en el caché, situación que se denomina **fallo de caché**. En este caso, la línea de caché de 32 bytes se trae de la memoria y se guarda en la entrada de caché, sustituyendo lo que estaba ahí. Sin embargo, si la entrada que estaba antes en la caché se modificó después de cargarse, deberá volver a escribirse en la memoria principal antes de desecharse.

A pesar de la complejidad de la decisión, el acceso a una palabra requerida puede ser notablemente rápido. Tan pronto como se conoce la dirección, se conoce la ubicación exacta de la palabra *si está presente en la caché*. Esto implica que es posible leer la palabra de la caché y proporcionarla al procesador al mismo tiempo que se está determinando si era la palabra correcta (comparando etiquetas). Así, el procesador realmente recibe una palabra de la caché simultáneamente, o incluso antes de saber si la palabra es la que se solicitó.

Este esquema de correspondencia coloca líneas de memoria consecutivas en entradas de caché consecutivas. De hecho, se pueden almacenar hasta 64K bytes de datos contiguos en la caché. Sin embargo, dos líneas que difieran en sus direcciones por exactamente 64K (65,536 bytes) o cualquier múltiplo entero de ese número no podrán guardarse en la caché al mismo tiempo (porque tienen el mismo valor de LÍNEA). Por ejemplo, si un programa accesa a datos que están en la localidad X y luego ejecuta una instrucción que necesita datos que están en la localidad $X + 65{,}536$ (o cualquier otra localidad dentro de la misma línea), la segunda instrucción hará que la entrada de caché se vuelva a cargar, sobreescribiendo lo que estaba ahí. Si esto sucede con suficiente frecuencia, el funcionamiento de la caché puede ser deficiente. De hecho, el comportamiento de una caché en el peor de los casos, es peor que si no hubiera caché, ya que cada operación de memoria implica leer toda una línea de caché en lugar de una sola palabra.

Las cachés con mapeo directo son las más comunes, y son muy eficaces porque puede hacerse que colisiones como la que acabamos de describir ocurran muy raras veces, o nunca. Por ejemplo, un compilador muy inteligente puede tener en cuenta las colisiones de caché al colocar las instrucciones y los datos en la memoria. Cabe señalar que el caso específico que acabamos de describir no ocurriría en un sistema con cachés de instrucciones y de datos separados, porque las solicitudes en colisión serían atendidas por diferentes cachés. Éste es un segundo beneficio de tener dos cachés en lugar de una: más flexibilidad para manejar patrones de memoria en conflicto.

Cachés asociativas por conjuntos

Como ya mencionamos, muchas líneas de memoria distintas compiten por las mismas ranuras de caché. Si un programa que usa la caché de la figura 4-38(a) usa mucho las palabras que están en las direcciones 0 y 65,536, habrá conflictos constantes, y cada referencia podría expulsar a la otra de la caché. Una solución de este problema es permitir dos o más líneas en cada entrada de caché. Una caché con n posibles entradas para cada dirección se denomina **caché asociativa por conjuntos de n vías**. En la figura 4-39 se ilustra una caché asociativa de cuatro vías.

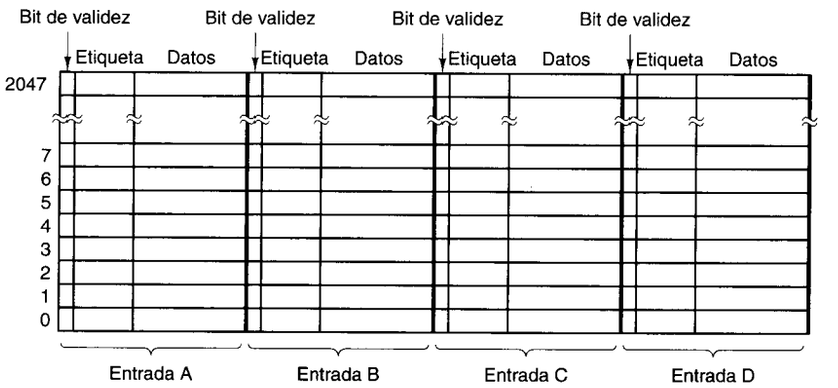


Figura 4-39. Caché asociativa de cuatro vías.

Una caché asociativa por conjuntos es inherentemente más complicada que una de mapeo directo porque, si bien puede calcularse la entrada de caché que debe examinarse a partir de la dirección de memoria a la que se hizo referencia, es preciso examinar un conjunto de n entradas de caché para ver si está presente la línea requerida. No obstante, la experiencia muestra que las cachés de dos y cuatro vías funcionan lo bastante bien como para que valga la pena la adición de estos circuitos.

El uso de una caché asociativa por conjuntos presenta al diseñador una opción. Cuando se debe traer una nueva línea a la caché, ¿cuál de las entradas actuales debe desecharse? Desde luego, la decisión óptima requiere conocer el futuro, pero un algoritmo excelente para casi todas las aplicaciones es el de **menos recientemente utilizada** (**LRU**, *Least Recently*

Used). Este algoritmo mantiene un ordenamiento de cada conjunto de localidades a las que podría accesarse desde una localidad de memoria dada. Siempre que se accede a alguna de las líneas presentes, el algoritmo actualiza la lista, marcando esa entrada como la accesada más recientemente. Cuando se hace necesario reemplazar una entrada, se desecha la que está al final de la lista: aquella a la que se accesó menos recientemente.

Llevado al extremo, es posible tener una caché de 2048 vías que contiene un solo conjunto de 2048 entradas de líneas. Aquí todas las direcciones de memoria corresponden al mismo conjunto único, así que la búsqueda implica comparar la dirección con las 2048 etiquetas de la caché. Cabe señalar que en este caso cada entrada necesitaría lógica de comparación de etiquetas. Puesto que el campo LÍNEA tiene longitud 0, el campo ETIQUETA es toda la dirección excepto los campos PALABRA y BYTE. Además, cuando se reemplaza una línea de caché, las 2048 posiciones son posibles candidatas para el reemplazo. Mantener una lista ordenada de 2048 entradas requiere mucha contabilidad y hace impráctico el reemplazo LRU. (Recuerde que esta lista tiene que actualizarse en cada operación de memoria, no sólo cuando no hay un acierto.) Resulta sorprendente que las cachés con alta asociatividad no mejoran el desempeño mucho más que las de baja asociatividad en la mayor parte de las circunstancias, y en algunos casos tienen un desempeño peor. Por estas razones, casi nunca se usa una asociatividad mayor que de cuatro vías.

Por último, las escrituras presentan un problema especial para las cachés. Cuando un procesador escribe una palabra, y la palabra está en la caché, obviamente se debe actualizar la palabra o bien desecharse la entrada de caché. Casi todos los diseños actualizan la caché. Pero, ¿y la actualización de la copia que está en la memoria principal? Esta operación puede diferirse hasta que la línea de caché esté lista para ser reemplazada por el algoritmo LRU. Esta decisión es difícil, y ninguna de las opciones ofrece una ventaja clara. Actualizar inmediatamente la entrada en la memoria principal se conoce como **escritura a través**. Este enfoque suele ser más fácil de implementar y más confiable, puesto que la memoria siempre está al día; esto es útil, por ejemplo, si ocurre un error y se hace necesario recuperar el estado de la memoria. Lo malo es que casi siempre requiere más tráfico de escritura hacia la memoria, por lo que las implementaciones más complejas tienden a utilizar la alternativa, llamada **escritura diferida** o **reescritura**. Hay un problema relacionado que debemos resolver cuando escribimos: ¿qué sucede si hay que escribir en una localidad que no está actualmente en caché? ¿Se deberán traer los datos a la caché o simplemente escribirse en la memoria? Una vez más, ninguna de las respuestas es siempre la mejor. Casi todos los diseños que difieren las escrituras en la memoria tienden a traer los datos viejos a la caché cuando no están ya en ella, técnica llamada **asignación de escritura**. En cambio, casi ningún diseño que practique la escritura a través asignará una entrada en una operación de escritura porque tal opción complicaría un diseño por lo demás sencillo. La asignación de escritura es mejor sólo si se repiten las escrituras a la misma palabra o a palabras que están en la misma línea de caché.

4.5.2 Predicción de ramificaciones

Las computadoras modernas usan filas de procesamiento. La fila de procesamiento de la figura 4-35 tiene siete etapas; las computadoras del extremo superior a veces tienen filas de

procesamiento de diez etapas o más. Las filas de procesamiento funcionan mejor con código lineal, pues así la unidad de obtención puede leer palabras consecutivas de la memoria y enviarlas a la unidad decodificadora antes de que se necesiten.

El único problema menor con este maravilloso modelo es que no se ajusta para nada a la realidad. Los programas no son secuencias lineales de código; están llenos de instrucciones de ramificación. Considere los sencillos enunciados de la figura 4-40(a). Una variable, *i*, se compara con 0 (lo que probablemente sea la prueba más común en la práctica). Dependiendo del resultado, se asigna a otra variable, *k*, uno de dos posibles valores.

```
if (i == 0)                    CMP i,0     ; comparar i con 0
   k = 1;                      BNE Else    ; saltar a Else si no es igual
else                   Then:   MOV k,1     ; poner 1 en k
   k = 2;                      BR Next     ; salto incondicional a Next
                       Else:   MOV k,2     ; poner 2 en k
                       Next:
   (a)                             (b)
```

Figura 4-40. (a) Fragmento de programa. (b) Su traducción a un lenguaje de ensamblador genérico.

En la figura 4-40(b) se muestra una posible traducción a lenguaje ensamblador. Estudiaremos el lenguaje ensamblador más adelante; los detalles no son importantes ahora, pero dependiendo de la máquina y el compilador es probable que se produzca código similar al de la figura. La primera instrucción compara *i* con 0. La segunda ramifica a la etiqueta *Else* (el inicio de la cláusula else) si *i* no es 0. La tercera instrucción asigna 1 a *k*. La cuarta instrucción ramifica al código para la siguiente instrucción. El compilador puso una etiqueta cómoda ahí, *Next*, para tener a dónde ramificar. La quinta instrucción asigna 2 a *k*.

Lo que debemos observar aquí es que dos de las cinco instrucciones son ramificaciones. Además, una de ellas, BNE, es una ramificación condicional (una rama que se sigue si y sólo si se cumple alguna condición, en este caso, que los dos operandos del CMP previo sean iguales). La secuencia de código lineal más larga aquí es de dos instrucciones. Por consiguiente, traer instrucciones a gran velocidad para alimentar los conductos no es fácil.

A primera vista, podría parecer que las ramificaciones incondicionales, como la instrucción BR Next de la figura 4-40(b) no presentan un problema. Después de todo, no hay ambigüedad respecto a dónde ir. ¿Por qué la unidad de obtención no puede seguir leyendo instrucciones a partir de la dirección objetivo (el lugar a donde se saltará)?

El problema radica en la naturaleza de los conductos. En la figura 4-35, por ejemplo, vemos que la decodificación de instrucciones se efectúa en la segunda etapa. Así, la unidad de obtención tiene que decidir de dónde traerá bytes a continuación antes de saber qué tipo de instrucción acaba de traer. Un ciclo después, la unidad de búsqueda podrá enterarse de que dicha instrucción fue una ramificación incondicional, pero para entonces ya empezó a buscar la instrucción que seguía a la ramificación incondicional. Es por esto que una proporción considerable de las máquinas que usan filas de procesamiento (como el UltraSPARC II) tienen la propiedad de que se ejecuta la instrucción *que sigue* a una ramificación incondicional,

aunque lógicamente no debería hacerse. La posición que sigue a una ramificación se llama **ranura de retraso**. En Pentium II [y la máquina de la figura 4-40(b)] no tienen esta propiedad, pero la complejidad interna que se requiere para superar este problema suele ser enorme. Un compilador optimante tratará de encontrar alguna instrucción útil que pueda colocar en la ranura de retraso, pero es común que no haya nada disponible, y el compilador se verá obligado a insertar una instrucción NOP ahí. Ello asegura que el programa sea correcto, pero lo hace más grande y lento.

Aunque las ramificaciones incondicionales son molestas, las condicionales son peores. No sólo tienen ranuras de retraso, sino que ahora la unidad de búsqueda no sabe de dónde tendrá que leer sino hasta mucho más adelante en la fila de procesamiento. Las primeras máquinas con fila de procesamiento simplemente **paraban** hasta que se sabía si se iba a tomar la rama o no. Parar durante tres o cuatro ciclos en cada ramificación condicional, sobre todo si el 20% de las instrucciones son ramificaciones condicionales, arruina el desempeño.

Por ello, lo que la mayor parte de las máquinas hace cuando llega a una rama condicional es predecir si se va a tomar o no. Sería bueno que pudiéramos conectar una bola de cristal a una ranura PCI libre para que nos ayudara con la predicción, pero hasta ahora este enfoque no ha sido fructífero.

Ante la falta de tal periférico, se han ideado varias formas de efectuar la predicción. Una muy sencilla es la siguiente: suponer que se tomarán todas las ramas condicionales hacia atrás y que no se tomará ninguna rama condicional hacia adelante. El razonamiento que justifica la primera parte es que es común encontrar ramas hacia atrás al final de un ciclo. Casi todos los ciclos se ejecutan varias veces, así que adivinar que se tomará una rama hacia atrás, hacia el principio del ciclo, es una buena apuesta.

La segunda parte no está tan bien fundamentada. Ocurren algunas ramas hacia adelante cuando se detectan condiciones de error en software (por ejemplo, no se puede abrir un archivo). Los errores son raros, así que casi todas las ramas asociadas a ellos no se tomarán. Desde luego, hay muchas ramas hacia adelante que no están relacionadas con el manejo de errores, así que la tasa de aciertos no es tan buena como con las ramas hacia atrás. Aunque no es una maravilla, esta regla al menos es mejor que nada.

Si se predice correctamente una rama, no hay nada especial que hacer. La ejecución simplemente continúa en la dirección objetivo. El problema surge cuando se predice erróneamente una rama. Averiguar a dónde hay que ir e ir ahí no es difícil. La parte difícil es anular instrucciones que ya se ejecutaron y no debieron haberse ejecutado.

Hay dos formas de atacar el problema. La primera es permitir que las instrucciones que se adquieren después de una ramificación condicional predicha se ejecuten en tanto no traten de modificar el estado de la máquina (por ejemplo, guardar algo en un registro). En lugar de sobreescribir el registro, el valor calculado se almacena en un registro de borrador (secreto) y sólo se copia en el registro real una vez que se sabe que la predicción fue correcta. La segunda técnica es guardar el valor de cualquier registro que esté a punto de sobreescribirse (digamos en un registro de borrador secreto) para que la máquina pueda revertirse al estado en que estaba cuando se llegó a la rama predicha erróneamente. Ambas soluciones son complejas y requieren una contabilización muy estricta para funcionar bien. Y si se llega a una segunda ramificación condicional antes de que se sepa si la primera se predijo correctamente, las cosas pueden ponerse difíciles.

Predicción dinámica de ramificaciones

Obviamente, es importantísimo que las predicciones sean correctas, ya que permite a la CPU trabajar a toda velocidad. Por ello, muchas investigaciones actuales buscan mejorar los algoritmos de predicción (por ejemplo, Driesen y Holzle, 1998; Juan *et al.*, 1998; Pan *et al.*, 1992; Sechrest *et al.*, 1996; Sprangle *et al.*, 1997; y Yeh y Patt, 1991). Una estrategia es que la CPU mantenga una tabla histórica (en hardware especial) en la que registra las ramificaciones condicionales conforme ocurren, para poder consultarlas cuando vuelvan a ocurrir. La versión más simple de este esquema se muestra en la figura 4-41(a). Aquí la tabla histórica contiene una entrada por cada instrucción de ramificación condicional. La entrada contiene la dirección de la instrucción de ramificación junto con un bit que indica si se tomó la última vez que se ejecutó. Cuando se utiliza este esquema, la predicción es simplemente que la ramificación tomará la misma trayectoria que adoptó la vez anterior. Si la predicción es errónea, el bit de la tabla histórica se cambia.

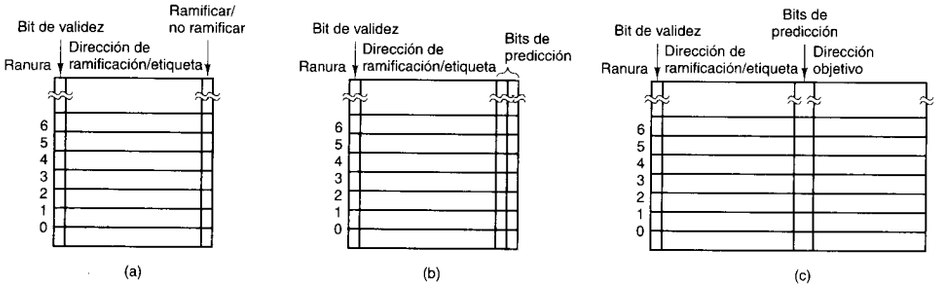


Figura 4-41. (a) Historia de ramificación de 1 bit. (b) Historia de ramificación de 2 bits. (c) Correspondencia entre la dirección de la instrucción de ramificación y la dirección objetivo.

Hay varias formas de organizar la tabla histórica; de hecho, son precisamente las técnicas que se usan para organizar una caché. Considere una máquina con instrucciones de 32 bits que están alineadas por palabras de modo que los dos bits de orden bajo de cada dirección de memoria son 00. Con una tabla histórica de mapeo directo que contiene 2^n entradas, los $n + 2$ bits de orden bajo de una instrucción de ramificación se pueden extraer y desplazar hacia la derecha dos bits. Este número de n bits se puede usar como índice de la tabla histórica, donde se verifica si la dirección guardada coincide con la dirección de la ramificación. Al igual que con una caché, no hay necesidad de guardar los $n + 2$ bits de orden bajo, así que pueden omitirse (es decir, sólo se guardan los bits de dirección superiores, la etiqueta). Si hay un acierto, se usa el bit de predicción para predecir la ramificación. Si está presente la etiqueta equivocada o la entrada no es válida, hay un fallo, igual que en una caché. En este caso puede usarse la regla de la rama hacia adelante o hacia atrás.

Si la tabla de historial de ramificación tiene, digamos, 4096 entradas, las ramificaciones en las direcciones 0, 16384, 32768, . . . chocarán, en un problema análogo al que se presenta

en una caché. Se puede aplicar la misma solución: una entrada asociativa de 2, 4 o *n* vías. Al igual que en una caché, el caso limitante es una sola entrada asociativa de *n* vías, que requiere una búsqueda con plena asociatividad.

Si la tabla es lo bastante grande y tiene suficiente asociatividad, este esquema funciona bien en la mayor parte de las situaciones. Sin embargo, siempre se presenta un problema sistemático. Cuando por fin se sale de un ciclo, la predicción de la ramificación final siempre será errónea. Lo que es peor, el error en la predicción hará que se modifique el bit de la tabla histórica para indicar una predicción futura de "no tomar la rama". La siguiente vez que se ingrese en el ciclo, la predicción de ramificación al final de la primera iteración será errónea. Si el ciclo está dentro de un ciclo exterior, o en un procedimiento que se invoca a menudo, este error podría ocurrir con mucha frecuencia.

Para eliminar esta falsa predicción, podríamos dar una segunda oportunidad a la entrada de la tabla. Con este método, la predicción sólo se cambia después de dos predicciones incorrectas consecutivas. Esto requiere dos bits de predicción en la tabla histórica, uno para lo que "se supone" que debe hacer la ramificación y otro para lo que hizo la última vez, como se muestra en la figura 4-41(b).

Una forma un poco distinta de ver este algoritmo es como una máquina de estados finitos con cuatro estados, la cual se ilustra en la figura 4-42. Después de una serie de predicciones de "no tomar la rama" acertadas, la FSM estará en el estado 00 y su predicción para la próxima vez será "no tomar la rama". Si esa predicción es errónea, pasará al estado 10, pero su predicción para la próxima vez seguirá siendo "no tomar la rama". Sólo si esta predicción es errónea la FSM pasará al estado 11 y comenzará a predecir "tomar la rama" todo el tiempo. Efectivamente, el bit de la izquierda del estado es la predicción, y el de la derecha es lo que se hizo la vez anterior. Aunque este diseño usa sólo 2 bits de historial, existen diseños que siguen la pista a 4 u 8 bits de historial.

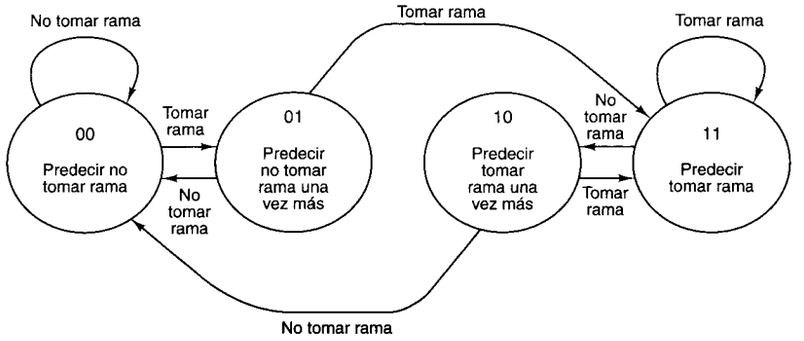


Figura 4-42. Máquina de estados finitos de dos bits para predecir ramificaciones.

Ésta no es nuestra primera FSM. La figura 4-28 también era una FSM. De hecho, todos nuestros microprogramas pueden considerarse como máquinas de estados finitos, puesto que cada línea representa un estado específico en el que puede estar la máquina, con transiciones bien definidas a un conjunto finito de estados distintos. Las FSM se usan mucho en todos los aspectos del diseño de hardware.

Hasta aquí hemos supuesto que se conoce el objetivo de cada ramificación condicional, casi siempre como una dirección explícita a la cual saltar (contenida en la instrucción misma) o como distancia relativa desde la instrucción actual (por ejemplo, un número con signo que se suma al contador de programa). Con frecuencia este supuesto es válido, pero algunas instrucciones de ramificación condicional calculan la dirección objetivo realizando operaciones aritméticas con registros, y luego saltan a la dirección calculada. Incluso si la FSM de la figura 4-42 predice con exactitud la rama que se tomará, de nada sirve tal predicción si no se conoce la dirección objetivo. Una forma de manejar esta situación es guardar en la tabla histórica la dirección a la que se saltó la última vez, como se muestra en la figura 4-41(c). De este modo, si la tabla dice que la última vez que tomamos la rama que sale de la dirección 516 nos llevó a la dirección 4000, y lo que se predice es "tomar la rama", podremos suponer que de nuevo se saltará a 4000.

Una estrategia de predicción de ramificaciones distinta consiste en llevar un registro de si se tomaron o no las últimas *k* ramas condicionales que se encontraron, sin importar de cuál instrucción se trató (Pan *et al.*, 1992). Este número de *k* bits, que se guarda en el **registro de desplazamiento histórico de ramificación**, se compara en paralelo con todas las entradas de una tabla histórica con una clave de *k* bits y, si coincide, se usa la predicción que está ahí. Aunque parezca sorprendente, esta técnica funciona muy bien.

Predicción de ramificación estática

Todas las técnicas de predicción de ramificaciones que hemos visto hasta ahora son dinámicas, es decir, se ejecutan sobre la marcha durante la ejecución del programa. Además, se adaptan al comportamiento actual del programa, lo cual es bueno. La desventaja es que requieren hardware especializado y costoso y chips muy complejos.

Una estrategia diferente es hacer que el compilador nos ayude. Cuando el compilador ve un enunciado como

```
for (i = 0; i < 1000000; i++) { ... }
```

sabe muy bien que casi todas las veces se tomará la rama al final del ciclo. Si hubiera alguna forma de decírselo al hardware, se ahorraría mucho trabajo.

Aunque éste es un cambio arquitectónico (y no sólo una cuestión de implementación), algunas máquinas, como la UltraSPARC II, tienen un segundo conjunto de instrucciones de ramificación condicional además de las normales (que se necesitan por compatibilidad hacia atrás). Las instrucciones nuevas contienen un bit en el que el compilador puede especificar que cree que la rama se tomará (o no se tomará). Cuando se llega a una de estas instrucciones, la unidad de búsqueda hace lo que se le dice. Además, no hay necesidad de desperdiciar espacio valioso de la tabla histórica de ramificación para estas instrucciones, lo que reduce el número de conflictos en la tabla.

Nuestra última técnica de predicción de ramificaciones se basa en crear perfiles (Fisher y Freudenberger, 1992). Esta técnica también es estática, pero en lugar de pedir al compilador que trate de adivinar cuáles ramas se tomarán y cuáles no, el programa se ejecuta realmente (por lo regular en un simulador) y se captura el comportamiento de ramificación. Esta infor-

mación se alimenta al compilador, que entonces usa las instrucciones de ramificación condicional especiales para indicar al hardware qué debe hacer.

4.5.3 Ejecución fuera de orden y cambio de nombres de registros

Casi todas las CPU modernas usan filas de procesamiento y son superescalares, como se muestra en la figura 2-6. Lo que esto significa en general es que existe una unidad de búsqueda que saca palabras de instrucciones de la memoria antes de que se necesiten, a fin de alimentar una unidad decodificadora. Ésta emite las instrucciones decodificadas a las unidades funcionales apropiadas para su ejecución. En algunos casos, la unidad decodificadora podría desglosar instrucciones individuales en microoperaciones antes de emitirlas a las unidades funcionales, dependiendo de las capacidades de estas últimas.

Es evidente que el diseño de la máquina es el más sencillo posible si todas las instrucciones se ejecutan en el orden en que se traen de la memoria (suponiendo por el momento que el algoritmo de predicción de ramificaciones nunca se equivoca). Sin embargo, la ejecución en orden no siempre da lugar a un desempeño óptimo debido a las dependencias entre instrucciones. Si una instrucción requiere un valor calculado por la instrucción anterior, la segunda instrucción no podrá iniciar su ejecución hasta que la primera haya producido el valor requerido. En esta situación (una dependencia RAW), la segunda instrucción tiene que esperar. Existen también otros tipos de dependencias, como pronto veremos.

En un intento por superar estos problemas y lograr un mejor desempeño, algunas CPU permiten saltarse por el momento las instrucciones dependientes para llegar a instrucciones futuras que no sean dependientes. Sobra decir que el algoritmo de planificación de instrucciones interno que se use debe producir el mismo efecto que se obtendría si el programa se ejecutara en el orden en que está escrito. A continuación demostraremos cómo funciona el reordenamiento de instrucciones empleando un ejemplo detallado.

Para ilustrar la naturaleza del problema, comenzaremos con una máquina que siempre emite las instrucciones en el orden del programa y también requiere que terminen su ejecución en el orden del programa. La importancia de la segunda condición se hará evidente más adelante.

Nuestra máquina de ejemplo tiene ocho registros que el programador puede ver, R0 a R7. Todas las instrucciones aritméticas usan tres registros: dos para los operandos y uno para el resultado, igual que el Mic-4. Supondremos que si una instrucción se decodifica en el ciclo n, su ejecución inicia en el ciclo $n + 1$. En el caso de una instrucción sencilla, como una suma o resta, la escritura del resultado en el registro de destino ocurre al final del ciclo $n + 2$. Si la instrucción es más complicada, como una multiplicación, la escritura ocurre al final del ciclo $n + 3$. Para que el ejemplo sea realista, permitiremos a la unidad decodificadora emitir hasta dos instrucciones en cada ciclo de reloj. Las CPU superescalares comerciales a menudo pueden emitir cuatro o incluso seis instrucciones por ciclo de reloj.

Nuestra secuencia de ejecución de ejemplo se muestra en la figura 4-43. Aquí la primera columna da el número del ciclo, y la segunda, el número de la instrucción. La tercera columna es una lista de las instrucciones decodificadas. La cuarta indica cuál instrucción se está emitiendo (con un máximo de dos por ciclo de reloj). La quinta indica cuál instrucción se

retiró (terminó). Recuerde que en este ejemplo estamos exigiendo tanto emisión en orden como terminación en orden, de modo que la instrucción $k + 1$ no puede emitirse hasta que se ha emitido la instrucción k, y la instrucción $k + 1$ no puede retirarse (después de efectuarse la escritura de resultados en el registro de destino) hasta que se ha retirado la instrucción k. A continuación explicaremos las otras 16 columnas.

					Registros leídos								Registros escritos							
Cic.	**#**	**Decodif.**	**Emit.**	**Ret.**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1 2	R3=R0*R1 R4=R0+R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5=R0+R1 R6=R1+R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3					3	2	1									1	1	1		
4				1 2 3	2 1	1 1	1										1	1 1		
5	 5	R7=R1*R2	4 5			1 2	 1		1 1										1 1	 1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9	 7	R3=R3*R1	6 7		1 1	 1	1 1	 1						1 1		 1				
10					1	1	1	1						1		1				
11				6		1		1								1				
12				7																
13	8	R1=R4+R4	8						2					1						
14									2					1						
15				8																

Figura 4-43. Funcionamiento de una CPU superescalar con emisión en orden y terminación en orden.

Después de decodificar una instrucción, la unidad decodificadora tiene que decidir si se puede emitir de inmediato o no. Para tomar esta decisión, la unidad necesita conocer la situación de todos los registros. Si, por ejemplo, la instrucción en curso necesita un registro cuyo valor todavía no se ha calculado, la instrucción no podrá emitirse y la CPU deberá parar.

Seguimos la pista al uso de registros con un dispositivo llamado **marcador**, que se usó por primera vez en la CDC 6600. El marcador tiene un pequeño contador para cada registro

que indica cuántas veces ese registro es utilizado como fuente por las instrucciones que se están ejecutando. Si puede haber un máximo de, digamos, 15 instrucciones en ejecución en un momento dado, basta con un contador de 4 bits. Cuando se emite una instrucción, se incrementan las entradas del marcador correspondientes a sus registros operandos. Cuando se retira una instrucción, las entradas se decrementan.

El marcador también tiene un contador para cada registro que sigue la pista a los registros que se están usando como destinos. Puesto que sólo se permite una escritura a la vez, estos contadores pueden tener una capacidad de 1 bit. Las 16 columnas de la derecha de la figura 4-43 muestran el marcador.

En las máquinas reales, el marcador también sigue la pista al uso de las unidades funcionales, a fin de evitar emitir una instrucción para la cual no está disponible una unidad funcional. Para simplificar, supondremos que siempre está disponible una unidad funcional apropiada, por lo que no mostraremos las unidades funcionales en el marcador.

La primera línea de la figura 4-43 muestra I1 (instrucción 1) que multiplica R0 por R1 y coloca el resultado en R3. Puesto que ninguno de estos registros se está usando todavía, la instrucción se emite y el marcador se actualiza de modo que refleje que R0 y R1 se están leyendo y R3 se está escribiendo. Ninguna instrucción subsecuente podrá escribir en ninguno de éstos ni podrá leer R3 hasta que I1 se haya retirado. Puesto que esta instrucción es una multiplicación, terminará al final del ciclo 4. Los valores del marcador que se muestran en cada línea reflejan su estado después de emitirse la instrucción de esa línea. Las entradas en blanco son ceros.

Puesto que nuestro ejemplo es una máquina superescalar que puede emitir dos instrucciones por ciclo, se emite una segunda instrucción (I2) durante el ciclo 1. Ésta suma R0 y R2, y guarda el resultado en R4. Para ver si es posible emitir esta instrucción, se aplican las reglas siguientes:

1. Si se está escribiendo en cualquier operando, no emitir (dependencia RAW).
2. Si se está leyendo el registro del resultado, no emitir (dependencia WAR).
3. Si se está escribiendo en el registro de resultado, no emitir (dependencia WAW).

Ya vimos las dependencias RAW, que ocurren cuando una instrucción necesita usar como fuente un resultado que una instrucción anterior todavía no produce. Las otras dos dependencias son menos graves. Se trata básicamente de conflictos de recursos. En una **dependencia WAR** (escritura después de lectura, *Write After Read*), una instrucción está tratando de sobreescribir un registro que una instrucción anterior tal vez no haya terminado de leer todavía. Una **dependencia WAW** (escritura después de escritura, *Write After Write*) es similar. En muchos casos podemos evitar estas dependencias haciendo que la segunda instrucción coloque sus resultados en algún otro lugar (tal vez temporalmente). Si no existe ninguna de las tres dependencias anteriores, y la unidad funcional que se necesita está disponible, la instrucción se emite. En este caso, I2 usa un registro (R0) que está siendo leído por una instrucción pendiente, pero este traslape está permitido, por lo que I2 se emite. De forma similar, I3 se emite durante el ciclo 2.

Ahora llegamos a I4, que necesita usar R4. Por desgracia, vemos en la línea 3 que se está escribiendo en R4. Tenemos aquí una dependencia RAW, por lo que la unidad decodificadora

parará hasta que R4 esté disponible. Mientras está parada, la unidad deja de sacar instrucciones de la unidad de búsqueda. Cuando los buffers internos de la unidad de búsqueda se llenen, ésta dejará de prebuscar instrucciones.

Vale la pena señalar que la siguiente instrucción en el orden del programa, I5, no tiene conflictos con ninguna de las instrucciones pendientes; podría haberse decodificado y emitido si no fuera por el hecho de que este diseño requiere emitir las instrucciones en orden.

Veamos ahora qué sucede durante el ciclo 3. I2, al ser una suma (dos ciclos), termina al final del ciclo 3. Lamentablemente, no podemos retirarla (lo cual dejaría libre R4 para I4). ¿Por qué no? La razón es que este diseño también requiere retirar en orden. ¿Por qué? ¿Qué de malo podría tener realizar el almacenamiento en R4 ahora y marcar el registro como disponible?

La respuesta es sutil, pero importante. Supongamos que las instruccines pudieran terminar en desorden. Si entonces ocurriera una interrupción, sería muy difícil guardar el estado de la máquina para poder restaurarlo después. En particular, no sería posible decir que todas las instrucciones hasta cierta dirección ya se ejecutaron y todas las instrucciones posteriores no. Esto se llama **interrupción precisa** y es una característica deseable de una CPU (Moudgill y Vassiliadis, 1996). El retiro en desorden hace que las interrupciones no sean precisas, y es por esto que algunas máquinas exigen que las instrucciones terminen en orden.

Retomando nuestro ejemplo, al final del ciclo 4 se pueden retirar las tres instrucciones pendientes, así que en el ciclo 5 ya es posible emitir por fin I4 junto con la recién decodificada I5. Cada vez que se retira una instrucción, la unidad decodificadora tiene que verificar si hay una instrucción en espera que ya sea posible emitir.

En el ciclo 6 I6 se para porque necesita escribir en R1 y R1 está ocupado. I6 por fin se inicia en el ciclo 9. Toda la secuencia de ocho instrucciones tarda 15 ciclos en completarse debido a muchas dependencias, aunque el hardware es capaz de emitir dos instrucciones en cada ciclo. Sin embargo, al examinar la columna *Emit.* de la figura 4-43 podemos ver que todas las instrucciones se emitieron en orden. De igual manera, la columna *Ret.* muestra que las instrucciones también se retiraron en orden.

Consideremos ahora un diseño alternativo: la ejecución en desorden. En este diseño, las instrucciones pueden emitirse en desorden y también retirarse en desorden. En la figura 4-44 se muestra la misma secuencia de ocho instrucciones, sólo que ahora permitiendo emisión y retiro en desorden.

La primera diferencia ocurre en el ciclo 3. Aunque I4 está parada, podemos decodificar y emitir I5 porque no está en conflicto con ninguna instrucción pendiente. Sin embargo, saltarse instrucciones causa un nuevo problema. Suponga que I5 hubiera utilizado un operando calculado por la instrucción que se pasó por alto, I4. Con el marcador actual, no nos habríamos dado cuenta de ello. Por consiguiente, tenemos que extender el marcador para seguir la pista a las escrituras realizadas por instrucciones pasadas por alto. Esto puede hacerse añadiendo un segundo mapa de bits, con un bit por registro, para contabilizar las escrituras efectuadas por instrucciones paradas. (Estos contadores no se muestran en la figura.) Ahora es necesario extender la regla para emitir instrucciones, a fin de prohibir la emisión de cualquier instrucción cuyo operando esté programado para escritura por una instrucción que era anterior pero se pasó por alto.

					Registros leídos								Registros escritos							
Cic.	**#**	**Decodif.**	**Emit.**	**Ret.**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1 2	R3=R0*R1 R4=R0+R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5=R0+R1 R6=R1+R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3	5 6	R7=R1*R2 S1=R0–R2	5 6	 2	3 4 3	3 3 3	2 3 2									1 1 1	1 1	1 1 1		1 1 1
4	 7 8	 R3=R3-S1 S2=R4+R4	4 – 8	 1 3	3 3 3 2 1	4 4 4 3 2	2 2 2 2 2		1 1 3 3 3							1 1 1		1 1 1 1	1 1 1 1	1 1 1 1 1
5				6		2	1		3					1					1	1
6			7	 4 5 8		2 1	1 1	1 1 1 1	2 2 2					1 1 1		1 1 1			1	1 1
7								1								1				
8								1								1				
9				7																

Figura 4-44. Funcionamiento de una CPU superescalar con emisión en desorden y terminación en desorden.

Examinemos otra vez I6, I7 e I8 en la figura 4-43. Vemos que I6 calcula un valor en R1 que es utilizado por I7. Sin embargo, también vemos que el valor nunca se vuelve a usar porque I8 sobreescribe R1. No existe una razón de peso para usar R1 como lugar donde retener el resultado de I6. Peor aún, la decisión de usar R1 como registro intermedio es pésima, aunque perfectamente razonable para un compilador o programador acostumbrado a la idea de ejecución secuencial sin traslape de instrucciones.

En la figura 4-44 introducimos una nueva técnica para resolver este problema: el **cambio de nombre de registros**. La unidad decodificadora inteligente cambia el uso de R1 en I6 (ciclo 3) e I7 (ciclo 4) a un registro secreto, S1, que el programador no ve. Ahora I6 puede emitirse junto con I5. Las CPU modernas suelen tener docenas de registros secretos que usan para cambiar el nombre de los registros. En muchos casos esta técnica puede eliminar las dependencias WAR y WAW.

En I8 podemos volver a usar el cambio de nombre de registros. Esta vez R1 cambia de nombre a S2 para que la suma pueda iniciarse antes de que R1 se desocupe, al final del ciclo 6. Si en este caso el resultado realmente tiene que estar en R1, siempre cabe la posibilidad de

copiar S2 otra vez ahí justo a tiempo. Mejor aún, todas las instrucciones futuras que lo necesiten pueden cambiar el nombre de su fuente al del registro donde en verdad está guardado. En todo caso, la suma de I8 logró efectuarse antes de este modo.

En muchas máquinas reales, el cambio de nombre está incorporado profundamente en la organización de los registros. Hay muchos registros secretos y una tabla que establece la correspondencia entre los registros que el programador ve y los registros secretos. Así, el verdadero registro que se está usando para, digamos, R0, se encuentra examinando la entrada 0 de esta tabla de correspondencia. Así, no existe un registro R0 real, sólo un vínculo entre el nombre R0 y uno de los registros secretos. Este vínculo cambia con frecuencia durante la ejecución para evitar dependencias.

Si examina la cuarta columna de la figura 4-44, verá que las instrucciones no se emitieron en orden. Tampoco se retiraron en orden. La conclusión de este ejemplo es sencilla: gracias a la ejecución en desorden y el cambio de nombre de registros pudimos reducir el tiempo de ejecución casi a la mitad.

4.5.4 Ejecución especulativa

En la sección anterior presentamos el concepto de reordenar las instrucciones para mejorar el desempeño. Aunque no lo mencionamos explícitamente, lo que buscábamos ahí era reordenar las instrucciones dentro de un solo bloque básico. Es momento de examinar este punto con más detenimiento.

Los programas de computadora se pueden dividir en **bloques básicos**, cada uno de los cuales consiste en una secuencia lineal de código con un punto de entrada arriba y uno de salida abajo. Un bloque básico no contiene estructuras de control (como enunciados if o enunciados while), así que su traducción a lenguaje de máquina no contiene ramificaciones. Los bloques básicos se conectan mediante enunciados de control.

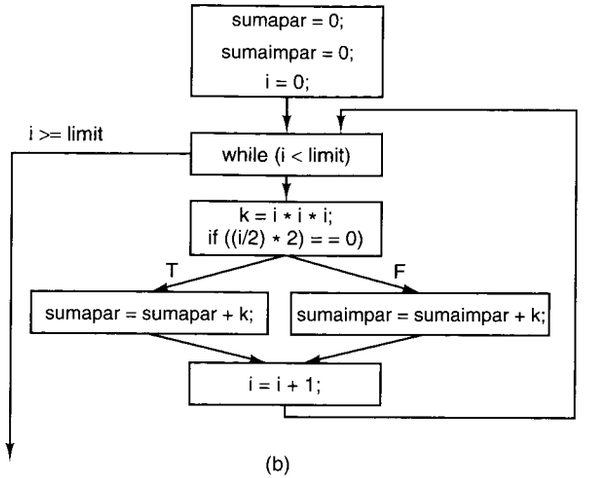
Un programa que adopta esta forma se puede representar como grafo dirigido, como se ilustra en la figura 4-45. Ahí calculamos la suma de los cubos de los enteros pares e impares hasta cierto límite y los acumulamos en *sumapar* y *sumaimpar*, respectivamente. Dentro de cada bloque básico, las técnicas de reordenamiento de la sección anterior funcionan de maravilla.

El problema es que casi todos los bloques básicos son cortos y no contienen suficiente paralelismo como para aprovecharlos de forma eficaz. Por consiguiente, nuestro siguiente paso es permitir que el reordenamiento cruce las fronteras de los bloques básicos en un intento por llenar todas las ranuras de emisión. Las mayores ganancias se obtienen cuando una operación que podría ser lenta se puede adelantar en el grafo e iniciarse antes. Ésta podría ser una instrucción LOAD, una operación de punto flotante o incluso el inicio de una cadena de dependencias larga. La acción de adelantar código que inicia después de una ramificación se denomina **levantamiento de código**.

Imagine que en la figura 4-45 todas las variables se guardan en registros excepto *sumapar* y *sumaimpar* (por falta de registros). Podría ser razonable adelantar sus instrucciones LOAD al principio del ciclo, antes de calcular *k*, a fin de iniciarlas antes y que los valores cargados estén disponibles cuando se necesiten. Desde luego, sólo uno de ellos se necesitará en cada iteración, y el otro LOAD se desperdiciaría, pero si la caché y la memoria usan filas de

```
sumapar = 0;
sumaimpar = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if ((i/2) * 2) == 0)
        sumapar = sumapar + k;
    else
        sumaimpar = sumaimpar + k;
    i = i + 1;
}
```

(a)



(b)

Figura 4-45. (a) Fragmento de programa. (b) El grafo de bloque básico correspondiente.

procesamiento y hay ranuras de emisión disponibles, podría valer la pena hacerlo. La ejecución de código antes de saber si va a necesitarse se llama **ejecución especulativa**. El uso de esta técnica requiere apoyo del compilador y del hardware, además de ciertas extensiones arquitectónicas. En casi todos los casos, reordenar las instrucciones más allá de las fronteras de un bloque básico rebasa la capacidad del hardware, por lo que el compilador debe cambiar de lugar las instrucciones explícitamente.

La ejecución especulativa introduce algunos problemas interesantes. Uno de ellos es que es indispensable que ninguna de las instrucciones especulativas tenga efectos irrevocables, porque después podría resultar que no debieron de haberse ejecutado. En la figura 4-45 no hay problema si buscamos *sumapar* y *sumaimpar*, y tampoco si efectuamos la suma tan pronto como tenemos *k* (aún antes del enunciado if), pero no sería correcto guardar los resultados en la memoria. En secuencias de código más complicadas, una forma común de evitar que el código especulativo sobreescriba registros antes de que se sepa si es lo que debe hacerse es cambiar el nombre de todos los registros de destino empleados por el código especulativo. De este modo, sólo se modificarán registros de borrador, y no habrá problema si después resulta que el código no se necesitaba. Si el código es necesario, los registros de borrador se copian en los verdaderos registros de destino. Como podrá imaginar, el manejo de contador para llevar la contabilidad de todo esto no es sencillo, pero si se cuenta con suficiente hardware, es factible.

Sin embargo, el código especulativo introduce otro problema que no puede resolverse cambiando el nombre de los registros. ¿Qué sucede si una instrucción ejecutada especulativamente causa una excepción? Un ejemplo doloroso, aunque no fatal, es una instrucción LOAD que causa un fallo de caché en una máquina con tamaño de línea de caché grande (digamos, 256 bytes) y una memoria mucho más lenta que la CPU y la caché. Si un LOAD

que realmente se necesita para en seco la máquina durante muchos ciclos mientras la línea de caché se está cargando, pues ni modo: así es la vida; la palabra era necesaria. Por otra parte, parar la máquina para traer una palabra que resulta innecesaria es contraproducente. Un número excesivo de tales "optimaciones" puede hacer que la CPU sea más lenta que si no las tuviera. (Si la máquina tiene memoria virtual, que veremos en el capítulo 6, un LOAD especulativo podría incluso causar un fallo de página, que requiere una operación de disco para traer la página requerida. Los fallos de página falsos pueden arruinar el desempeño, así que es importante evitarlos.)

Una solución que implementan varias máquinas modernas es tener una instrucción especial de carga especulativa que trata de traer la palabra de la caché, pero si no está ahí simplemente se da por vencida. Si el valor en realidad se necesita y está ahí, podrá usarse, pero si no está el hardware deberá ir por él en ese momento. Si resulta que el valor no era necesario, no se habrá pagado ningún precio por el fallo de caché.

El siguiente enunciado ilustra una situación mucho peor:

```
if (x > 0) z = y/x;
```

donde *x*, *y* y *z* son variables de punto flotante. Suponga que todas las variables se buscan y se colocan en registros por adelantado y que la división de punto flotante (lenta) se levanta de modo que se inicie antes de la prueba if. El problema es que *x* es 0 y el error de división entre cero hace que el programa termine. El resultado neto es que la especulación ha hecho que un programa correcto falle. Peor aún, el programador incluyó código explícito para evitar esta situación, y de todos modos ocurrió. No es probable que el programador esté muy contento.

Una posible solución es tener versiones especiales de las instrucciones que podrían causar excepciones. Además, se añade a cada registro un bit, llamado **bit venenoso**. Cuando una instrucción especulativa especial falla, en lugar de causar un error asigna 1 al bit venenoso del registro de resultado. Si posteriormente una instrucción normal toca ese registro, ocurrirá el error (como debería). En cambio, si el resultado nunca se usa, el bit venenoso tarde o temprano se pone en 0 y no habrá pasado nada.

4.6 EJEMPLOS DEL NIVEL DE MICROARQUITECTURA

En esta sección presentaremos breves ejemplos de tres procesadores de vanguardia, mostrando cómo usan los conceptos que exploramos en este capítulo. La exposición será breve por necesidad, pues las máquinas reales tienen una complejidad enorme y contienen millones de compuertas. Los ejemplos son los mismos que hemos estado usando hasta ahora: Pentium II, UltraSPARC II y picoJava II.

4.6.1 La microarquitectura de la CPU Pentium II

El Pentium II representa el extremo superior de la familia de CPU de Intel. Maneja operandos y operaciones aritméticas de 32 bits, así como operaciones de punto flotante de 64 bits. Además, maneja operandos y operaciones de 8 y 16 bits, lo cual es un legado de anteriores

procesadores de la familia. El Pentium II puede direccionar hasta 64 GB de memoria y lee palabras de la memoria en grupos de 64 bits. En la figura 3-50 se ilustra un sistema Pentium II representativo.

Como vimos antes, y como se ilustra en la figura 3-43, el paquete SEC del Pentium II consiste en dos circuitos integrados: la CPU (que incluye las cachés de nivel 1 divididos) y la caché de nivel 2 unificado. En la figura 4-46 se muestran los componentes primarios de la CPU: las unidades de Búsqueda/Decodificación, Despacho/Ejecución y Retiro, que juntas actúan como una fila de procesamiento de alto nivel. Estas tres unidades se comunican a través de una reserva de instrucciones, un lugar donde se mantiene la información relacionada con instrucciones cuya ejecución todavía no termina. La información de la reserva de instrucciones se guarda en una tabla llamada **buffer de reordenamiento** (**ROB**, *ReOrder Buffer*). En pocas palabras, la unidad de Búsqueda/Decodificación busca instrucciones y las divide en microoperaciones que guarda en el ROB. La unidad de Despacho/Ejecución toma microoperaciones del ROB y las ejecuta. La unidad de Retiro finaliza la ejecución de cada microoperación y actualiza los registros. Las instrucciones ingresan en el ROB en orden, se pueden ejecutar en desorden, pero se retiran en orden.

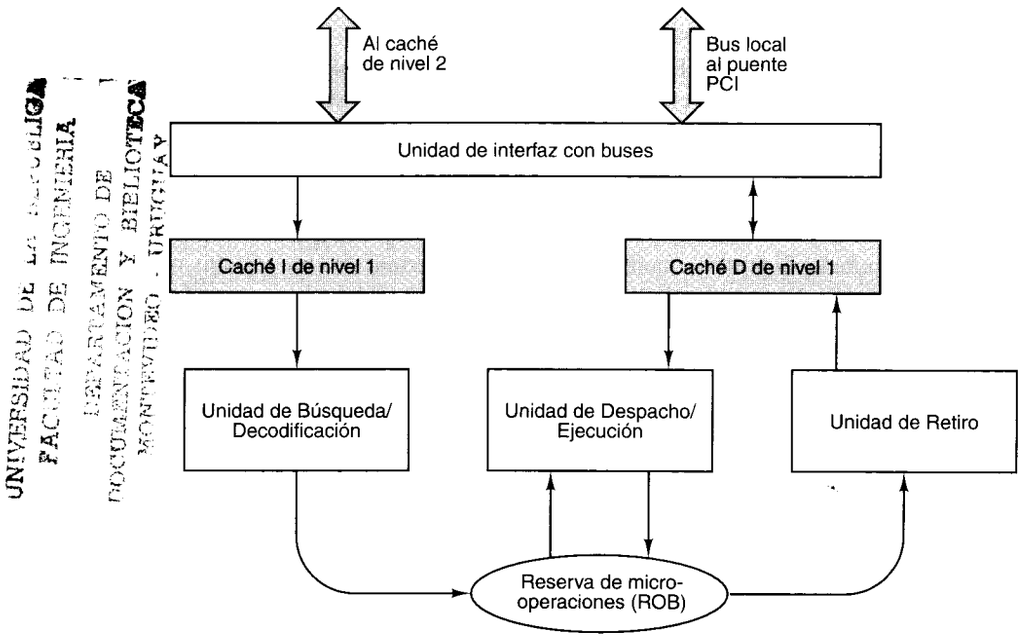


Figura 4-46. Microarquitectura del Pentium II.

La unidad de interfaz con buses se encarga de comunicarse con el sistema de memoria, tanto la caché de nivel 2 como la memoria principal. La caché de nivel 2 no está conectado al bus local, por lo que la unidad de interfaz con buses se encarga de buscar los datos de la

memoria principal a través del bus local y de cargar todas las cachés. El Pentium II usa el protocolo de coherencia de cachés MESI, que describiremos cuando lleguemos a los multiprocesadores en la sección 8.3.2.

La unidad de Búsqueda/Decodificación

La unidad de Búsqueda/Decodificación usa filas de procesamiento intensivamente, con siete etapas rotuladas IFU0 a ROB en la figura 4-47 (Las unidades de Despacho/Ejecución y de Retiro tienen otras cinco, lo que hace que las filas de procesamiento tengan 12 etapas en total.) Las instrucciones ingresan en la fila de procesamiento en la etapa IFU0, donde se cargan líneas de 32 bytes enteras de la caché I. Siempre que el buffer interno está vacío, se introduce otra línea de caché. El registro NEXT IP guía el proceso de búsqueda.

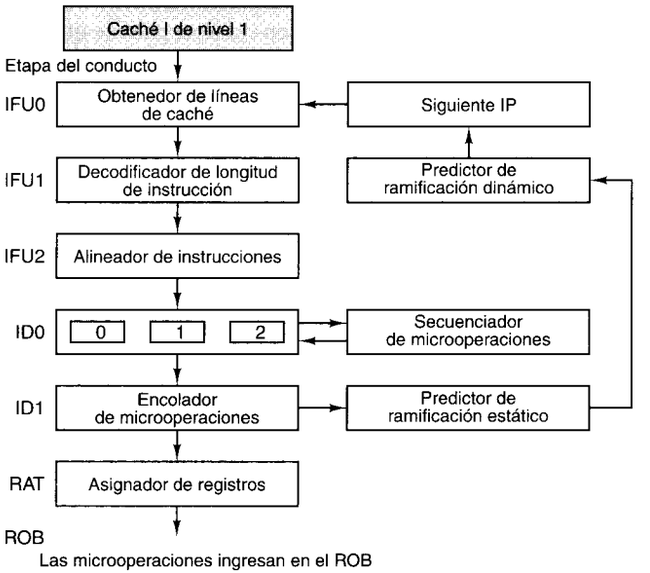


Figura 4-47. Estructura interna de la unidad de Búsqueda/Decodificación (simplificada).

Puesto que el conjunto de instrucciones Intel, conocido como **IA-32** tiene instrucciones de longitud variable con muchos formatos, la siguiente etapa de la fila de procesamiento, IFU1, analiza el flujo de bytes para localizar el principio de cada instrucción. Si es necesario, IFU1 puede examinar hasta 30 instrucciones IA-32 por adelantado. Lo malo es que adelantarse tanto implica encontrar cuatro o cinco ramificaciones condicionales en promedio, no todas las cuales pueden predecirse correctamente, por lo que el valor de poder ver tan lejos en el futuro es dudoso. La etapa IFU2 alinea las instrucciones para que la siguiente etapa pueda decodificarlas con facilidad.

La decodificación inicia en la etapa ID0. Decodificar en el Pentium II consiste en convertir cada instrucción IA-32 en una o más microoperaciones, como hacía el Mic-4. Las instrucciones IA-32 más sencillas, como las transferencias de registro a registro, se pueden traducir a una microoperación. Las más complejas pueden requerir hasta cuatro microoperaciones. Unas cuantas instrucciones extremadamente complejas pueden requerir aún más, y utilizar la ROM del secuenciador de microoperaciones para determinar la secuencia.

La etapa ID0 tiene tres decodificadores internos. Dos de ellos son para instrucciones sencillas; el tercero maneja las demás. Lo que sale de la etapa ID0 es una secuencia de microoperaciones. Cada una contiene un código de operación, dos registros fuente y un registro de destino.

Las microoperaciones colocadas en la cola de espera en la etapa ID1, análoga a la unidad de cola de espera de la figura 4-35. Esta etapa también detecta ramificaciones. Primero viene una predicción estática, por si acaso. La predicción depende de varios factores, pero en el caso de ramificaciones relativas a la instrucción en curso se predice que las ramas hacia atrás se tomarán y las ramas hacia adelante no. Después viene el predictor de ramificación dinámico que emplea un algoritmo basado en el historial, como el de la figura 4-42, sólo que en lugar de usar sólo 2 bits de historial usa 4. Debe ser evidente que con una fila de procesamiento de 12 etapas el castigo por predecir mal es enorme, de ahí el uso de tantos bits de historial. Si la ramificación no está en la tabla histórica, se usa la predicción estática.

Para evitar dependencias WAR y WAW, el Pentium II usa cambio de nombres de registro, como vimos en la figura 4-44. Los registros reales nombrados en las instrucciones IA-32 se pueden sustituir en las microoperaciones por cualesquiera de 40 registros de borrador internos ubicados en el ROB. Este cambio de nombres se efectúa en la etapa RAT.

Por último, las microoperaciones se depositan en el ROB a razón de tres por ciclo de reloj. Los operandos también se reúnen ahí, si están disponibles. Si los operandos y el registro de resultado de una microoperación están disponibles, y la unidad de ejecución está libre, la microoperación es elegible para emitirse; si no, permanece en el ROB hasta que se han adquirido todos sus recursos.

La unidad de Despacho/Ejecución

Ahora llegamos a la unidad de Despacho/Ejecución, que se ilustra en la figura 4-48. Esta unidad planifica y ejecuta las microoperaciones, resolviendo dependencias y conflictos de recursos. Aunque sólo se pueden decodificar tres instrucciones ISA por ciclo de reloj (en ID0), se pueden emitir hasta cinco microoperaciones para ejecutarse en un ciclo, una por cada puerto. Este ritmo no puede sostenerse porque excede la capacidad de la unidad de Retiro. Las microoperaciones se pueden emitir en desorden, pero la unidad de Retiro las retira en orden. Se emplea un marcador complejo para llevar la contabilidad de las microoperaciones pendientes, registros y unidades de ejecución. Cuando una microoperación es elegible para ejecutarse, se le puede iniciar, aunque otras que se colocaron antes en el ROB no estén listas. Cuando hay varias microoperaciones elegibles para ejecutarse en la misma unidad de ejecución, un algoritmo complejo escoge la que más conviene emitir a continuación. Por ejemplo,

es mucho más importante que se ejecute una ramificación que una instrucción aritmética, ya que la primera afecta el flujo del programa.

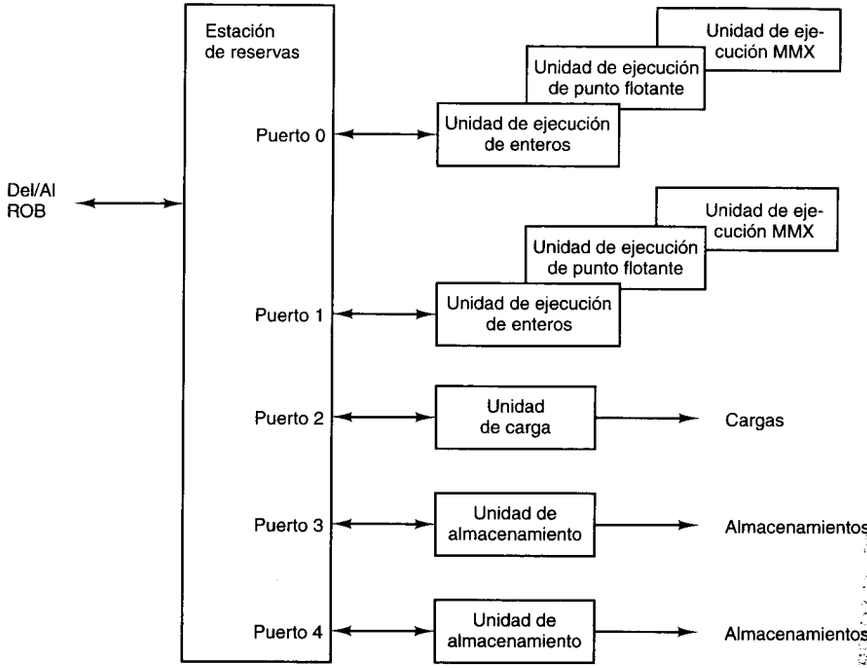


Figura 4-48. La unidad de Despacho/Ejecución.

La unidad de Despacho/Ejecución contiene una estación de reservas y unidades de ejecución conectadas a cinco puertos. La estación de reservas es una cola con 20 entradas para las microoperaciones que ya tienen todos sus operandos. Éstas esperan su turno en la estación de reservas hasta que la unidad de ejecución que necesitan se desocupa.

Hay cinco puertos que van de la estación de reserva a las unidades de ejecución. Algunas unidades de ejecución comparten un mismo puerto, como se muestra. Las unidades de carga y almacenamiento establecen la información apropiada para las operaciones de carga y almacenamiento, respectivamente. Hay dos puertos para los almacenamientos. Puesto que sólo puede emitirse una microoperación por puerto por ciclo, si dos microoperaciones pendientes necesitan pasar por el mismo puerto una de ellas tendrá que esperar.

La unidad de Retiro

Una vez que se ha ejecutado una microoperación, regresa a la estación de reservas y luego al ROB para esperar que se le retire. La unidad de Retiro se encarga de enviar los resultados al

lugar apropiado: al registro correcto, pero también a otras estaciones de la unidad de Despacho/Ejecución que están esperando ese valor. La unidad de Búsqueda/Decodificación mantiene los registros "oficiales", es decir, los valores producidos por las instrucciones que ya terminaron. La unidad de Retiro mantiene un conjunto de registros pendientes, es decir, los valores que se calcularon en una instrucción que no ha terminado porque instrucciones previas todavía no terminan.

El Pentium II apoya plenamente la ejecución especulativa, por lo que algunas instrucciones se habrán ejecutado en vano y no se retirarán. Aquí es donde reside la capacidad de reversión. Si se determina que alguna microoperación provino de una instrucción IA-32 que no debió haberse ejecutado, sus resultados se desechan. Corresponde a la unidad de Retiro seguir la pista a todo esto. Sólo las instrucciones ejecutadas "oficialmente" pueden retirarse, y deben retirarse en el orden del programa, aunque se hayan ejecutado fuera de orden.

4.6.2 La microarquitectura de la CPU UltraSPARC II

La serie UltraSPARC es la implementación hecha por Sun de la arquitectura SPARC Versión 9. Los diversos modelos son muy similares, y difieren principalmente en su desempeño y su precio. No obstante, para evitar confusiones nos referiremos al "UltraSPARC II" en toda esta sección y describiremos sus características en las áreas en que difieren.

El UltraSPARC II es una máquina cabal de 64 bits, con registros de 64 bits y una trayectoria de datos de 64 bits, aunque por razones de compatibilidad hacia atrás con SPARC Versión 8 (es decir, de 32 bits), también puede manejar operandos de 32 bits y, de hecho, ejecutar software para SPARC de 32 bits sin modificación. Aunque la arquitectura interna es de 64 bits, el bus de memoria tiene una anchura de 128 bits, análogo a la arquitectura de 32 bits del Pentium II con bus de memoria de 64 bits, pero una generación más adelante de la CPU misma. En la figura 3-47 se muestra el corazón del sistema UltraSPARC II.

Toda la serie SPARC ha sido un diseño RISC desde que nació. Casi todas las instrucciones tienen dos registros fuente y un registro de destino, por lo que se prestan a la ejecución en filas de procesamiento en un ciclo. No hay necesidad de descomponer instrucciones CISC antiguas en microoperaciones RISC, como tiene que hacer el Pentium II.

El UltraSPARC II es una máquina superescalar que puede emitir cuatro instrucciones por ciclo de reloj de forma sostenida indefinidamente. Las instrucciones se emiten en orden pero pueden terminar fuera de orden. No obstante, las interrupciones son precisas (es decir, no hay ambigüedad respecto a dónde estaba exactamente la máquina cuando ocurrió una interrupción). Existe apoyo de hardware para cargas especulativas en forma de una instrucción PREFETCH (prebúsqueda) que no causa un fallo cuando no encuentra lo que busca en la caché. De hecho, ni siquiera se bloquean los accesos subsecuentes a la memoria. Por tanto, un compilador puede plantar uno o más PREFETCH en el flujo de código mucho antes del punto en el que se necesitan con la esperanza de tener los datos ahí cuando se necesiten, pero sin ser castigado en caso de ocurrir un fallo de caché.

Generalidades del UltraSPARC II

En la figura 4-49 se presenta un diagramas de bloques del UltraSPARC II. Todos los componentes que se muestran están en el chip de la CPU excepto la caché de nivel 2, que es totalmente externo. La caché I es una caché asociativa de 16 KB de dos vías con una línea de 32 bytes, pero que puede devolver media línea de caché. Media línea (16 bytes) contiene exactamente cuatro instrucciones, todas las cuales pueden emitirse en un solo ciclo de reloj. La caché D es de 16 KB, con mapeo directo, escritura a través y sin asignación, que también usa líneas de 32 bytes divididas en dos sublíneas de 16 bytes.

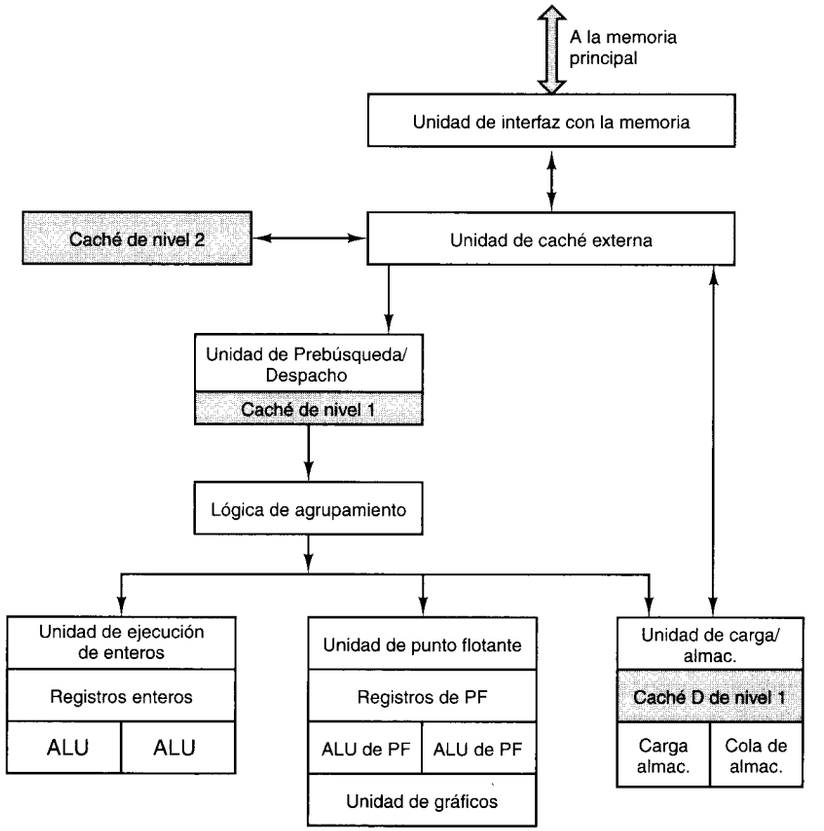


Figura 4-49. La microarquitectura de UltraSPARC II.

La unidad de caché externa maneja los fallos de caché de nivel 1 buscando la línea requerida en la caché externa (de nivel 2). Si la búsqueda tiene éxito, la línea se copia en la caché de nivel 1 apropiado. Si no, la unidad de caché externa pide a la unidad de interfaz con la memoria que busque la línea de la memoria principal.

Veamos ahora las unidades funcionales del UltraSPARC II. La unidad de Prebúsqueda/Despacho es similar a la unidad de Búsqueda/Decodificación del Pentium II (vea la figura 4-46). Sin embargo, su tarea es más fácil porque las instrucciones que llegan ya son microoperaciones de tres registros, así que no necesitan desglosarse. La unidad puede buscar instrucciones de la caché de nivel 1 o de la caché de nivel 2 sin pérdida de tiempo. Se traen cuatro instrucciones por ciclo de reloj.

Para reducir el efecto de las ramificaciones, cada grupo de cuatro instrucciones de la caché I tiene una dirección que indica cuál media línea debe tomarse a continuación. Además, hay un predictor de ramificaciones completo dentro de la unidad de Prebúsqueda/Despacho que usa un algoritmo de predicción de 2 bits similar al de la figura 4-42. Además, el UltraSPARC II tiene un conjunto de instrucciones de ramificación que permiten al compilador decirle al hardware qué predicción debe hacer, ya que en muchos casos tiene una muy buena idea de qué rama se tomará. Las instrucciones prebuscadas se almacenan en una cola de 12 entradas que alimenta a la lógica de agrupamiento.

La lógica de agrupamiento es una unidad que selecciona de la cola cuatro instrucciones a la vez para emitirlas. La cuestión es encontrar cuatro que se puedan emitir simultáneamente. La unidad de ejecución de enteros tiene dos ALU discretas, lo que le permite ejecutar dos instrucciones en paralelo. De igual manera, la unidad de punto flotante también tiene dos ALU de PF. Por tanto, un grupo podría contener dos de cada tipo, pero no cuatro de un mismo tipo. Para que el agrupamiento sea óptimo, las instrucciones se deben emitir en desorden, lo cual está permitido. El retiro también es entonces en desorden.

Las unidades de enteros y de punto flotante contienen sus respectivos registros, así que las instrucciones recogen sus operandos dentro de la unidad y dejan sus resultados ahí también. Los registros de enteros y de punto flotante son distintos, por lo que nunca hay transferencia de valores de una unidad a la otra. La unidad de punto flotante contiene además una unidad de gráficos que ejecuta instrucciones especiales para procesar imágenes 2D y 3D, audio y video, análogas a las instrucciones MMX del Pentium II.

La unidad de Carga/Almacenamiento maneja las instrucciones LOAD y STORE. Si hay un acierto en la caché D de nivel 1, la ejecución es inmediata; si no, se pide a la unidad de caché externa que obtenga la línea de la caché de nivel 2 o de la memoria principal (en un fallo de LOAD) o escriba la palabra ahí (en un fallo de STORE). Si la caché D hubiera sido de asignación de escritura, en un fallo de STORE la línea de caché requerida se habría traído de la caché de nivel 2 o de la memoria principal. En cambio, la palabra que se va a almacenar simplemente se escribe a través de la caché de nivel 2 o, si no acierta ahí, en la memoria principal. Para evitar un bloqueo causado por un fallo de la caché D, la unidad de Carga/Almacenamiento mantiene colas (individuales) de instrucciones LOAD y STORE pendientes para poder seguir procesando nuevas instrucciones mientras espera que las anteriores terminen.

La fila de procesamiento del UltraSPARC II

El UltraSPARC II tiene una fila de procesamiento de nueve etapas, algunas de las cuales son diferentes para las instrucciones con enteros y de punto flotante, como se ilustra en la figura 4-50. La primera etapa busca instrucciones de la caché I (si es posible). En condiciones

ideales (sin fallos de caché, sin errores de predicción, sin instrucciones difíciles, la combinación perfecta de instrucciones, etc.), esta etapa puede seguir trayendo y emitiendo cuatro instrucciones por ciclo indefinidamente. La etapa de decodificación añade algunos bits extra a cada instrucción antes de depositarla en la cola de instrucciones de 12 entradas que mencionamos antes. Estos bits agilizan el procesamiento subsecuente (por ejemplo, indicando la unidad de ejecución apropiada).

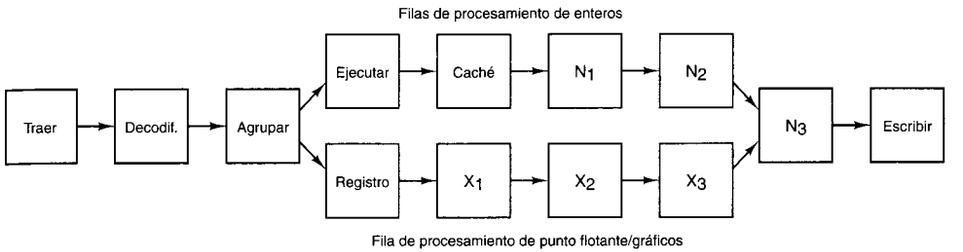


Figura 4-50. Fila de procesamiento del UltraSPARC II.

La etapa de agrupamiento corresponde a la unidad de lógica de agrupamiento que vimos antes. Esta etapa examina las instrucciones decodificadas en busca de grupos de cuatro que sean compatibles para emitirse simultáneamente.

En este punto las filas de procesamiento para enteros y para punto flotante se dividen. La etapa 4 de la unidad de enteros ejecuta casi todas las instrucciones directamente en un ciclo. Sin embargo, las instrucciones LOAD y STORE requieren procesamiento adicional en la etapa de caché. Las etapas N_1 y N_2 no hacen nada en el caso de instrucciones de registros pero se incluyen para mantener las dos filas de procesamiento sincronizadas. Que las instrucciones con enteros terminen unos cuantos nanosegundos más tarde es un precio pequeño que se paga por mantener un flujo continuo en las filas de procesamiento.

La unidad de punto flotante tiene cuatro etapas propias que inician aquí. La primera es para acceder a los registros de punto flotante; las tres siguiente son para ejecutar la instrucción. Todas las instrucciones de punto flotante se ejecutan en tres ciclos, con excepción de la división (12 ciclos) y la raíz cuadrada (22 ciclos), por lo que una serie larga de las demás instrucciones de punto flotante no frena la fila de procesamiento.

La etapa N_3, que es común a ambas unidades, sirve para resolver condiciones de excepción que podrían haber ocurrido, como la división entre cero. La última etapa es en la que se escriben los resultados en los registros. Esta etapa es comparable a la unidad de Retiro del Pentium II en cuanto a que una vez que una instrucción ha pasado por esta etapa, está confirmada.

4.6.3 La microarquitectura de la CPU picoJava II

El picoJava II puede ejecutar programas JVM binarios sin (mucha) interpretación en software. Casi todas las instrucciones JVM son ejecutadas directamente por el hardware en un solo ciclo de reloj. Cerca de 30 instrucciones JVM están microprogramadas. Unas pocas

instrucciones no pueden ser ejecutadas por el hardware de picoJava II y causan saltos a un intérprete en software. Estas instrucciones por lo regular tienen que ver con características de JVM que no hemos estudiado, como crear y manejar objetos de software complejos.

Generalidades del picoJava II

En la figura 4-51 se muestra un diagrama de bloques de la microarquitectura del picoJava II. Hay una caché de nivel 1 dividido en el chip. La caché I es opcional y puede tener tamaños de 1 KB, 2 KB, 4 KB, 8 KB o 16 KB si está presente; es de mapeo directo, con un tamaño de línea de 16 bytes. La caché D también es opcional y también puede tener tamaños de 1 KB, 2 KB, 4 KB, 8 KB o 16 KB si está presente; es asociativa por conjuntos de dos vías, y también tiene un tamaño de línea de 16 bytes. La caché D usa asignación de escritura y reescritura. Cada caché tiene una trayectoria de 32 bits de anchura al bus de memoria. El microJava 701 tiene ambas cachés presentes, cada una con los 16 KB completos. Una unidad de punto flotante opcional también forma parte del diseño del picoJava II.

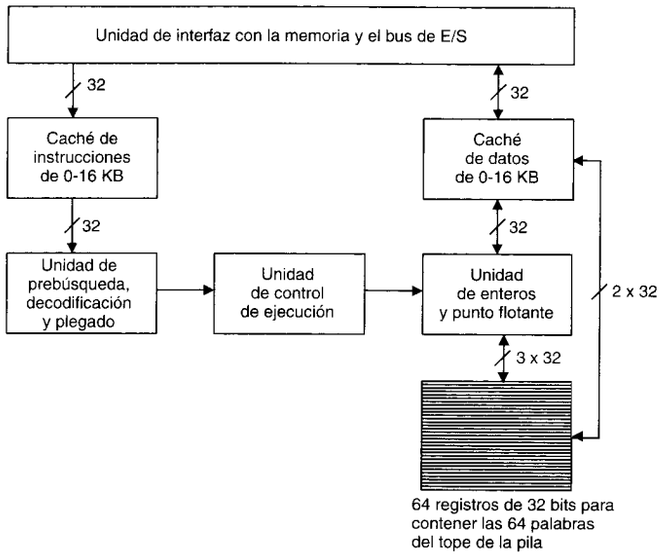


Figura 4-51. Diagrama de bloques del picoJava II con ambas cachés de nivel 1 y la unidad de punto flotante. Ésta es la configuración del microJava 701.

La caché I alimenta la unidad de búsqueda, decodificación y plegado de 8 en 8 bytes. Esta unidad, a su vez, está conectada al controlador de la ejecución y a la trayectoria de datos principal (unidad de enteros/punto flotante). La trayectoria de datos tiene una anchura de 32 bits para las operaciones con enteros, pero también puede manejar punto flotante IEEE 754 con precisión sencilla y doble.

La parte más interesante de la figura 4-51 es un archivo de registros que consiste en 64 registros de 32 bits. Estos registros pueden contener las 64 palabras superiores de la pila de la máquina JVM, lo que agiliza considerablemente el acceso a las palabras de la pila. Tanto la pila de operandos como la pila de variables locales que está inmediatamente abajo de ella se pueden guardar en el archivo de registros. Los accesos al archivo de registros son "gratuitos" (es decir, ocurren sin causar retraso, mientras que los accesos a la caché D requieren un ciclo extra). La trayectoria entre la unidad de enteros/punto flotante tiene 96 bits de anchura y puede manejar dos lecturas de 32 bits de la pila y una escritura de 32 bits en la pila en un solo ciclo.

Por ejemplo, si la pila de operandos consiste en dos palabras, puede haber hasta 62 palabras de variables locales en el archivo de registros. Desde luego, si se mete otra palabra en la pila, surge un problema. Lo que sucede es que una o más palabras del fondo de la pila se reescriben a la memoria por un proceso llamado **goteo**. De igual manera, si se sacan varias palabras del tope de la pila de operandos, quedará espacio disponible en el archivo de registros, y podrán volverse a cargar en él algunas palabras de las profundidades de la pila. Registros especiales del chip determinan qué tan lleno tiene que estar el archivo de registros antes de que algunas palabras del fondo se escriban en la memoria, y qué tan vacío tiene que estar antes de que se vuelva a cargar desde la memoria. Para facilitar el goteo sin copiado, el archivo de registros opera como un buffer circular, con apuntadores a la palabra válida más baja y a la más alta. El goteo es automático, y ocurre siempre que el archivo de registros está demasiado lleno o demasiado vacío.

La fila de procesamiento del picoJava II

El picoJava II tiene una fila de procesamiento de seis etapas que se ilustra en la figura 4-52. La primera etapa busca instrucciones de la caché I de 8 en 8 bytes y las coloca en un buffer de instrucciones cuya capacidad es de 16 bytes. La siguiente etapa decodifica las instrucciones y las pliega (combina) de ciertas maneras que describiremos en breve. Lo que sale de la unidad decodificadora es una secuencia de microoperaciones, cada una de las cuales contiene un código de operación y tres números de registro (dos registros fuente y un registro de destino). En este sentido, el picoJava II es similar al Pentium II: ambas máquinas aceptan una serie de instrucciones CISC que se desglosan internamente para dar una secuencia de microoperaciones que se alimentan a un núcleo RISC con filas de procesamiento. Sin embargo, a diferencia del Pentium II, el picoJava II no es superescalar y las microoperaciones se emiten y ejecutan en el orden en que se emitieron. Si hay un fallo de caché, si hay que buscar un operando de la memoria principal, la CPU se para y lo espera.

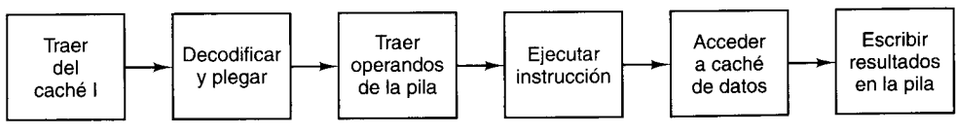


Figura 4-52. El picoJava II tiene una fila de procesamiento de seis etapas.

La tercera etapa de la fila de procesamiento busca operandos de la pila (en realidad del archivo de registros) para que estén listos para la cuarta etapa, la unidad de ejecución, donde

se llevan a cabo las instrucciones. La quinta etapa accede a la caché de datos si es necesario; por ejemplo, para guardar resultados ahí. Por último, la sexta etapa escribe resultados de vuelta en la pila.

Plegado de instrucciones

Como dijimos antes, la unidad decodificadora puede **plegar** varias instrucciones para juntarlas. Como ejemplo de este proceso, consideremos la evaluación del enunciado

```
n = k + m;
```

La traducción a (I)JVM podría ser

```
ILOAD 7
ILOAD 1
IADD
ISTORE 3
```

suponiendo que *k*, *m* y *n* son las variables locales 7, 1 y 3, respectivamente, la ejecución de estas cuatro instrucciones sería como se muestra en la figura 4-53(a).

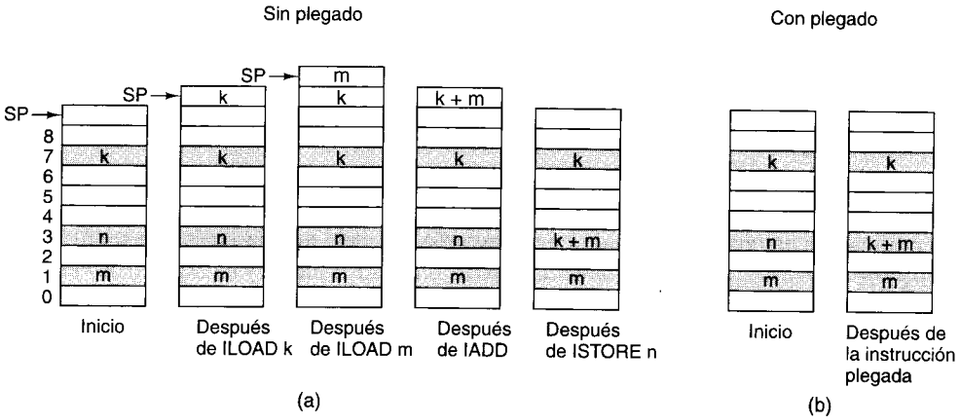


Figura 4-53. (a) Ejecución de una secuencia de cuatro instrucciones para calcular n = k + m. (b) La misma secuencia plegada en una instrucción.

Suponiendo que las tres variables están lo bastante alto en la pila como para que todas estén contenidas en el archivo de registros, no es necesario hacer ninguna referencia a la memoria para ejecutar esta secuencia de instrucciones. La primera instrucción, ILOAD 7 copia en el tope de la pila la palabra que está en la séptima variable local e incrementa el apuntador de la pila. Igualmente, ILOAD 1 es también una copia de registro a registro. IADD suma dos registros e ISTORE copia un registro en la porción de variables locales del archivo de registros. Ya de por sí, ahorrarse todas las referencias a la memoria es una importante optimación del desempeño.

Sin embargo, el picoJava II hace mucho más. Lo que la secuencia de cuatro instrucciones en realidad hace es sumar dos registros y guardar el resultado en un tercer registro. La unidad decodificadora detecta esta condición y emite una sola microoperación: una instrucción ADD de tres registros. Así, en lugar de tener cuatro instrucciones JVM que hacen un total de nueve referencias a la memoria, tenemos una sola microoperación de tres registros para efectuar una suma, como se muestra en la figura 4-53(b). El resultado neto es que, aunque instrucciones CISC con muchas referencias a la memoria ingresaron en la fila de procesamiento, sólo se ejecutó realmente una microoperación sencilla. Por tanto, el picoJava II puede ejecutar programas Java compilados para JVM con la misma rapidez que si se compilaran para el lenguaje máquina de una máquina RISC de extremo superior.

Como acabamos de ver, la capacidad para plegar varias instrucciones JVM en una microoperación es la clave para la ejecución de alto desempeño con filas de procesamiento. Por ello, vale la pena examinar brevemente cómo la unidad decodificadora realiza el plegado. Para fines de plegado, las instrucciones se dividen en seis grupos, que se enumeran en la figura 4-54. El primer grupo contiene instrucciones que no se pliegan. El segundo contiene cargas de variables locales. IJVM tiene una variable de este tipo, ILOAD, pero JVM tiene otras además. El tercer grupo contiene almacenamientos, como por ejemplo ISTORE. El cuarto y quinto grupos son para ramificaciones, con uno y dos operandos, respectivamente. El último grupo consiste en instrucciones que sacan dos operandos de la pila, realizan algún cálculo con ellos, y meten el resultado otra vez en la pila.

Grupo	Descripción	Ejemplo
NF	Instrucciones no plegables	GOTO
LV	Meter una palabra en la pila	ILOAD
MEM	Sacar una palabra y guardarla en la memoria	ISTORE
BG1	Operaciones que usan un operando de la pila	IFEQ
BG2	Operaciones que usan dos operandos de la pila	IF_CMPEQ
OP	Cálculos con dos operandos y un resultado	IADD

Figura 4-54. Grupos de instrucciones JVM para fines de plegado.

La unidad decodificadora emite microoperaciones de 74 bits a la máquina de ejecución (vía la unidad de búsqueda de operandos). Casi todas éstas contienen un código de operación y tres registros y se pueden ejecutar en un solo ciclo. A medida que la unidad decodificadora trae instrucciones JVM del buffer de instrucciones, las convierte en sucesiones de microoperaciones, pero al mismo tiempo busca ciertos patrones de secuencias de instrucciones JVM que puedan plegarse en una sola microoperación. Los patrones pueden tener hasta cuatro instrucciones de longitud. Cuando se encuentra un patrón, se emite la microoperación correspondiente y se desechan las instrucciones originales.

En la figura 4-55 se presenta una tabla con algunas secuencias JVM comunes que se pueden plegar. Cuando la unidad decodificadora encuentra una de estas secuencias, sustituye la descomposición de instrucciones acostumbrada por una sola microoperación que realiza el

trabajo de toda la secuencia en un ciclo. Por ejemplo, una secuencia de cuatro instrucciones se convierte en una sola microoperación ADD de tres registros, como vimos en la figura 4-53. El plegado sólo funciona cuando las variables locales a las que se hace referencia están lo bastante cerca del tope de la pila como para estar contenidas en el archivo de registros.

Secuencia de instrucciones				Ejemplo
LV	LV	OP	MEM	ILOAD, ILOAD, IADD, ISTORE
LV	LV	OP		ILOAD, ILOAD, IADD
LV	LV	BG2		ILOAD, ILOAD, IF_CMPEQ
LV	BG1			ILOAD, IFEQ
LV	BG2			ILOAD, IF_CMPEQ
LV	MEM			ILOAD, ISTORE
OP	MEM			IADD, ISTORE

Figura 4-55. Algunas secuencias de instrucciones JVM que se pueden plegar.

El plegado ocurre con suficiente frecuencia como para que una fracción sustancial de un programa JVM se pueda ejecutar como si se hubiera compilado directamente para un procesador RISC con filas de procesamiento, lo que le confiere la transportabilidad de JVM combinada con el alto desempeño de una máquina RISC. Mediciones realizadas muestran que el picoJava II ejecuta programas en Java hasta cinco veces más rápidamente que los mismos programas compilados a lenguaje máquina en un Pentium con la misma frecuencia de reloj, y hasta 15 veces más rápidamente que si se ejecutaran interpretados en un Pentium.

El picoJava II tiene un algoritmo de predicción de ramas muy primitivo: siempre predice que no se tomará la rama. La justificación para esto fue mantener el chip sencillo y económico en lugar de dedicar una porción sustancial del área del chip a circuitos de predicción. Además, debido a la longitud del conducto (seis etapas, en lugar de 12 en el Pentium II), el castigo por predecir erróneamente es de sólo tres ciclos.

4.6.4 Comparación del Pentium, UltraSPARC y picoJava

Nuestros tres ejemplos son muy diferentes pero es asombroso lo mucho que tienen en común en el fondo, y esto dice algo acerca de la forma óptima de diseñar una computadora. El Pentium II tiene un conjunto de instrucciones CISC prehistórico que a los ingenieros de Intel les encantaría tirar a la bahía de San Francisco, sólo que ello violaría las leyes contra la contaminación del agua de California. El UltraSPARC II es un diseño RISC puro, con instrucciones magras y potentes. El picoJava II es una máquina de pila, con instrucciones de longitud variable que hacen una enormidad de referencias a memoria (que en principio son costosas).

A pesar de estas grandes diferencias, las tres máquinas tienen unidades de ejecución muy similares. Todas las unidades de ejecución aceptan microoperaciones que contienen

un código de operación, dos registros fuente y un registro de destino. Todas ellas pueden ejecutar una microoperación en un ciclo; todas ellas tienen filas de procesamiento profundas y predicción de ramificaciones, y todas ellas tienen cachés I y D divididos, con 16 KB por caché.

Esta similitud interna no es accidental y ni siquiera se debe al incesante saltar de una empresa a otra de los ingenieros del Valle del Silicio. Como vimos con nuestros ejemplos Mic-3 y Mic-4, es fácil y natural construir una trayectoria de datos con filas de procesamiento que tome dos registros fuente, los pase por una ALU y guarde el resultado en un registro. La figura 4-34 muestra esta fila de procesamiento gráficamente. Con la tecnología actual, éste es el diseño más eficaz.

La diferencia primordial entre el Pentium II, el UltraSPARC II y el picoJava II es la forma como llegan de su conjunto de instrucciones de ISA a la unidad de ejecución. El Pentium II tiene que descomponer sus instrucciones CISC para ponerlas en el formato de tres registros que la unidad de ejecución necesita. De eso se trata la figura 4-47: partir instrucciones grandes en microoperaciones simples y precisas. El picoJava II tiene el problema opuesto: cómo combinar varias instrucciones para obtener microoperaciones simples y claras. Eso es lo que hace el plegado. El UltraSPARC II no tiene que hacer nada porque sus instrucciones nativas ya son microoperaciones. Es por esto que casi todas las ISA nuevas son del tipo RISC, a menos que tengan algún motivo ulterior (como bajar programas en Java de Internet y ejecutarlos en una máquina arbitraria).

Aprenderemos algo si comparamos nuestro diseño final, el Mic-4, con estos tres ejemplos del mundo real. El Mic-4 se parece más al Pentium II. Ambos tienen que interpretar un conjunto de instrucciones ISA no RISC. Ambos lo hacen descomponiendo las instrucciones ISA en microoperaciones que tienen un código de operación, dos registros fuente y un registro de destino. En ambos casos, las microoperaciones se depositan en una cola para ejecutarse posteriormente. El Mic-4 tiene un diseño estricto de emisión en orden, ejecución en orden y retiro en orden, mientras que el Pentium II tiene una política de emisión en orden, ejecución en desorden y retiro en orden. La estructura interna de la fila de procesamiento del Pentium II, sobre todo la parte que se muestra en la figura 4-47, también tiene algo en común con el diseño del Mic-4.

Comparemos ahora el Mic-4 con el picoJava II. Aunque parece lógico que las dos arquitecturas sean muy similares —después de todo, interpretan el mismo conjunto de instrucciones ISA— no son en realidad tan parecidas. La razón es que las unidades decodificadoras tienen estrategias diametralmente opuestas. El Mic-4 toma cada instrucción IJVM que llega e inmediatamente la descompone en microoperaciones, mientras que el picoJava II trata de combinar (plegar) varias instrucciones JVM en una sola microoperación. La sencilla asignación i = j + k tarda 14 ciclos en el Mic-4 y un ciclo en el picoJava II. En este caso, el plegado gana por un factor de 14. Obviamente, el segundo método da pie a una ejecución mucho más rápida, pero la complejidad del proceso de plegado es sustancial.

El Mic-4 y el UltraSPARC II no se pueden comparar realmente porque el segundo tiene instrucciones RISC (es decir, microoperaciones de tres registros) como conjunto de instrucciones ISA. No hay necesidad de descomponerlas ni de combinarlas; se pueden ejecutar tal cual, cada una en un solo ciclo de la trayectoria de datos.

Las cuatro máquinas hacen uso intensivo de filas de procesamiento. El Pentium II tiene 12 etapas, el UltraSPARC II tiene nueve, el picoJava II tiene seis y el Mic-4 tiene siete. El Pentium II tiene unas pocas más que los otros porque sus instrucciones no son fáciles de descomponer. El UltraSPARC II tiene más de las que realmente necesita porque la fila de procesamiento de enteros se alargó artificialmente en dos etapas para hacer que las operaciones con enteros tardaran el mismo tiempo que las de punto flotante. La conclusión aquí es que con la tecnología moderna y un conjunto de instrucciones razonable, lo más recomendable es una fila de procesamiento de seis o siete etapas que procese microoperaciones de tres registros. El Mic-4 nos da una buena idea de cómo podría funcionar una fila de procesamiento semejante (al menos con un conjunto de instrucciones no RISC).

4.7 RESUMEN

El corazón de una computadora es la trayectoria de datos. Éste contiene algunos registros, dos o tres buses y una o más unidades funcionales como ALU y desplazadores. El ciclo de ejecución principal consiste en buscar algunos operandos de los registros y enviarlos por los buses a la ALU y otras unidades funcionales para efectuar la ejecución. Luego, los resultados se vuelven a guardar en registros.

La trayectoria de datos puede controlarse con un secuenciador que busca microinstrucciones de un almacén de control. Cada microinstrucción contiene bits que controlan la trayectoria de datos durante un ciclo. Dichos bits especifican qué operandos hay que seleccionar, cuál operación debe efectuarse y qué debe hacerse con los resultados. Además, cada microinstrucción especifica su sucesora, por lo regular de forma explícita al contener su dirección. Algunas microinstrucciones modifican esta dirección base calculando el OR de ciertos bits y la dirección antes de usarla.

La máquina IJVM es una máquina de pila con códigos de operación de un byte que almacena palabras en la pila, sacan palabras de la pila y combinan (por ejemplo, suman) palabras en la pila. Se dio una implementación microprogramada para la microarquitectura Mic-1. Mediante la adición de una unidad para buscar las instrucciones y precargar los bytes en el flujo de instrucciones, fue posible eliminar muchas referencias al contador de programa y hacer la máquina mucho más rápida.

Hay muchas maneras de diseñar el nivel de microarquitectura. Existen muchos puntos de equilibrio, como los diseños de dos buses *versus* tres buses, campos de microinstrucción codificados *versus* decodificados, presencia o ausencia de prebúsqueda, filas de procesamiento profundas o someras, y mucho más. El Mic-1 es una máquina sencilla, controlada por software, con ejecución secuencial y sin paralelismo. En contraste, el Mic-4 tiene una microarquitectura altamente paralela y una fila de procesamiento de siete etapas.

Existen varias formas de mejorar el desempeño. La memoria caché es uno de los más importantes. Las cachés con mapeo directo y de conjunto asociativo se usan comúnmente para agilizar las referencias a la memoria. La predicción de ramificaciones, tanto estática como dinámica, es importante, lo mismo que la ejecución fuera de orden y la ejecución especulativa.

Nuestras tres máquinas de ejemplo, el Pentium II, el UltraSPARC II y el picoJava II, son muy diferentes en lo exterior pero tienen unidades de ejecución sorprendentemente similares. El Pentium II toma un conjunto de instrucciones tipo CISC y descompone las instrucciones en microoperaciones que se procesan en un núcleo superescalar con predicción de ramificaciones, ejecución en desorden y especulación. El UltraSPARC II es una CPU moderna de 64 bits con instrucciones tipo RISC que también tiene predicción de ramificaciones, ejecución en desorden y especulación. El picoJava II es una máquina más sencilla diseñada para aparatos de bajo costo, por lo que carece de algunas funciones de extremo superior como predicción dinámica de ramificaciones. No obstante, gracias a que pliega las instrucciones, puede ejecutar instrucciones JVM casi como si fueran instrucciones RISC orientadas a registros. Las tres máquinas tienen una unidad de ejecución similar que maneja microoperaciones de tres registros que fluyen por una fila de procesamiento profunda.

PROBLEMAS

1. En la figura 4-2 se muestra una forma de producir *A* como salida de la ALU. Sugiera otra forma.
2. En el Mic-1 se requiere 1 ns para preparar MIR, 1 ns para colocar el contenido de un registro en el bus B, 3 ns para operar la ALU y el desplazador, y 1 ns para que los resultados se propaguen de vuelta a los registros. El ancho del pulso de reloj es de 2 ns. ¿Esta máquina puede operar a 100 MHz? ¿Y a 150 MHz?
3. En la figura 4-6 el registro del bus B se codifica en un campo de 4 bits, pero el bus C se representa como mapa de bits. ¿Por qué?
4. En la figura 4-6 hay un cuadro rotulado "Bit alto". Dibuje el diagrama de circuito correspondiente.
5. Cuando el campo JMPC de una microinstrucción está habilitado, se calcula el OR de MBR y NEXT_ADDRESS para formar la dirección de la siguiente microinstrucción. ¿Hay alguna situación en la que sea lógico que NEXT_ADDRESS sea 0x1FF y se use JMPC?
6. Suponga que en el ejemplo de la figura 4-14(a) se añade el enunciado

   ```
   i = 0;
   ```

   después del enunciado if. ¿Qué haría el nuevo código de ensamblador? Suponga que el compilador es optimante.
7. Dé dos traducciones a IJVM diferentes del siguiente enunciado en Java:

   ```
   i = j + k + 4;
   ```
8. Dé el enunciado en Java que produjo el siguiente código IJVM:

   ```
   ILOAD j
   ILOAD k
   ISUB
   BIPUSH 6
   ISUB
   DUP
   IADD
   ISTORE i
   ```

9. En el texto mencionamos que al traducir el enunciado

```
if (Z) goto L1; else goto L2
```

a binario, *L2* tiene que estar en las 256 palabras inferiores del almacén de control. ¿No sería igualmente posible tener *L1* en, digamos 0x40 y *L2* en 0x140? Explique su respuesta.

10. En el microprograma del Mic-1, en if_icmpeq3, MDR se copia en H y en la siguiente línea se le resta TOS. Seguramente sería mejor tener un solo enunciado aquí

```
if+cmpeq3      Z = MDR – TOS; rd
```

¿Por qué no se hace así?

11. ¿Cuánto tarda un Mic-1 de 200 MHz en ejecutar el enunciado Java

```
i = j + k;
```

Calcule su respuesta en nanosegundos.

12. Repita la pregunta anterior, sólo que ahora para un Mic-2 de 200 MHz. Con base en este cálculo, ¿cuánto tardaría en ejecutarse en el Mic-2 un programa que tarda 100 s en el Mic-1?

13. En la máquina JVM completa hay códigos de operación especiales de un byte para cargar las variables locales 0 a 3 en la pila en lugar de usar la instrucción ILOAD general. ¿Cómo debería modificarse IJVM para aprovechar al máximo estas instrucciones?

14. La instrucción ISHR (desplazamiento aritmético entero hacia la derecha) existe en JVM pero no en IJVM. ISHR usa los dos valores del tope de la pila y los sustituye por un solo valor, el resultado. La segunda palabra desde el tope de la pila es el operando a desplazar. Su contenido se desplaza hacia la derecha entre 0 y 31 bits, inclusive, dependiendo del valor de los cinco bits menos significativos de la palabra que está en el tope de la pila (los otros 27 bits de esa palabra no se toman en cuenta). El bit del signo se repite a la izquierda tantos bits como el contenido se desplazó. El código de operación de ISHR es 122 (0x7A).

a. ¿Qué operación aritmética equivale a desplazar a la derecha con una cuenta de 2?
b. Extienda el microcódigo para incluir esta instrucción como parte de IJVM.

15. La instrucción ISHL (desplazamiento entero hacia la izquierda) existe en JVM pero no en IJVM. ISHL usa los dos valores del tope de la pila y los sustituye por un solo valor, el resultado. La segunda palabra desde el tope de la pila es el operando a desplazar. Su contenido se desplaza hacia la izquierda entre 0 y 31 bits, inclusive, dependiendo del valor de los cinco bits menos significativos de la palabra que está en el tope de la pila (los otros 27 bits de esa palabra no se toman en cuenta). Se introducen por la derecha tantos ceros como bits se desplazó el contenido. El código de operación de ISHL es 120 (0x78).

a. ¿Qué operación aritmética equivale a desplazar a la izquierda con una cuenta de 2?
b. Extienda el microcódigo para incluir esta instrucción como parte de IJVM.

16. La instrucción INVOKEVIRTUAL de JVM necesita saber cuántos parámetros tiene. ¿Por qué?

17. Escriba microcódigo para implementar la instrucción POPTWO de JVM en el Mic-1. Esta instrucción obtiene dos palabras del tope de la pila.

18. Implemente la instrucción DLOAD de JVM en el Mic-2. La instrucción tiene un índice de un byte y almacena en la pila la variable local que está en esa localidad. Luego almacena también en la pila la siguiente palabra más alta.

19. Dibuje una máquina de estados finitos para llevar la puntuación en un partido de tenis. Las reglas son las siguientes. Para ganar se necesitan al menos cuatro puntos y es preciso tener al menos dos puntos más que el contrincante. Parta de un estado (0, 0), que indica que nadie tiene puntos todavía. Luego añada un estado (1, 0), que indica que *A* anotó un punto. Rotule la flecha de (0, 0) a (1, 0) con *A*. Ahora añada un estado (0, 1) que indique que *B* anotó, y rotule la flecha de (0, 0) con *B*. Siga agregando estados y flechas hasta incluir todos los posibles estados.

20. Reconsidere el problema anterior. ¿Hay estados que puedan colapsarse sin alterar el resultado del juego? Si los hay, ¿cuáles son equivalentes?

21. Dibuje una máquina de estados finitos para predicción de ramificaciones que sea más tenaz que la figura 4-42. La predicción sólo deberá cambiar después de tres predicciones equivocadas consecutivas.

22. El registro de desplazamiento de la figura 4-27 tiene una capacidad máxima de 6 bytes. ¿Podría construirse una versión más económica de la IFU con un registro desplazador de 5 bytes? ¿Y con uno de 4 bytes?

23. Después de examinar las IFU más económicas en la pregunta anterior, examinemos IFU más costosas. ¿Tendría sentido incluir un registro de desplazamiento mucho más grande en la IFU, digamos de 12 bytes? ¿Por qué sí o por qué no?

24. En el microprograma para el Mic-2, el código de if_icmpeq6 salta a T cuando Z se pone en 1. Sin embargo, el código en T es igual al de goto1. ¿Habría sido posible ir a goto1 directamente? ¿La máquina sería más rápida si así se hiciera?

25. En el Mic-4, la unidad decodificadora establece una correspondencia entre el código de operación IJVM y el índice en ROM en el que están almacenadas las microoperaciones correspondientes. Podría parecer más sencillo omitir la etapa de decodificación y alimentar el código de operación IJVM directamente a la unidad de cola de espera. Ésta podría usar el código de operación IJVM como índice para el ROM, igual que se hace en el Mic-1. ¿Qué defecto tiene este plan?

26. Una computadora tiene una caché de dos niveles. Suponga que el 80% de las referencias a la memoria aciertan en la caché de nivel 1, el 15% aciertan en la caché de segundo nivel y el 5% fallan. Los tiempos de acceso son de 5 ns, 15 ns y 60 ns, respectivamente, y los tiempos para la caché de nivel 2 y la memoria principal comienzan a contar en el momento en que se sabe que se necesitan (por ejemplo, no se inicia un acceso a la caché de nivel 2 sino hasta después de que ocurre un fallo de caché de nivel 1). Calcule el tiempo de acceso promedio.

27. Al final de la sección 4.5.1 dijimos que la asignación de escritura es mejor sólo si es probable que vayan a efectuarse varias escrituras consecutivas en la misma línea de caché. ¿Qué sucedería en el caso de una escritura seguida de varias lecturas? ¿No habría también una ganancia apreciable?

28. En el primer borrador de este libro, la figura 4-39 mostraba una caché asociativa de tres vías en lugar de una de cuatro vías. Uno de los revisores se molestó mucho, asegurando que ello crearía una confusión terrible entre los estudiantes porque tres no es una potencia de dos y las computadoras hacen todo en binario. Puesto que el cliente siempre tiene la razón, se cambió la figura a una caché asociativa de cuatro vías. ¿Tenía razón el revisor? Comente su respuesta.

29. Una computadora con una fila de procesamiento de cinco etapas maneja las ramificaciones condicionales parando durante los siguientes tres ciclos después de encontrarse con una. ¿Qué tanto perjudica esto al desempeño si el 20% de todas las instrucciones son ramificaciones condicionales? No tome en cuenta otras cosas que podrían parar la máquina, sólo las ramificaciones condicionales.

30. Suponga que una computadora prebusca hasta 20 instrucciones por adelantado. Sin embargo, cuatro de ellas en promedio son ramificaciones condicionales, cada una con una probabilidad del 90% de ser predicha correctamente. Calcule la probabilidad de que la prebúsqueda siga el camino correcto.

31. Suponga que se modifica el diseño de la máquina de la figura 4-43 de modo que tenga 16 registros en lugar de ocho. Luego modificamos I6 de modo que use R8 como destino. ¿Qué sucede en los ciclos a partir del sexto?

32. Normalmente, las dependencias causan problemas en las CPU con filas de procesamiento. ¿Existen optimaciones que puedan efectuarse con las dependencias WAW y que mejoren realmente la situación? ¿Cuáles son?

33. Reescriba el intérprete de Mic-1 de modo que ahora LV apunte a la primera variable local en lugar de al apuntador de enlace.

34. Escriba un simulador de una caché con mapeo directo de una vía. Haga que el número de entradas y el tamaño de línea sean parámetros de la simulación. Experimente con ella y prepare un informe de sus hallazgos.

5

EL NIVEL DE ARQUITECTURA DEL CONJUNTO DE INSTRUCCIONES

En este capítulo estudiaremos con detalle el nivel de arquitectura del conjunto de instrucciones (ISA, *Instruction Set Architecture*). Este nivel se ubica entre el nivel de microarquitectura y el del sistema operativo, como vimos en la figura 1-2. Históricamente, este nivel se desarrolló antes que cualquiera de los otros niveles y, de hecho, originalmente era el único nivel. Aun hoy no es raro oír hablar de este nivel simplemente como "la arquitectura" de una máquina o a veces (incorrectamente) como "lenguaje ensamblador".

El nivel ISA tiene una importancia especial que lo hace crucial para los arquitectos de sistemas: es la interfaz entre el software y el hardware. Si bien sería posible hacer que el hardware ejecute directamente programas escritos en C, C++, FORTRAN 90 o algún otro lenguaje de alto nivel, no sería una buena idea. Se perdería la ventaja en términos de desempeño de la compilación sobre la interpretación. Además, para ser útiles en la práctica, casi todas las computadoras deben ser capaces de ejecutar programas escritos en varios lenguajes, no sólo en uno.

La estrategia que adoptan casi todos los diseñadores de sistemas es hacer que programas en diversos lenguajes de alto nivel se traduzcan a una forma intermedia común –el nivel ISA– y construir hardware que pueda ejecutar directamente programas en el nivel ISA. El nivel ISA define la interfaz entre los compiladores y el hardware; es el lenguaje que ambos tienen que entender. La relación entre los compiladores, el nivel ISA y el hardware se muestra en la figura 5-1.

Idealmente, cuando los arquitectos diseñan una máquina nueva hablan tanto con los escritores de compiladores como con los ingenieros de hardware para averiguar qué caracte-

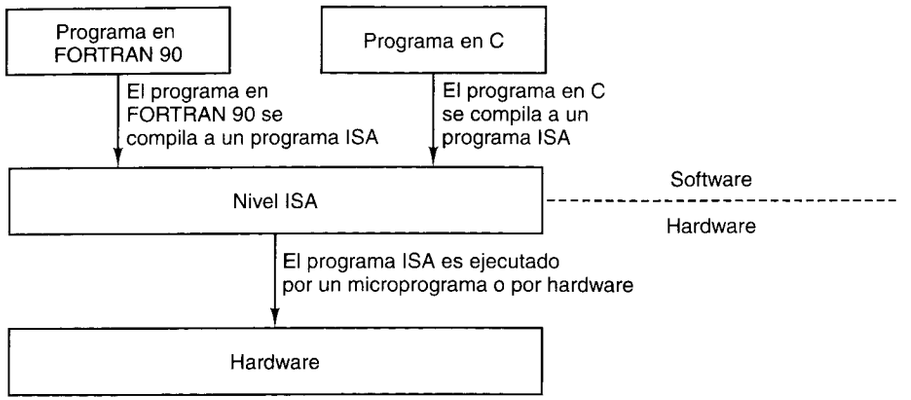


Figura 5-1. El nivel ISA es la interfaz entre los compiladores y el hardware.

rísticas quieren que se incluyan en el nivel ISA. Si los escritores de compiladores quieren alguna característica que los ingenieros no pueden implementar de forma económica (por ejemplo, una instrucción de ramificar y hacer la nómina), no se incluye. Asimismo, si la gente de hardware tiene una maravillosa característica nueva que quiere incluir (digamos una memoria en la que las palabras cuyas direcciones son números primos son superrápidas) pero la gente de software no puede pensar en un código que pueda aprovecharla, se quedará en los planos. Después de muchas negociaciones y simulaciones, surgirá un nivel ISA perfectamente optimado para los lenguajes de programación objetivo, y se implementará.

Eso es la teoría. Ahora viene la triste realidad. Cuando llega una nueva máquina, la primera pregunta que todos los clientes en potencia hacen es: "¿Es compatible con su predecesora?" La segunda es: "¿Puedo ejecutar mi sistema operativo antiguo en ella?" La tercera es: "¿Ejecutará todos mis programas de aplicación existentes sin modificación?" Si cualquiera de las respuestas es "no", los diseñadores tendrán mucho que explicar. A los clientes generalmente no les gusta tirar a la basura todo su software viejo y comenzar otra vez desde cero.

Esta actitud somete a los arquitectos de computadoras a una presión enorme en favor de mantener inalterada la ISA al cambiar de modelo, o al menos hacerla **compatible con modelos anteriores**. Con esto queremos decir que la nueva máquina debe ejecutar programas viejos sin modificación. Sin embargo, es perfectamente aceptable que la nueva máquina tenga nuevas instrucciones y otras características que aprovechar, además del software nuevo. En términos de la figura 5-1, en tanto los diseñadores hagan a la ISA compatible con los modelos anteriores, gozan de libertad más o menos completa para hacer lo que quieran con el hardware, pues a casi nadie le interesa el verdadero hardware (o siquiera saber qué hace). Los diseñadores pueden cambiar de un diseño microprogramado a uno de ejecución directa, o añadir filas de procesamiento o recursos superescalares o cualquier otra cosa que se les ocurra, siempre que mantengan la compatibilidad con la ISA anterior. La meta es asegurarse de que los programas viejos funcionen en la máquina nueva. El reto entonces es construir mejores máquinas sujetas a la restricción de compatibilidad hacia atrás.

Con lo anterior no queremos implicar que el diseño de ISA carezca de importancia. Una buena ISA es mucho mejor que una mala, sobre todo en términos de potencia de cómputo bruta *versus* costo. Si dos diseños son equivalentes en todo lo demás, diferentes ISA podrían representar una diferencia de hasta un 25% en el desempeño. Lo que queremos destacar es que las fuerzas del mercado hacen que sea muy difícil (aunque no imposible) desechar una ISA antigua e introducir una nueva. No obstante, de vez en cuando surge una nueva ISA de aplicación general, y en mercados especializados (como sistemas incorporados o procesadores multimedia) se dan con mucha mayor frecuencia. Por tanto, entender el diseño de la ISA es importante.

¿Qué hace que una ISA sea buena? Hay dos factores primordiales. Primero, una buena ISA debe definir un conjunto de instrucciones que se pueda implementar con eficiencia en las tecnologías actuales y futuras y permita crear diseños económicos durante varias generaciones. Un diseño deficiente es más difícil de implementar; podría requerir muchas más compuertas para implementar el procesador y más memoria para ejecutar los programas. También podría ser más lento porque la ISA no permite distinguir oportunidades para traslapar operaciones, y requiere diseños mucho más complicados para alcanzar un desempeño equivalente. Un diseño que aprovecha las peculiaridades de una tecnología dada podría ser algo efímero, que ofrece una sola generación de implementaciones económicas y luego es superado por unas ISA más progresivas.

Segundo, una buena ISA debe ofrecer un objetivo claro para el código compilado. La regularidad y exhaustividad de una gama de opciones son rasgos importantes que no siempre están presentes en una ISA. Éstas son propiedades cruciales para un compilador, el cual podría tener problemas para escoger la mejor opción entre alternativas limitadas, sobre todo cuando la ISA no permite algunas alternativas aparentemente obvias. En pocas palabras, dado que la ISA es la interfaz entre el hardware y el software, debe complacer a los diseñadores de hardware (debe ser fácil de implementar con eficiencia) y a los de software (debe ser fácil generar buen código para ella).

5.1 GENERALIDADES DEL NIVEL ISA

Iniciemos nuestro estudio del nivel ISA preguntándonos qué es. Ésta podría parecer una pregunta sencilla, pero tiene más complicaciones de las que podríamos imaginar. En la sección que sigue presentaremos algunos de estos aspectos; luego examinaremos modelos de memoria, registros e instrucciones.

5.1.1 Propiedades del nivel ISA

En principio, el nivel ISA está definido por el aspecto que la máquina presenta al programador de lenguaje de máquina. Puesto que ya ninguna persona (en sus cabales) programa mucho en lenguaje de máquina, modifiquemos la definición para decir que el código en el nivel ISA es lo que un compilador produce (haciendo caso omiso por el momento de las llamadas al

sistema operativo y del lenguaje ensamblador simbólico). Para producir código en el nivel ISA, el escritor de compiladores tiene que conocer el modelo de memoria, los registros disponibles, los tipos de datos e instrucciones con que se cuenta, etc. El conjunto de toda esta información es lo que define el nivel ISA.

Según esta definición, cuestiones como si la microarquitectura está microprogramada o no, si usa filas de procesamiento o no, si es superescalar o no, etc., no forman parte del nivel ISA porque el escritor de compiladores no las percibe. Sin embargo, esta afirmación no es del todo correcta porque algunas de esas propiedades afectan el desempeño, y el escritor de compiladores sí percibe eso. Consideremos, por ejemplo, un diseño superescalar que puede emitir instrucciones una tras otra en el mismo ciclo siempre que una manipule enteros y la otra sea de punto flotante. Si el compilador alterna instrucciones de enteros con instrucciones de punto flotante, logrará un desempeño claramente mejor que si no lo hace. Por tanto, los detalles del funcionamiento superescalar *sí son* visibles en el nivel ISA, y la separación entre las capas no es tan nítida como podría parecer a primera vista.

En algunas arquitecturas el nivel ISA se especifica con un documento de definición formal, a menudo producido por un consorcio de la industria. En otras no hay tal documento. Por ejemplo, tanto SPARC V9 (la versión 9 de SPARC) como la JVM tienen definiciones oficiales (Weaver y Germond, 1994; y Lindholm y Yellin, 1997, respectivamente). El propósito de un documento reglamentado es que diferentes implementadores puedan construir la máquina y que todas ellas ejecuten exactamente el mismo software y obtengan exactamente los mismos resultados.

En el caso de SPARC, lo que se busca es que diversos fabricantes de chips produzcan chips SPARC que sean funcionalmente idénticos, con diferencias sólo en el desempeño y el precio. Para que esta idea funcione, los fabricantes de chips tienen que saber qué se supone que hace un chip SPARC (en el nivel ISA). Por tanto, el documento reglamentado especifica el modelo de memoria, los registros presentes, lo que hacen las instrucciones, etc., pero no la microarquitectura.

Tales documentos reglamentados contienen secciones **normativas**, que imponen requisitos, y secciones **informativas**, que ayudan al lector pero no forman parte de la definición formal. Las secciones normativas usan mucho palabras como *debe*, *no puede* y *debería*, para requerir, prohibir y sugerir aspectos de la arquitectura, respectivamente. Por ejemplo, un enunciado como

La ejecución de un código de operación reservado debe causar una trampa.

dice que si un programa ejecuta un código de operación que no está definido, debe causar una trampa (una interrupción en software) y no hacer caso omiso de él. Una estrategia alternativa podría ser dejar este aspecto abierto, en cuyo caso el enunciado podría ser

El efecto de ejecutar un código de operación reservado está definido por la implementación.

Esto significa que el escritor de compiladores no puede confiar en un comportamiento específico, pues los diferentes implementadores están en libertad de tomar diferentes decisiones. Casi todas las especificaciones de arquitectura van acompañadas por juegos de programas de prueba que verifican si una implementación que asegura cumplir con la especificación realmente lo hace.

Es evidente por qué el SPARC V9 tiene un documento que define su nivel ISA: para que todos los chips SPARC V9 ejecuten el mismo software. Por lo mismo, existe un documento reglamentado para JVM: a fin de que todos los intérpretes (o chips como el picoJava II) puedan ejecutar cualquier programa JVM válido. No existe un documento de definición formal para el nivel ISA del Pentium II porque Intel no quiere que para otros fabricantes sea fácil producir chips Pentium II. De hecho, Intel ha acudido a las cortes para tratar de impedir que otros vendedores produzcan clones de sus chips.

Otra propiedad importante del nivel ISA es que en casi todas las máquinas hay al menos dos modos. El **modo de Kernel** sirve para ejecutar el sistema operativo y permite la ejecución de todas las instrucciones. El **modo de usuario** sirve para ejecutar programas de aplicación y no permite que se ejecuten ciertas instrucciones peligrosas (como las que manipulan directamente la caché). En este capítulo nos concentraremos primordialmente en las instrucciones y propiedades del modo de usuario.

5.1.2 Modelos de memoria

Todas las computadoras dividen la memoria en celdas que tienen direcciones consecutivas. El tamaño de celda más común actualmente es de 8 bits, pero se han usado celdas de 1 bit hasta 60 bits (véase la figura 2-10). Una celda de 8 bits se llama **byte**. La razón para usar bytes es que los caracteres ASCII tienen 7 bits, de modo que un carácter ASCII y un bit de paridad caben en un byte. Si UNICODE llega a dominar la industria en el futuro, las computadoras podrían basarse en unidades de 16 bits numeradas consecutivamente. Después de todo, 2^4 es un número más bonito que 2^3, ya que 4 es una potencia de 2, y 3 no lo es.

Los bytes generalmente se agrupan en palabras de 4 bytes (32 bits) u 8 bytes (64 bits), y se proporcionan instrucciones para manipular palabras enteras. Muchas arquitecturas exigen que las palabras estén alineadas respecto a sus fronteras naturales, de modo que, por ejemplo, una palabra de 4 bytes puede comenzar en la dirección, 0, 4, 8, etc., pero no en la dirección 1 ni en la 2. Asimismo, una palabra de 8 bytes puede comenzar en la dirección 0, 8, 16, etc., pero no en la dirección 4 ni en la 6. La alineación de palabras de 8 bytes se ilustra en la figura 5-2.

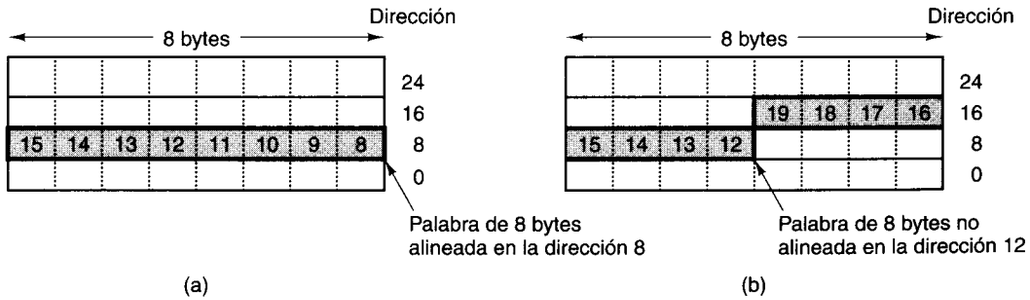


Figura 5-2. Palabra de 8 bytes en una memoria little-endian. (a) Alineada. (b) No alineada. Algunas máquinas requieren que las palabras estén alineadas en la memoria.

Normalmente se exige la alineación porque las palabras operan con mayor eficiencia así. El Pentium II, por ejemplo, que obtiene grupos de 8 bytes de la memoria, usa direcciones físicas de 36 bits, pero tiene sólo 33 bits de dirección, como se muestra en la figura 3-44. Así, el Pentium II no podría hacer una referencia no alineada a la memoria aunque quisiera, porque los tres bits de orden bajo no se especifican explícitamente; siempre son ceros, lo que obliga a todas las direcciones de memoria a ser múltiplos de 8 bytes.

Sin embargo, este requisito de alineación a veces causa problemas. En el Pentium II, los programas ISA pueden hacer referencia a palabras que comienzan en cualquier dirección, propiedad que se remonta al 8088, que tenía un bus de datos de un byte de anchura (y por ende ningún requisito de alineación de las referencias a la memoria según fronteras de 8 bytes). Si un programa del Pentium II lee una palabra de 4 bytes en la dirección 7, el hardware tiene que hacer una referencia a la memoria para obtener los bytes del 0 al 7 y una segunda referencia a la memoria para obtener los bytes del 8 al 15. Luego la CPU tiene que extraer los 4 bytes requeridos de los 16 bytes que se leyeron de la memoria y armarlos en el orden correcto para formar una palabra de 4 bytes.

La capacidad para leer palabras en direcciones arbitrarias requiere lógica adicional en el chip, lo que lo hace más grande y más costoso. A los ingenieros de diseño les encantaría deshacerse de esa capacidad y simplemente exigir a todos los programas que hagan referencias a la memoria alineadas por palabra. El problema es que cada vez que los ingenieros dicen "¿A quién le interesa ejecutar programas apolillados para el 8088 que no hacen referencia a la memoria como debe ser?" la gente de comercialización tiene una respuesta adecuada: "A nuestros clientes".

Casi todas las máquinas tienen un solo espacio de direcciones lineal en el nivel ISA, que se extiende desde la dirección 0 hasta algún máximo, que suele ser 2^{32} bytes o 2^{64} bytes. Sin embargo, unas cuantas máquinas tienen espacios de direcciones distintos para las instrucciones y para los datos, de modo que una obtención de instrucción en la dirección 8 acude a un espacio de direcciones distinto que una obtención de datos en la dirección 8. Este esquema es más complejo que tener un solo espacio de direcciones, pero tiene dos ventajas. Primera, es posible tener 2^{32} bytes de programa y otros 2^{32} bytes de datos usando sólo direcciones de 32 bits. Segunda, como todas las escrituras se efectúan automáticamente en el espacio de datos, es imposible sobreescribir accidentalmente el programa, con lo que se elimina una fuente de errores de programa.

Cabe señalar que tener espacios de direcciones distintos para instrucciones y datos no es lo mismo que tener una caché de nivel 1 dividida. En el primer caso, la cantidad total de espacio de direcciones se duplica y las lecturas a una dirección dada producen resultados diferentes, dependiendo de si se está leyendo una palabra de instrucciones o de datos. Con una caché dividida, sigue habiendo un solo espacio de direcciones, sólo de diferentes cachés guardan diferentes partes de ese espacio.

Otro aspecto del modelo de memoria en el nivel ISA es la semántica de memoria. Es natural esperar que una instrucción LOAD que ocurre después de una instrucción STORE y que hace referencia a la misma dirección devuelva el valor que se acaba de guardar. Sin embargo, como vimos en el capítulo 4, en muchos diseños las microinstrucciones se reordenan, y existe el peligro de que la memoria no tenga el comportamiento esperado. El problema empeora en un multiprocesador, donde cada una de las CPU envía un flujo de solicitudes de lectura y escritura (posiblemente reordenadas) a una memoria compartida.

Los diseñadores de sistemas pueden adoptar varias estrategias para resolver este problema. En un extremo, todas las solicitudes de memoria se pueden colocar en serie, de modo que una se finalice antes de que se emita la siguiente. Esta estrategia perjudica el desempeño pero simplifica al máximo la semántica de memoria (todas las operaciones se ejecutan en el orden estricto del programa).

En el otro extremo, no se ofrecen en general garantías de ningún tipo. Para forzar un ordenamiento de la memoria, el programa debe ejecutar una instrucción SYNC, que bloquea la emisión de operaciones de memoria nuevas hasta que todas las anteriores hayan terminado. Este diseño somete a los compiladores a un gran esfuerzo porque tienen que entender los detalles del funcionamiento de la microarquitectura subyacente, pero ofrece a los diseñadores de hardware el máximo de libertad para optimar el uso de la memoria.

También puede haber modelos de memoria intermedios, en los que el hardware bloquea automáticamente la emisión de ciertas referencias a la memoria (por ejemplo, las que implican una dependencia RAW o WAR) pero no otras. Si bien es molesto que todas estas peculiaridades causadas por la microarquitectura queden expuestas en el nivel ISA (al menos para los escritores de compiladores y los programadores en lenguaje ensamblador), es la tendencia actual. Dicha tendencia se debe a las implementaciones subyacentes, como el reordenamiento de microinstrucciones, las filas de procesamiento de gran capacidad, las cachés con múltiples niveles, y demás. Veremos más ejemplos de tales efectos artificiales más adelante en este capítulo.

5.1.3 Registros

Todas las computadoras tienen algunos registros que son visibles en el nivel ISA. Su función es controlar la ejecución del programa, almacenar resultados temporales y otras cosas. En general, los registros visibles en el nivel de microarquitectura, como TOS y MAR en la figura 4-1, no son visibles en el nivel ISA. Sin embargo, unos cuantos de ellos, como el contador de programa y el apuntador de la pila, son visibles en ambos niveles. Por otra parte, los registros visibles en el nivel ISA siempre son visibles en el nivel de microarquitectura porque ahí es donde se implementan.

Los registros del nivel ISA se pueden dividir en dos categorías amplias: registros de propósito especial y registros de propósito general. Los primeros incluyen cosas como el contador de programa y el apuntador a la pila, además de otros registros con funciones específicas. En contraste, los registros de propósito general sirven para guardar variables locales clave y los resultados intermedios de los cálculos. Su función principal es proporcionar acceso rápido a datos que se usan continuamente (básicamente para evitar accesos a la memoria). Las máquinas RISC, con sus CPU rápidas y memorias (relativamente) lentas, casi siempre tienen por lo menos 32 registros de propósito general, y la tendencia en los nuevos diseños de CPU es a tener aún más.

En algunas máquinas los registros de propósito general son totalmente simétricos e intercambiables. Si todos los registros son equivalentes, un compilador puede usar R1 para contener un resultado temporal, pero igual podría usar R25. El registro escogido no importa.

En cambio, en otras máquinas algunos de los registros de propósito general podrían tener algo de especial. Por ejemplo, en el Pentium II hay un registro llamado EDX que se

puede usar como registro general, pero que también recibe la mitad del producto en una multiplicación y que contiene la mitad del dividendo en una división.

Aun si los registros de propósito general son totalmente intercambiables, es común que el sistema operativo o los compiladores adopten convenciones sobre su uso. Por ejemplo, algunos registros podrían contener parámetros de procedimientos invocados y otros podrían usarse como registros de borrador. Si un compilador coloca una variable local importante en R1 y luego invoca a un procedimiento de biblioteca que cree que R1 es un registro de borrador que puede usar, cuando el procedimiento regrese, R1 podría contener basura. Si existen convenciones en el nivel de sistema sobre el uso de registros, es recomendable que los compiladores y los programadores en lenguaje ensamblador se ajusten a ellas.

Además de los registros en el nivel ISA que los programas de usuario pueden ver, siempre hay una cantidad sustancial de registros de propósito especial que sólo son accesibles en modo de kernel. Estos registros controlan las diversas cachés, memoria, dispositivos de E/S y otras características de hardware de la máquina. Sólo el sistema operativo los usa, así que los compiladores y los usuarios no tienen que saber de ellos.

Un registro de control que es una especie de híbrido kernel/usuario es el **registro de banderas** o **palabra de estado del programa** (**PSW**, *Program Status Word*). Este registro contiene diversos bits que la CPU necesita. Los bits más importantes son los **códigos de condición**. Estos bits se ajustan en cada ciclo de la ALU y reflejan la situación del resultado de la operación más reciente. Entre los bits de condición más comunes están

N – Se enciende si el resultado fue negativo

Z – Se enciende si el resultado fue cero

V – Se enciende si el resultado causó un desbordamiento

C – Se enciende si el resultado causó un acarreo de salida del bit de la extrema izquierda

A – Se enciende si hubo un acarreo de salida del bit 3 (acarreo auxiliar)

P – Se enciende si el resultado tuvo paridad par

Los códigos de condición son importantes porque las instrucciones de comparación y ramificación condicional (es decir, las instrucciones de salto condicional) los usan. Por ejemplo, la instrucción CMP generalmente resta dos operandos y establece los códigos de condición con base en la diferencia. Si los operandos son iguales, la diferencia será cero y se encenderá el bit de código de condición *Z* de la PSW. Una instrucción BEQ (ramificar si son iguales, *Branch Equal*) subsecuente prueba el bit *Z* y toma la rama si está encendido.

La PSW contiene más que sólo los códigos de condición, pero el contenido total varía de una máquina a otra. Otros campos comunes indican el modo de la máquina (es decir, de kernel o de usuario), un bit de rastreo (que sirve para depurar), el nivel de prioridad de la CPU y la habilitación de interrupciones. En muchos casos es posible leer la PSW en modo de usuario, pero algunos de los campos sólo pueden modificarse en modo de kernel (por ejemplo, el bit de modo de usuario/kernel).

5.1.4 Instrucciones

La característica principal del nivel de ISA es su conjunto de instrucciones de máquina. Éstas controlan lo que la máquina puede hacer. Siempre hay instrucciones LOAD y STORE (en una forma o en otra) para trasladar datos entre la memoria y los registros, e instrucciones MOVE para copiar datos entre los registros. Siempre se cuenta con instrucciones aritméticas, lo mismo que instrucciones booleanas e instrucciones para comparar datos y tomar una rama u otra según los resultados. Ya hemos visto algunas instrucciones ISA representativas (vea la figura 4-11) y estudiaremos muchas más en este capítulo.

5.1.5 Generalidades del nivel ISA del Pentium II

En este capítulo veremos tres ISA muy diferentes: la IA-32 de Intel, encarnada en el Pentium II, la arquitectura SPARC versión 9, implementada en los procesadores UltraSPARC, y JVM, implementada en el picoJava II. La intención no es ofrecer una descripción exhaustiva de cualquiera de las ISA, sino mostrar los aspectos importantes de una ISA y cómo dichos aspectos pueden variar de una ISA a otra. Comencemos con el Pentium II.

El procesador Pentium II ha evolucionado a través de muchas generaciones, y su linaje se remonta a algunos de los primeros microprocesadores jamás construidos, como vimos en el capítulo 1. Si bien la ISA básica mantiene apoyo completo para la ejecución de programas escritos para los procesadores 8086 y 8088 (que tenían la misma ISA), contiene restos del 8080, un procesador de 8 bits popular en la década de los setenta. El 8080, a su vez, tuvo una marcada influencia de restricciones de compatibilidad con el 8008, más antiguo aún, que se basó en el 4004, un chip de 4 bits que se usaba cuando los dinosaurios dominaban el planeta.

Desde el punto de vista del software, el 8086 y el 8088 eran máquinas sencillas de 16 bits (aunque el 8088 tenía un bus de datos de 8 bits). Su sucesor, el 80286, también fue una máquina de 16 bits. Su ventaja principal fue un mayor espacio de direcciones, aunque pocos programas lo llegaron a usar porque consistía en 16,384 segmentos de 64K en lugar de una memoria lineal de 2^{24} bytes.

El 80386 fue la primera máquina de 32 bits de la familia Intel. Todas las máquinas subsecuentes (80486, Pentium, Pentium Pro, Pentium II, Celeron y Xeon) han tenido prácticamente la misma arquitectura de 32 bits que el 80386, llamada **IA-32**, por lo que es ésta arquitectura en la que nos concentraremos aquí. El único cambio arquitectónico importante desde el 80386 fue la introducción de las instrucciones MMX en versiones posteriores del Pentium, y su subsecuente inclusión en el Pentium II y otros procesadores.

El Pentium II tiene tres modos de operación, dos de los cuales hacen que actúe como un 8088. En **modo real**, todas las funciones que se han añadido desde el 8088 se desactivan y el Pentium II se comporta como un simple 8088. Si cualquier programa hace algo indebido, toda la máquina se cae. Si Intel hubiera diseñado a los seres humanos, habría incluido un bit que los hiciera revertir a modo de chimpancé (casi todo el cerebro inhabilitado, no habla, come casi únicamente plátanos, etc.).

Un paso más arriba está el **modo 8086 virtual**, que permite ejecutar programas para 8088 viejos con protección. En este modo, un verdadero sistema operativo controla toda la

máquina. Para ejecutar un programa para 8088, el sistema operativo crea un entorno aislado especial que actúa como un 8088, excepto que si su programa se cae se notifica al sistema operativo en lugar de hacer que la máquina se caiga. Cuando un usuario de Windows inicia una ventana MS-DOS, el programa que se ejecuta ahí se inicia en modo 8086 virtual para proteger a Windows de programas MS-DOS mal comportados.

El último modo es el modo protegido, en el que el Pentium II realmente actúa como Pentium II y no como un 8088 muy costoso. Hay cuatro niveles de privilegio controlados por bits de la PSW. El nivel 0 corresponde al modo de kernel en otras computadoras y tiene pleno acceso a la máquina. El sistema operativo usa este nivel. El nivel 3 es para programas de usuario; bloquea el acceso a ciertas instrucciones y registros de control críticos para evitar que un programa de usuario desbocado haga caer a toda la máquina. Los niveles 1 y 2 casi nunca se usan.

El Pentium II tiene un espacio de direcciones enorme, con la memoria dividida en 16,384 segmentos, cada uno de los cuales va desde la dirección 0 hasta la dirección $2^{32} - 1$. Sin embargo, casi todos los sistemas operativos (incluidos UNIX y todas las versiones de Windows) sólo reconocen un segmento, así que la generalidad de los programas de aplicación ven un espacio de direcciones lineal de 2^{32} bytes, y a veces una parte de éste está ocupado por el sistema operativo. Cada byte del espacio de direcciones tiene su propia dirección, y las palabras son de 32 bits. El formato para guardar las palabras es little endian (el byte de orden más bajo tiene la dirección más baja).

Los registros del Pentium II se muestran en la figura 5-3. Los primeros cuatro registros, EAX, EBX, ECX y EDX son registros de 32 bits de propósito más o menos general, aunque cada uno tiene sus propias peculiaridades. EAX es el principal registro aritmético; EBX es bueno para contener apuntadores (direcciones de memoria); ECX desempeña un papel en los ciclos; EDX se necesita para la multiplicación y la división donde, junto con EAX, guarda productos y dividendos de 64 bits. Cada uno de estos registros contiene un registro de 16 bits en los 16 bits de orden bajo y un registro de 8 bits en los 8 bits de orden bajo. Estos registros facilitan la manipulación de cantidades de 16 y 8 bits, respectivamente. El 8088 y el 80286 sólo tenían los registros de 8 y 16 bits. Los registros de 32 bits se añadieron en el 80386, junto con el prefijo E, que significa Extendido.

Los siguientes tres registros también son de propósito más o menos general, pero con más peculiaridades. Los registros ESI y EDI se incluyeron para contener apuntadores a la memoria, sobre todo para las instrucciones de manipulación de cadenas en hardware, en las que ESI apunta a la cadena de origen y EDI apunta a la cadena de destino. El registro EBP también es para apuntadores; generalmente se usa para apuntar a la base del marco de pila vigente, como hace LV en IJVM. Cuando un registro (como EBP) sirve para apuntar a la base del marco de pila local, suele llamársele **apuntador de cuadro**. Por último, ESP es el apuntador de la pila.

El siguiente grupo de registros, CS a GS, son registros de segmento. En gran medida, se trata de trilobites electrónicos, fósiles que tienen que ver con la forma en que el 8088 trataba de direccionar 2^{20} bytes de memoria usando direcciones de 16 bits. Baste decir que cuando el Pentium II se configura para usar un solo espacio de direcciones lineal de 32 bits, podemos hacer caso omiso de ellos sin peligro. Luego viene el EIP, que es el contador de programa

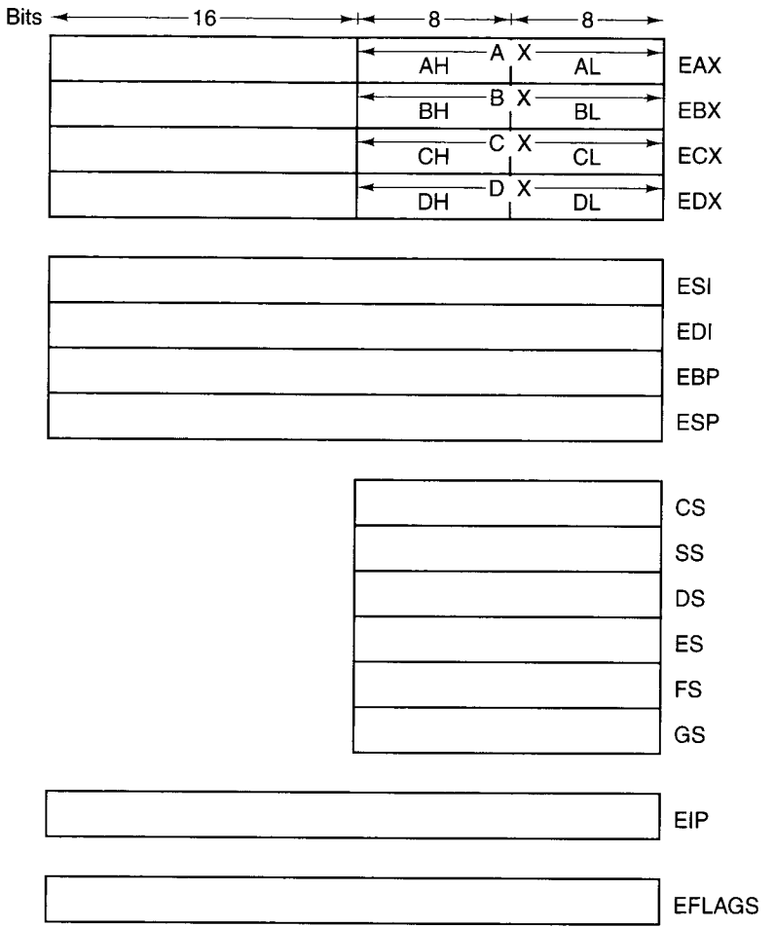


Figura 5-3. Los registros primarios del Pentium II.

(*Extended Instruction Pointer*, apuntador a instrucciones extendido). Por último, llegamos a EFLAGS, que es la PSW.

5.1.6 Generalidades del nivel ISA del UltraSPARC II

Sun Microsystems introdujo la arquitectura SPARC en 1987. SPARC fue una de las primeras arquitecturas comerciales calificadas como RISC, y se basó en gran medida en las investigaciones realizadas en Berkeley a fines de la década de los ochenta (Patterson, 1985; Patterson y Séquin, 1982). El SPARC original era una arquitectura de 32 bits, pero el UltraSPARC II es una máquina de 64 bits basada en la versión 9 de la arquitectura, y es la que describiremos en este capítulo. Por congruencia con el resto del libro, nos referiremos al UltraSPARC II aquí, pero en el nivel ISA todos los UltraSPARC son idénticos.

La estructura de memoria del UltraSPARC II es sencilla y clara: la memoria direccionable es una formación lineal de 2^{64} bytes. Lo malo es que esta memoria es tan grande (18,446,744,073,709,551,616 bytes) que ninguna máquina actual puede implementarla. Las implementaciones actuales limitan el tamaño de las direcciones a las que pueden accesar (2^{44} bytes en el UltraSPARC II), pero esto aumentará en modelos futuros. El orden por omisión de los bytes es big endian, pero puede convertirse en little endian encendiendo un bit en la PSW.

Es importante que la ISA tenga un límite más alto del que necesitan las implementaciones, porque las implementaciones futuras con toda seguridad necesitarán aumentar el tamaño de la memoria a la que el procesador puede accesar. Uno de los problemas más graves que surgen en las arquitecturas que tienen éxito ha sido que su ISA limita la cantidad de memoria direccionable. En computación, el único error que no se puede subsanar es la falta de suficientes bits. Un día sus nietos le preguntarán cómo las computadoras lograban hacer algo con direcciones de sólo 32 bits y sólo 4 GB de memoria real.

El ISA de SPARC es claro, aunque la organización de los registros es un tanto compleja en un intento por hacer a las llamadas a procedimientos más eficientes. La experiencia muestra que la organización de los registros no vale el trabajo que se invierte en ella, pero la regla de la compatibilidad con modelos anteriores hizo imposible deshacerse de ella.

El UltraSPARC II tiene dos grupos de registros: 32 registros de propósito general de 64 bits y 32 registros de punto flotante. Los registros de propósito general se llaman R0 a R31 aunque en ciertos contextos se usan otros nombres. Los nombres alternativos y las funciones de los registros se muestran en la figura 5-4.

Registro	Nombre alt.	Función
R0	G0	Conectado a 0. Escribir en él no hace nada
R1 – R7	G1 – G7	Contiene variables globales
R8 – R13	O0 – O5	Contiene parámetros al procedimiento invocado
R14	SP	Apuntador a la pila
R15	O7	Registro de borrador
R16 – R23	L0 – L7	Contiene variables locales para el procedimiento actual
R24 – R29	I0 – I5	Contiene parámetros entrantes
R30	FP	Apuntador a la base del marco de pila vigente
R31	I7	Contiene dirección de retorno del procedimiento actual

Figura 5-4. Los registros generales del UltraSPARC II.

Todos los registros de propósito general tienen una capacidad de 64 bits y, con excepción de R0 que siempre es 0, pueden ser leídos o escritos por diversas instrucciones de carga y

almacenamiento. Los usos que se dan en la figura 5-4 se basan en parte en la convención, pero también en cómo el hardware los trata. En general, no es prudente darles un uso distinto del que se indica en la figura 5-4 a menos que usted sea un Gurú SPARC Cinta Negra y de veras sepa perfectamente lo que está haciendo. Es responsabilidad del compilador o del programador asegurarse de que el programa accese a los registros correctamente y realice los tipos correctos de operaciones aritméticas con ellos. Por ejemplo, es muy fácil cargar números de punto flotante en los registros generales y luego realizar suma de enteros con ellos, operación que producirá resultados sin sentido pero que la CPU realizará de buena gana si se le pide.

Las variables globales sirven para retener constantes, variables y apuntadores que se necesitan en todos los procedimientos, aunque pueden guardarse y volverse a cargar al salir de un procedimiento y entrar en él si es necesario. Los registros Ix y Ox se usan para pasar parámetros a procedimientos y así evitar referencias a la memoria. Más adelante explicaremos cómo funciona esto.

Tres registros dedicados se usan para cosas especiales. Los registros FP y SP delimitan el cuadro vigente. El primero apunta a la base del cuadro vigente y sirve para direccionar variables locales, exactamente igual como hacía LV en la figura 4-10. El segundo indica el tope de pila actual y fluctúa a medida que se meten palabras en la pila y se sacan de ella. En cambio, FP sólo cambia cuando se invoca un procedimiento o cuando se regresa de un procedimiento. El tercer registro de propósito especial es R31; se usa en llamadas a procedimientos para guardar la dirección de retorno.

El UltraSPARC II en realidad tiene más de 32 registros de propósito general, aunque sólo 32 son visibles para el programa en un momento dado. Esta característica llamada **ventana de registros**, se incluyó para apoyar de forma eficiente las llamadas a procedimientos, y se ilustra en la figura 5-5. La idea básica es emular una pila, pero usando registros. Es decir, en realidad hay varios conjuntos de registros, así como hay varios cuadro en una pila. Exactamente 32 registros generales son visibles en un instante dado. El registro CWP (apuntador a la ventana vigente, *Current Window Pointer*) indica cuál conjunto de registros se está usando actualmente.

La instrucción de llamada a procedimiento oculta el conjunto anterior de registros y proporciona un conjunto nuevo para que lo use el procedimiento invocado decrementando CWP. Sin embargo, algunos registros se perpetúan del procedimiento invocador al procedimiento invocado, lo que ofrece un mecanismo eficiente para pasar parámetros entre procedimientos. Esta técnica opera cambiando el nombre de algunos de los registros: después de la llamada al procedimiento, los antiguos registros de salida, R8 a R15, siguen siendo visibles, pero ahora son los registros de entrada, R24 a R31. Sin embargo, los ocho registros globales no cambian, es decir, siempre son el mismo conjunto de registros.

A diferencia de la memoria, que es casi infinita (al menos en lo que toca a la pila), cuando los procedimientos se anidan con demasiada profundidad la máquina se queda sin ventanas de registros desocupadas. En ese punto el conjunto de mayor antigüedad se vacía a la memoria para desocupar un conjunto. Asimismo, después de muchos retornos de procedimientos, podría ser necesario traer un conjunto de registros de la memoria. En general, esta complejidad es una lata y probablemente no vale la pena. Sólo ayuda cuando las llamadas no se anidan con demasiada profundidad.

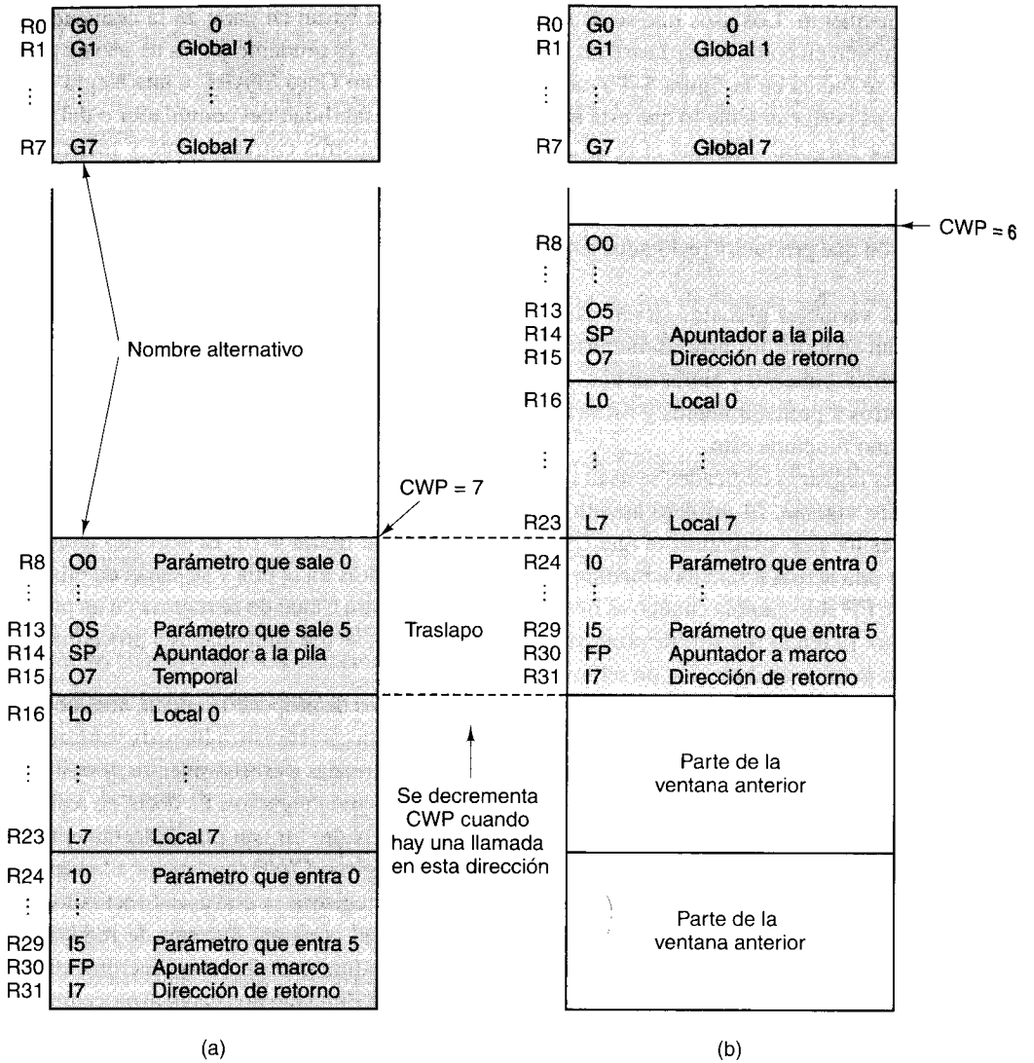


Figura 5-5. Operación de las ventanas de registros del UltraSPARC II.

El UltraSPARC II también tiene 32 registros de punto flotante, que pueden contener valores de 32 bits (precisión sencilla) o bien de 64 bits (doble precisión). También es posible usar pares de estos registros para manejar valores de 128 bits (cuádruple precisión).

La arquitectura del UltraSPARC II es una **arquitectura de carga/almacenamiento**. Es decir, las únicas operaciones que accesan a la memoria directamente son los LOAD y STORE, instrucciones para trasladar datos entre los registros y la memoria. Todos los operandos para instrucciones aritméticas y lógicas deben provenir de registros o ser proporcionados por la instrucción (no la memoria), y todos los resultados se deben guardar en un registro (no en la memoria).

5.1.7 Generalidades del nivel ISA de la Máquina Virtual Java (JVM)

El nivel ISA de la JVM es poco común, pero limpio y sencillo. Ya lo vimos en su mayor parte cuando estudiamos la JVM. El modelo de memoria JVM es igual al modelo IJVM que presentamos en el capítulo 4 e ilustramos en la figura 4.10, pero con una región adicional, que presentaremos en breve. El orden de los bytes es big endian.

La memoria tiene cuatro regiones principales: el marco de variables locales, la pila de operandos, el área de métodos y la reserva de constantes. Recuerde que las implementaciones Mic-*x* de IJVM tienen los registros LV, SP, PC y CPP que apuntan a estas regiones. Todos los accesos a la memoria deben efectuarse como una distancia respecto a uno de estos registros; nunca se usan apuntadores ni direcciones de memoria absolutas. Aunque JVM no requiere estos registros (ni los menciona), es probable que la mayor parte de las implementaciones los tengan o tengan algo muy parecido.

La ausencia de apuntadores para accesar a variables locales y constantes no es un accidente; es crucial para alcanzar una de las principales metas de diseño de Java: que los usuarios puedan bajar un programa JVM (binario) de Internet y ejecutarlo de forma segura, sin que el programa pueda espiar o dañar la máquina en la que se está ejecutando. Al limitar el uso de apuntadores, puede aumentarse la seguridad.

Recuerde que ninguna de las regiones de memoria definidas en IJVM podía ser muy grande. El área de métodos sólo podía tener 64 KB. Asimismo, el espacio ocupado por las variables locales no podía ser mayor que 64 KB. También la reserva de constantes está limitada a 64 KB. JVM tiene exactamente las mismas restricciones y por la misma razón exactamente: las distancias que se usan como índices para direccionar estas regiones están limitadas a 16 bits.

Desde luego, el área del marco de variables locales es distinta para cada procedimiento, de modo que cada procedimiento recién invocado tiene 64 KB para sus propias variables locales. Asimismo, la reserva de constantes es distinta para cada clase, de modo que cada una puede tener sus propios 64 KB. Aun así, no hay lugar para guardar arreglos grandes o estructuras de datos que se crean dinámicamente como las listas o los árboles. Para resolver este problema, JVM contiene una región extra de memoria, el **montículo** (*heap*), que se puede asignar para guardar objetos dinámicos o muy grandes. Cuando el compilador de Java ve un enunciado como

```
int a[ ] = new int[4096]
```

genera una llamada al asignador de memoria de tiempo de ejecución, que asigna el espacio en el montículo y devuelve un apuntador a él. Así pues, sí se usan apuntadores en JVM, pero de una forma muy controlada para que los programadores no puedan manipularlos directamente.

Si se crean suficientes estructuras de datos nuevas en el montículo, llegará un momento en el que éste se quedará sin espacio. Cuando el sistema de tiempo de ejecución de Java detecta que el montículo está (casi) lleno, invoca un programa llamado **recolector de basura** que busca en el montículo objetos que ya no se están usando. Se emplea un algoritmo sofisticado (generacional). Luego, los objetos no utilizados se desechan para desocupar espacio en el montículo. La asignación con new y la liberación con el recolector de basura automático constituyen un esquema de gestión de memoria totalmente diferente del uso de una pila para variables locales. Las dos técnicas son complementarias; cada una tiene su sitio.

JVM no tiene registros de propósito general que puedan cargarse o almacenarse bajo control del programa; es una máquina de pila pura. Si bien ésta es una decisión poco común en estos días, el resultado es una ISA sencilla y elegante, para la cual es fácil compilar.

El problema principal de las máquinas de pila es el gran número de referencias a la memoria que se requieren. Sin embargo, como vimos en la sección 4.6.3, un diseño ingenioso puede eliminar una gran cantidad de ellas mediante el plegado de varias instrucciones JVM. Además, la sencillez de la arquitectura implica que se puede implementar en una cantidad de silicio pequeña, lo que deja la mayor parte del área del chip disponible para una enorme caché de nivel 1, lo que reduce aún más el número de referencias a la memoria. (El picoJava II tiene un caché de cuando más 16 KB + 16 KB porque la meta era construir un chip de muy bajo costo más que uno muy veloz.)

5.2 TIPOS DE DATOS

Todas las computadoras necesitan datos. De hecho, la tarea principal de muchos sistemas de computadora es procesar datos financieros, comerciales, científicos, de ingeniería o de otro tipo. Los datos tienen que representarse de alguna forma específica dentro de la computadora. En el nivel ISA se emplean diversos tipos de datos, que explicaremos a continuación.

Una cuestión fundamental es si existe o no apoyo de hardware para un tipo de datos en particular. Apoyo de hardware significa que una o más instrucciones esperan datos en un formato dado, y el usuario no puede escoger un formato distinto. Por ejemplo, los contadores tienen el peculiar hábito de escribir los números negativos con el signo menos a la derecha del número, en lugar de a la izquierda, donde lo ponen los especialistas en computación. Suponga que, en un intento por impresionar a su jefe, el encargado del centro de cómputo de un bufete contable cambia todos los números de todas las computadoras de modo que el bit de la extrema derecha (en lugar del de la extrema izquierda) haga las veces de bit de signo. No cabe duda de que el jefe quedaría muy impresionado... porque todo el software dejaría de funcionar correctamente. El hardware espera cierto formato para los enteros y no funciona correctamente si recibe algo distinto.

Consideremos ahora otro bufete contable, uno que acaba de obtener un contrato para verificar la deuda federal (cuánto debe el gobierno federal a sus acreedores). No sería factible usar aritmética de 32 bits aquí porque los números en cuestión son mayores que 2^{32} (unos 4000 millones). Una solución es usar dos enteros de 32 bits para representar cada número, lo que da 64 bits en total. Si la máquina no reconoce este tipo de números de **doble precisión**, todas las operaciones aritméticas que se realicen con ellos se tendrán que efectuar en software, pero las dos partes pueden estar en cualquier orden porque al hardware no le importa. Éste es un ejemplo de tipo de datos sin apoyo de hardware y por tanto sin una representación obligatoria en hardware. En las secciones que siguen examinaremos los tipos de datos que tienen apoyo de hardware y que por ello requieren formatos específicos.

5.2.1 Tipos de datos numéricos

Los tipos de datos se pueden dividir en dos categorías: numéricos y no numéricos. El principal de los tipos numéricos es el tipo entero. Los enteros tienen diversas longitudes, como 8, 16, 32 y 64 bits. Los enteros cuentan cosas (por ejemplo, el número de destornilladores que una ferretería tiene en existencia), identifican cosas (por ejemplo, números de cuenta bancarios) y mucho más. Casi todas las computadoras modernas almacenan los enteros en notación binaria de complemento a dos, aunque se han llegado a usar otros sistemas. Los números binarios se tratan en el Apéndice A.

Algunas computadoras reconocen enteros sin signo además de enteros con signo. En el caso de los enteros sin signo, no hay bit de signo y todos los bits contienen datos. Este tipo de datos tiene la ventaja de contar con un bit extra de modo que, por ejemplo, una palabra de 32 bits puede contener un entero sin signo dentro del intervalo de 0 a $2^{32} - 1$, inclusive. En cambio, un entero de 32 bits con signo en formato de complemento a dos sólo puede manejar números hasta $2^{31} - 1$ aunque, por supuesto, también puede manejar números negativos.

Si un número no se puede expresar como un entero, digamos 3.5, se usan números de punto flotante, los cuales se tratan en el Apéndice B. Su longitud es de 32, 64 o a veces 128 bits. Casi todas las computadoras cuentan con instrucciones para efectuar operaciones aritméticas de punto flotante, y muchas cuentan con registros distintos para contener operandos enteros y operandos de punto flotante.

Algunos lenguajes de programación, entre los que sobresale COBOL, permiten números decimales como tipo de datos. Las máquinas diseñadas para ejecutar con frecuencia programas en COBOL suelen manejar números decimales en hardware, por lo regular codificando un número decimal en 4 bits y empaquetando dos números decimales en cada byte (formato decimal codificado en binario). Sin embargo, la aritmética no funciona correctamente con números decimales empaquetados, por lo que se requieren instrucciones especiales para corregir la aritmética decimal. Dichas instrucciones necesitan conocer el acarreo de salida del bit 3. Es por ello que el código de condición a menudo incluye un bit de acarreo auxiliar. Por cierto, el tristemente famoso problema del año 2000 (Y2K) fue causado por programadores en COBOL que decidieron que sería más económico representar el año como dos dígitos decimales que como un número binario de 16 bits. Vaya optimación.

5.2.2 Tipos de datos no numéricos

Aunque las primeras computadoras en su mayoría se ganaban la vida triturando números, las computadoras modernas a menudo se usan para aplicaciones no numéricas, como procesamiento de textos o gestión de bases de datos. Para estas aplicaciones se requieren otros tipos de datos que en muchos casos cuentan con apoyo de instrucciones en el nivel ISA. Es evidente que los caracteres son importantes aquí, aunque no todas las computadoras ofrecen apoyo de hardware para ellos. Los códigos de caracteres más comunes son ASCII y UNICODE, que manejan caracteres de 7 bits y de 16 bits, respectivamente. Ambos se describieron en el capítulo 2.

No es raro que el nivel ISA cuente con instrucciones especiales diseñadas para manejar cadenas de caracteres, es decir, series consecutivas de caracteres. Estas cadenas en ocasiones

se delimitan con un carácter especial al final. Como alternativa, puede incluirse un campo de longitud de cadena para saber dónde termina ésta. Las instrucciones pueden realizar funciones de copiado, búsqueda, edición y otras con cadenas.

Los valores booleanos también son importantes. Un valor booleano puede adoptar uno de dos valores: verdadero (*true*) o falso (*false*). En teoría, un solo bit puede representar un valor booleano, con 0 como falso y 1 como verdadero (o viceversa). En la práctica se usa un byte o una palabra para cada valor booleano porque los bits individuales de un byte no tienen cada uno su propia dirección y por tanto el acceso a ellos no es fácil. Un sistema común sigue la convención de que 0 significa falso y cualquier otra cosa significa verdadero.

La única situación en la que un valor booleano normalmente se representa con un bit es cuando existe un arreglo de ellos, de modo que una palabra de 32 bits puede contener 32 valores booleanos. Semejante estructura de datos se denomina **mapa de bits** y se presenta en muchos contextos. Por ejemplo, se puede usar un mapa de bits para seguir la pista a los bloques libres de un disco. Si el disco tiene *n* bloques, el mapa de bits tiene *n* bits.

Nuestro último tipo de datos es el apuntador, que es sólo una dirección de máquina. Ya hemos visto muchos apuntadores. En las máquinas Mic-*x*, SP, PC, LV y CPP son ejemplos de apuntadores. Accesar a una variable que está a una distancia fija de un apuntador, que es como funciona ILOAD, es de lo más común en todas las máquinas.

5.2.3 Tipos de datos en el Pentium II

El Pentium II reconoce enteros con signo en complemento a dos, enteros sin signo, números decimales codificados en binario y números de punto flotante IEEE 754, como se indica en la figura 5-6. A causa de sus orígenes como una humilde máquina de 8 bits/16 bits, el Pentium II maneja también registros con estas longitudes y cuenta con muchas instrucciones para realizar operaciones aritméticas, booleanas y de comparación. Los operandos no tienen que estar alineados en la memoria, pero se obtiene un mejor desempeño si las direcciones de palabra son múltiplos de 4 bytes.

Tipo	8 bits	16 bits	32 bits	64 bits	128 bits
Entero con signo	×	×	×		
Entero sin signo	×	×	×		
Entero decimal codificado en binario	×				
Punto flotante			×	×	

Figur a 5-6. Tipos de datos numéricos del Pentium II. Los tipos que se manejan se indican con ×.

El Pentium II también es bueno para manipular caracteres ASCII de 8 bits: hay instrucciones especiales para copiar y buscar cadenas de caracteres. Estas instrucciones se pueden usar tanto con cadenas cuya longitud se conoce desde antes, como con cadenas cuyo final está marcado; se usan a menudo en bibliotecas de manipulación de cadenas.

5.2.4 Tipos de datos en el UltraSPARC II

El UltraSPARC II reconoce una amplia gama de formatos de datos, como se muestra en la figura 5-7. En el caso de enteros, el UltraSPARC II reconoce operandos de 8, 16, 32 y 64 bits, tanto con signo como sin signo. Los enteros con signo se expresan como complemento a dos. Se manejan operandos de punto flotante de 32, 64 y 128 bits, y se ajustan a la norma IEEE 754 (para los números de 32 y 64 bits). No se reconocen los números decimales codificados en binario. Todos los operandos deben estar alineados en la memoria.

Tipo	8 bits	16 bits	32 bits	64 bits	128 bits
Entero con signo	×	×	×	×	
Entero sin signo	×	×	×	×	
Entero decimal codificado en binario					
Punto flotante			×	×	×

Figura 5-7. Tipos de datos numéricos del UltraSPARC II.

El UltraSPARC II tiene una fuerte orientación hacia los registros y casi todas las instrucciones operan con registros de 64 bits. Los tipos de datos de caracteres y cadenas no cuentan con apoyo de instrucciones especiales en hardware.

5.2.5 Tipos de datos en la Máquina Virtual Java

Java es un lenguaje con tipos fuertes, lo que significa que todo operando tiene un tipo y tamaño específicos que se conocen en el momento de la compilación. Estos tipos se reflejan en los tipos de tiempo de ejecución reconocidos por JVM. JVM reconoce los tipos numéricos que se indican en la figura 5-8. Los enteros con signo se representan como complemento a dos. El lenguaje Java no incluye enteros sin signo y JVM no los reconoce, como tampoco reconoce los números decimales codificados en binario.

Tipo	8 bits	16 bits	32 bits	64 bits	128 bits
Entero con signo	×	×	×	×	
Entero sin signo					
Entero decimal codificado en binario					
Punto flotante			×	×	

Figura 5-8. Tipos de datos numéricos de la JVM.

JVM reconoce caracteres, pero como caracteres UNICODE en lugar de los caracteres ASCII de 8 bits más tradicionales. El apoyo para apuntadores está limitado en su mayor parte al uso interno del compilador y del sistema de tiempo de ejecución, pues los programas de usuario no pueden accesar directamente a apuntadores, los cuales se usan principalmente para hacer referencia a objetos.

5.3 FORMATOS DE INSTRUCCIONES

Una instrucción consiste en un código de operación generalmente acompañado por información adicional como la dirección de los operandos y el destino de los resultados. El tema general de especificar dónde están los operandos (es decir, sus direcciones) se llama **direccionamiento** y nos ocuparemos de él ahora.

La figura 5-9 muestra varios posibles formatos para instrucciones de nivel 2. Las instrucciones siempre tienen un código de operación que indica lo que hace la instrucción. Puede haber cero, una, dos o tres direcciones presentes.

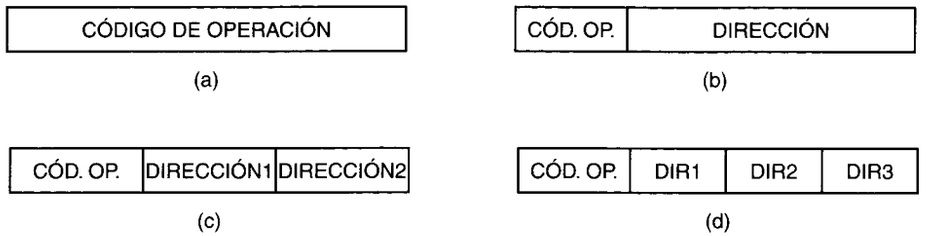


Figura 5-9. Cuatro formatos de instrucción comunes: (a) Instrucción con cero direcciones. (b) Instrucción con una dirección. (c) Instrucción con dos direcciones. (d) Instrucción con tres direcciones.

En algunas máquinas todas las instrucciones tienen la misma longitud; en otras podría haber muchas longitudes distintas. La longitud de las instrucciones puede ser menor, igual o mayor que la de una palabra. Exigir que todas las instrucciones tengan la misma longitud es más sencillo y facilita la decodificación, pero a menudo desperdicia espacio, ya que todas las instrucciones tienen que ser tan largas como la más larga. En la figura 5-10 se muestran algunas posibles relaciones entre la longitud de las instrucciones y de las palabras.

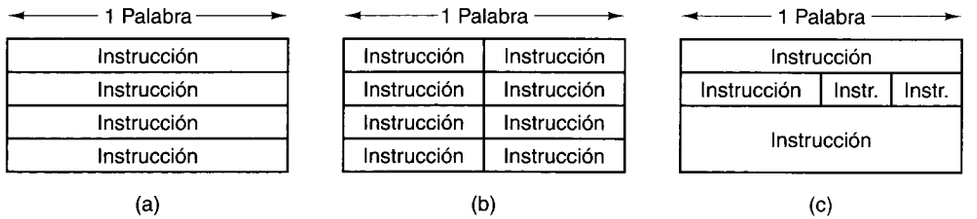


Figura 5-10. Algunas posibles relaciones entre la longitud de las instrucciones y de las palabras.

5.3.1 Criterios de diseño para los formatos de instrucciones

Cuando un equipo de diseño de computadoras tiene que escoger los formatos de instrucciones para su máquina, debe considerar varios factores. No debemos subestimar la dificultad de esta decisión, la cual debe tomarse en una etapa temprana del diseño de una computadora nueva. Si la computadora tiene éxito comercial, el conjunto de instrucciones podría sobrevivir 20 años o más. La capacidad para añadir instrucciones nuevas y aprovechar otras oportunidades que surgen durante un tiempo largo tiene gran importancia, pero sólo si la arquitectura –y la compañía que la construye– sobreviven el tiempo suficiente para que la arquitectura sea un éxito.

La eficiencia de una ISA dada depende en gran medida de la tecnología con que una computadora se va a implementar. Con los años, esta tecnología sufrirá grandes cambios, y en retrospectiva se verá que algunas de las decisiones de la ISA no fueron las mejores. Por ejemplo, si los accesos a la memoria son rápidos, un diseño basado en pila (como JVM) es bueno, pero si son lentos entonces lo mejor es tener muchos registros (como en el UltraSPARC II). Los lectores que creen que esta decisión es fácil podrían tomar una hoja de papel y anotar sus predicciones para (1) una velocidad de reloj de CPU típica y (2) un tiempo de acceso a RAM típico, para las computadoras de 20 años en el futuro. Doble bien la hoja de papel y guárdela 20 años. Luego desdóblela y léala. Quienes carezcan de humildad pueden olvidarse de la hoja de papel y simplemente publicar sus predicciones en Internet ahora.

Desde luego, aun los diseñadores con gran visión pueden tomar decisiones subóptimas. E incluso si pudieran tomar todas las decisiones correctas, también tienen que enfrentar el corto plazo. Si esta ISA elegante es un poco más costosa que sus competidoras actuales mucho menos atractivas, la compañía podría no sobrevivir el tiempo suficiente para que el mundo aprecie la elegancia de la ISA.

Si todo lo demás es igual, las instrucciones cortas son mejores que las largas. Un programa que consiste en n instrucciones de 16 bits ocupa la mitad del espacio de memoria que ocupan n instrucciones de 32 bits. En vista de la tendencia descendente de los precios de las memorias, este factor podría ser menos importante en el futuro, si no fuera por el hecho de que el software crece a un ritmo más rápido que aquél con que bajan los precios de la memoria.

Además, minimizar el tamaño de las instrucciones podría hacerlas más difíciles de decodificar o de traslapar. Por tanto, la reducción del tamaño de las instrucciones al mínimo debe compararse contra el tiempo que se requiere para decodificar y ejecutar las instrucciones.

Otra razón para minimizar la longitud de las instrucciones ya es importante y se vuelve más importante a medida que aumenta la rapidez de los procesadores: el ancho de banda de la memoria (el número de bits/s que la memoria puede proporcionar). El impresionante aumento en la rapidez de los procesadores durante la última década no ha ido acompañado por un aumento comparable en el ancho de banda de las memorias. Una de las restricciones a las que cada vez están más sujetos los procesadores tiene su origen en la incapacidad del sistema de memoria para proporcionar instrucciones y operandos al mismo ritmo que el procesador puede consumirlos. Cada memoria tiene un ancho de banda determinado por su tecnología y su diseño de ingeniería. El cuello de botella del ancho de banda aplica no sólo a la memoria principal, sino también a todas las cachés.

Si el ancho de banda de una caché de instrucciones es de t bps y la longitud media de las instrucciones es de r bits, la caché podrá suministrar cuando más t/r instrucciones por segundo. Cabe señalar que éste es un *límite superior* de la rapidez con que el procesador puede ejecutar instrucciones, aunque se están realizando investigaciones cuya meta es superar esta barrera al parecer infranqueable. Es evidente que la rapidez con que se pueden ejecutar las instrucciones (es decir, la velocidad del procesador) podría estar limitada por la longitud de

las instrucciones. El hecho de que las instrucciones sean más cortas implica un procesador más rápido. Dado que los procesadores modernos pueden ejecutar varias instrucciones en cada ciclo de reloj, es imperativo traer varias instrucciones en cada ciclo de reloj. Este aspecto de la caché de instrucciones hace que el tamaño de las instrucciones sea un criterio de diseño importante.

Un segundo criterio de diseño es suficiente espacio en el formato de las instrucciones para expresar todas las operaciones deseadas. No puede haber una máquina con 2^n operaciones e instrucciones de menos de n bits. Simplemente no habría suficiente espacio en el código de operación para indicar cuál instrucción se necesita. Además, la experiencia ha demostrado una y otra vez la insensatez de no dejar libre un número sustancial de códigos de operación para adiciones futuras al conjunto de instrucciones.

Un tercer criterio tiene que ver con el número de bits de un campo de dirección. Considere el diseño de una máquina con caracteres de 8 bits y una memoria principal que debe contener 2^{32} caracteres. Los diseñadores podrían optar por asignar direcciones consecutivas a unidades de 8, 16, 24 o 32 bits, además de otras posibilidades.

Imagine qué sucedería si el equipo de diseño degenerara en dos facciones opuestas, una que propone que el byte de 8 bits sea la unidad básica de la memoria, y otra partidaria de usar la palabra de 32 bits como unidad básica de memoria. El primer grupo propondría una memoria de 2^{32} bytes, numerados 0, 1, 2, 3, ..., 4,294,967,295. El segundo grupo propondría una memoria de 2^{30} palabras numeradas 0, 1, 2, 3, ..., 1,073,741,823.

El primer grupo señalaría que para comparar dos caracteres en la organización de 32 bits el programa no sólo tendría que obtener las palabras que contienen los caracteres, sino que también tendría que extraer cada carácter de su palabra para poder compararlos. Ello costaría instrucciones adicionales y por tanto desperdiciaría espacio. La organización de 8 bits, en cambio, asigna una dirección a cada carácter, lo que facilita mucho las comparaciones.

Los partidarios de las palabras de 32 bits contraatacarían señalando que su propuesta sólo requiere 2^{30} direcciones distintas, lo que implica direcciones de sólo 30 bits, mientras que la propuesta de bytes de 8 bits requiere 32 bits para direccionar la misma memoria. Direcciones más cortas implican instrucciones más cortas, lo que además de ocupar menos espacio requiere menos tiempo para la obtención. Como alternativa, podrían conservarse las direcciones de 32 bits para hacer referencia a una memoria de 16 GB en lugar de una miserable memoria de 4 GB.

Este ejemplo demuestra que para tener una definición de memoria más fina hay que pagar el precio de direcciones más largas y por tanto instrucciones más largas. Lo máximo en definición es una organización de memoria en la que es posible direccionar cada bit individualmente (como en la Burroughs B1700). En el otro extremo está una memoria que consiste en palabras muy largas (como la serie Cyber de CDC que tenía palabras de 60 bits).

Los sistemas de cómputo modernos han llegado a un término medio que, en cierto sentido, captura lo peor de ambos enfoques. Esta organización requiere todos los bits necesarios para direccionar bytes individuales, pero todos los accesos a la memoria leen una, dos o a veces cuatro palabras a la vez. Por ejemplo, la lectura de un byte de la memoria en el UltraSPARC II trae por lo menos 16 bytes (véase la figura 3-47) y probablemente toda una línea de caché de 64 bytes.

5.3.2 Expansión de códigos de operación

En la sección anterior vimos cómo puede establecerse un equilibrio entre la cortedad de las instrucciones y la definición de la memoria. En esta sección examinaremos equilibrios en los que intervienen tanto los códigos de operación como las direcciones. Considere una instrucción de $(n + k)$ bits con un código de operación de k bits y una sola dirección de n bits. Esta instrucción permite 2^k operaciones distintas y 2^n celdas de memoria direccionables. O bien, los mismos $(n + k)$ bits podrían dividirse en un código de operación de $(k - 1)$ bits y una dirección de $(n + 1)$ bits, lo que implica la mitad de las instrucciones pero el doble de memoria direccionable o bien la misma cantidad de memoria pero con el doble de definición. Un código de operación de $(k + 1)$ bits y direcciones de $(n - 1)$ bits implica más operaciones pero a expensas de reducir el número de celdas direccionables o bien la definición de la memoria. Es posible establecer equilibrios muy complejos entre los bits del código de operación y los de las direcciones, además de los equilibrios sencillos que acabamos de describir. El esquema que analizaremos en los párrafos que siguen se llama **expansión de código de operación**.

La forma más clara de explicar la expansión de códigos de operación es con un ejemplo. Consideremos una máquina en la que las instrucciones tienen 16 bits y las direcciones tienen 4 bits, como se muestra en la figura 5-11. Esta situación podría ser razonable para una máquina que tiene 16 registros (y por tanto direcciones de registro de 4 bits) en los que se efectúan todas las operaciones aritméticas. Un diseño sería un código de operación de 4 bits y tres direcciones en cada instrucción, para dar instrucciones de 16 bits.

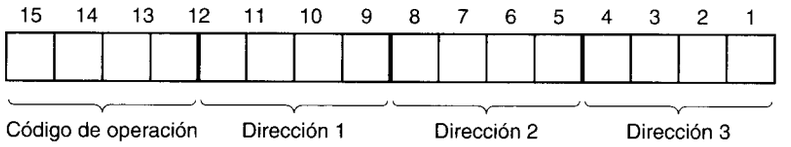


Figura 5-11. Instrucción con código de operación de 4 bits y tres campos de dirección de 4 bits.

Sin embargo, si los diseñadores necesitan 15 instrucciones de tres direcciones, 14 instrucciones de dos direcciones, 31 instrucciones de una dirección y 16 instrucciones que no requieren una dirección, pueden usar los códigos de operación del 0 al 14 como instrucciones de tres direcciones pero interpretar el código de operación 15 de forma distinta (vea la figura 5-12).

El código 15 indica que el código de operación está contenido en los bits 8 a 15, no en los bits 12 a 15. Los bits 0 a 3 y 4 a 7 forman dos direcciones, como siempre. Las 14 instrucciones de dos direcciones tienen 1111 en los 4 bits de la izquierda, y números del 0000 al 1101 en los bits 8 a 11. Las instrucciones que tienen 1111 en los bits de la izquierda y 1110 o bien 1111 en los bits 8 a 11 reciben un trato especial: se tratan como si sus códigos de operación estuvieran en los bits 4 a 15. El resultado es 32 códigos de operación nuevos. Puesto que sólo se necesitan 31, el código de operación 111111111111 se interpreta como señal de que el código de operación real está en los bits 0 a 15, lo que da 16 instrucciones que no usan ninguna dirección.

A medida que avanzaba nuestra explicación, los códigos de operación iban creciendo: las instrucciones de tres direcciones tienen un código de operación de 4 bits; las de dos

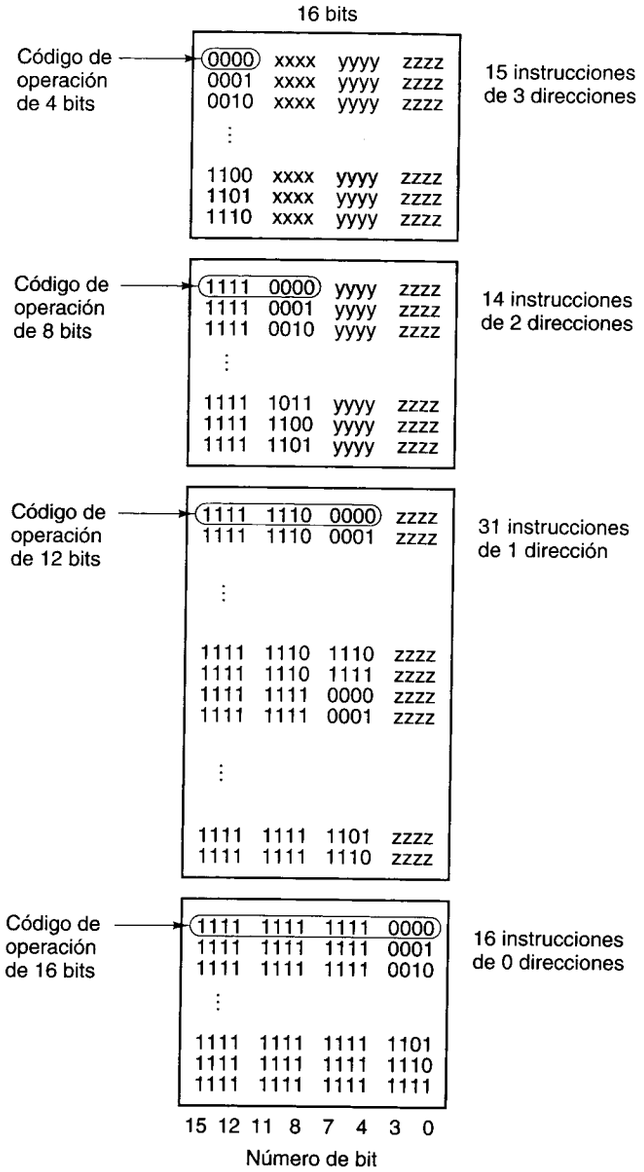


Figura 5-12. Código de operación expansible que permite tener 15 instrucciones de tres direcciones, 14 de dos direcciones, 31 de una dirección y 16 de cero direcciones. Los campos marcados *xxxx*, *yyyy* y *zzzz* son campos de dirección de 4 bits.

direcciones tienen un código de operación de 8 bits, las de una dirección tienen un código de operación de 12 bits, y las de cero instrucciones tienen un código de operación de 16 bits.

La idea de expandir los códigos de operación ilustra un equilibrio entre el espacio para códigos de operación y el espacio para otra información. En la práctica, la expansión de códigos de operación no son tan nítidas y regulares como en nuestro ejemplo. De hecho, la capacidad para usar códigos de operación con tamaño variable se puede aprovechar de una de dos maneras. Primera, puede hacerse que todas las instrucciones tengan la misma longitud, asignando los códigos de operación más cortos a las instrucciones que necesitan más bits para especificar otras cosas. Segunda, se puede minimizar el tamaño *medio* de las instrucciones escogiendo códigos de operación más cortos para las instrucciones comunes y más largos para las instrucciones menos usadas.

Llevando la idea de los códigos de operación de longitud variable al extremo, es posible minimizar la longitud promedio de las instrucciones codificando cada instrucción de modo tal que el número de bits requerido sea el mínimo. El problema es que entonces las instrucciones tendrían diversos tamaños que ni siquiera estarían alineados respecto a las fronteras de bytes. Aunque han existido algunas ISA con esta propiedad (como el malhadado Intel 432), la importancia de la alineación es tan grande para decodificar rápidamente las instrucciones que tal grado de optimación es con toda seguridad contraproducente. No obstante, es común aplicarlo en el nivel de bytes. Más adelante examinaremos la ISA de JVM, por ejemplo, para ver cómo se han escogido cuidadosamente los formatos de las instrucciones a fin de minimizar el tamaño de los programas.

5.3.3 Los formatos de instrucciones del Pentium II

Los formatos de instrucción del Pentium II son muy complejos e irregulares, con hasta seis campos de longitud variable, cinco de los cuales son opcionales. El patrón general se muestra en la figura 5-13. Se llegó a tal patrón porque la arquitectura evolucionó a lo largo de muchas generaciones e incluyó algunas decisiones poco afortunadas en etapas tempranas. En aras de la compatibilidad con modelos anteriores, se respetaron dichas decisiones en procesadores posteriores. En general, para las instrucciones de dos operandos, si un operando está en la memoria, el otro podría no estar en la memoria. Por ello existen instrucciones que suman dos registros, que suman un registro a un valor contenido en la memoria, y que suman un valor de la memoria a un registro, pero no que suman una palabra de la memoria a otra.

En las primeras arquitecturas Intel todos los códigos de operación eran de un byte, aunque se usaba mucho el concepto de byte prefijo para modificar algunas instrucciones. Un **byte prefijo** es un código de operación extra antepuesto a una instrucción para modificar su acción. La instrucción WIDE de IJVM y JVM es un ejemplo de byte prefijo. Lamentablemente, en algún punto durante la evolución, Intel se quedó sin códigos de operación, por lo que uno de ellos, 0xFF, se designó como **código de escape** para poder incluir un segundo byte de instrucción.

Los bits individuales de los códigos de operación del Pentium II no proporcionan mucha información acerca de la instrucción. La única estructura que tiene el campo de código de operación es el uso del bit de orden bajo en algunas instrucciones para indicar byte/palabra, y el uso del bit adyacente para indicar si la dirección de memoria (si está presente) es la fuente

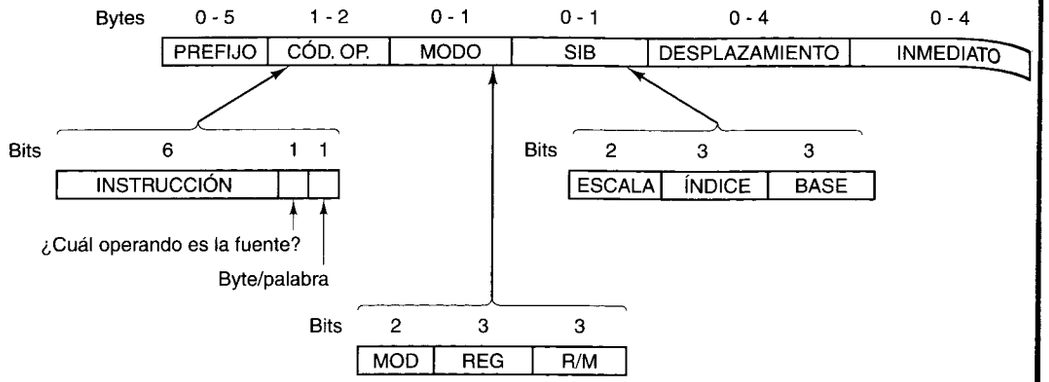


Figura 5-13. Formatos de instrucciones del Pentium II.

o el destino. Así, en general, es preciso decodificar cabalmente el código de operación para determinar qué clase de operación debe efectuarse, y por tanto qué longitud tiene la instrucción. Esto dificulta las implementaciones de alto desempeño, porque se requiere mucho trabajo de decodificación para siquiera determinar dónde comienza la siguiente instrucción.

En la mayor parte de las instrucciones que hacen referencia a un operando en la memoria, el byte de código de operación va seguido de un segundo byte que describe el operando. Estos 8 bits se dividen en un campo MOD de dos bits y dos campos de registro de tres bits, REG y R/M. A veces los tres primeros bits de este byte se usan como extensión del código de operación, lo que da un total de 11 bits para el código de operación. Sin embargo, el campo de modo de dos bits implica que sólo hay cuatro formas de direccionar operandos y uno de los operandos siempre debe ser un registro. Lógicamente, podría especificarse cualquiera de EAX, EBX, ECX, EDX, ESI, EDI, EBP o ESP como cualquiera de los registros, pero las reglas de codificación prohíben algunas combinaciones, por lo que se destinan a casos especiales. Algunos modos requieren un byte adicional, llamado **SIB (escala, índice, base**; *Scale, Index, Base*), que amplía la especificación. Este esquema no es ideal, sino un término medio dadas las demandas opuestas de compatibilidad hacia atrás y el deseo de añadir funciones nuevas que no se contemplaron originalmente.

Además de todo esto, algunas instrucciones tienen 1, 2 o 4 bytes más que especifican una dirección de memoria (desplazamiento) y tal vez otros 1, 2 o 4 bytes que contienen una constante (operando inmediato).

5.3.4 Los formatos de instrucciones del UltraSPARC II

La ISA del UltraSPARC II consiste exclusivamente en instrucciones de 32 bits, alineadas en la memoria. En general, las instrucciones son sencillas y especifican una sola acción. Una instrucción aritmética representativa especifica dos registros que proporcionan los operandos fuente y un solo registro de destino. Una variante permite a la instrucción proporcionar una constante de 13 bits con signo en lugar de uno de los registros. En el caso de un LOAD, se suman dos registros (o un registro y una constante de 13 bits) para especificar la dirección de memoria que se leerá. Los datos se colocan en el registro que se especifica.

El SPARC original tenía un número muy limitado de formatos de instrucciones, que se ilustran en la figura 5-14. Con el tiempo, se añadieron nuevos formatos. En el momento de escribirse esta obra, el número era 31 y en aumento. (¿Pasará mucho tiempo antes de que alguna compañía anuncie "la máquina RISC más compleja del mundo"?) Casi todas las nuevas variantes se obtuvieron recortándole unos cuantos bits a algún campo. Por ejemplo, las instrucciones de ramificación originales usaban el formato 3, con una distancia de 22 bits. Cuando se añadieron las ramificaciones predichas, se eliminaron tres de los 22 bits, pues se usaba uno para la predicción (tomar/no tomar la rama) y dos para especificar cuál conjunto de bits de código de condición se debía usar. Esto dejó un desplazamiento de 19 bits. Otro ejemplo es el de las muchas instrucciones que se añadieron para convertir un tipo de datos a otro (entero a punto flotante, etc.). Muchas de éstas usan una variante del formato 1b en la que el campo INMEDIATO se dividió en un campo de 5 bits que da el registro fuente y un campo de 8 bits que proporciona más bits para el código de operación. Sin embargo, la generalidad de las instrucciones sigue usando los formatos que se muestran en la figura.

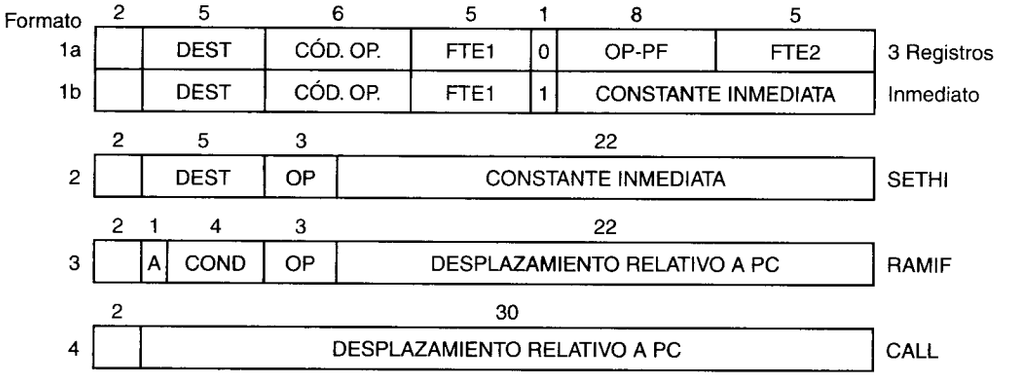


Figura 5-14. Los formatos de instrucción del SPARC original.

Los primeros dos bits de cada instrucción ayudan a determinar el formato de la instrucción y le dicen al hardware dónde puede encontrar el resto del código de operación, si hay más. En el formato 1a, las dos fuentes son registros; en el formato 1b, una fuente es un registro y la otra es una constante dentro del intervalo –4096 a +4095. El bit 13 selecciona cuál de las dos se usará. (El bit de la extrema derecha es el bit 0.) En ambos casos, el destino siempre es un registro. Hay suficiente espacio de codificación para un máximo de 64 instrucciones, algunas de las cuales están reservadas para uso futuro.

Con sólo instrucciones de 32 bits, no es posible incluir una constate de 32 bits en la instrucción. La instrucción SETHI establece 22 bits, y deja espacio para que otra instrucción establezca los otros 10. SETHI es la única instrucción que usa este formato.

Las ramificaciones condicionales sin predicción usan el formato 3, y el campo COND indica cuál condición debe probarse. El bit A tiene que ver con evitar ranuras de retraso en ciertas condiciones. Las ramificaciones con predicción usan el mismo formato, pero con un desplazamiento de 19 bits, como se mencionó antes.

El último formato es para la instrucción CALL, que sirve para invocar un procedimiento. Esta instrucción es especial porque es la única en la que se requieren 30 bits de datos para especificar una dirección. En esta ISA hay un solo código de operación de dos bits. La dirección es la dirección objetivo dividida entre 4, lo que hace que el alcance sea de aproximadamente $\pm 2^{31}$ bytes relativo a la instrucción actual.

5.3.5 Los formatos de instrucciones de la JVM

En general, los formatos de instrucciones JVM son muy sencillos. En la figura 5-15 se muestran todos ellos. Su sencillez probablemente se debe al hecho de que la JVM es nueva. Ya veremos dentro de 10 años. Todas las instrucciones inician con un código de operación de un byte. Algunos códigos van seguidos de un segundo byte que puede ser un índice (como en ILOAD), una constante (como en BIPUSH) o un indicador de tipo de datos (como en NEWARRAY, que crea en el montículo un arreglo unidimensional del tipo indicado). El formato 3 es prácticamente igual al segundo, sólo que usa una constante de 16 bits en lugar de una de 8 bits (por ejemplo, WIDE ILOAD o GOTO). El formato 4 sólo se usa para IINC. El formato 5 sólo se usa para MULTINEWARRAY, que crea un arreglo multidimensional en el montículo. El formato 6 sólo se usa con INVOKEINTERFACE, que llama a un método en ciertas circunstancias. El formato 7 sólo se usa con WIDE IINC para proporcionar un índice de 16 bits y una constante de 16 bits que se suman a la variable seleccionada. El formato 8 sólo se usa con las instrucciones WIDE GOTO y WIDE JSR que hacen posibles ramificaciones distantes y ciertas llamadas de métodos. El último formato sólo se usa con dos instrucciones que sirven para implementar el enunciado switch de Java. En síntesis, con la excepción de ocho instrucciones específicas, todas las instrucciones de la JVM usan los formatos 1, 2 o 3, que son cortos y sencillos.

De hecho, la situación es aún mejor de lo que parece. Las instrucciones de la Máquina Virtual Java se han codificado de tal modo que las instrucciones más comunes se especifican con un solo byte. De las 256 instrucciones posibles dentro de un byte, muchas de ellas son en realidad casos especiales de la forma general de una instrucción que vimos en IJVM.

Por ejemplo, veamos cómo se carga una variable local en Java. En realidad hay tres formas distintas de especificar esto: la más corta es la más común y la más larga cubre todos los posibles casos. JVM tiene una instrucción ILOAD que usa un índice de 8 bits para especificar la variable local que se debe meter en la pila. También vimos cómo el prefijo WIDE permite usar el mismo código de operación para especificar cualquiera de las primeras 65,536 entradas del marco de variables locales. Sin embargo, el uso de WIDE ILOAD requiere 4 bytes: uno para WIDE, uno para ILOAD y dos para el índice de 16 bits. Esta división se justifica por el hecho de que casi todos los ILOAD usan una de las primeras 256 variables locales. El prefijo WIDE se necesita por generalidad, pero pocas veces se usa.

Pero la JVM va más lejos aún. Puesto que los parámetros que se pasan a un método se pasan en las primeras palabras del marco de variables locales, los usos más comunes de ILOAD son las entradas con índices bajos. Los diseñadores de la JVM decidieron que estas posiciones podrían ser tan comunes que valdría la pena asignar códigos de operación de un byte distintos para cada una de estas combinaciones. ILOAD_0 (0x1A) almacena la variable local 0 en la pila. Esto equivale *exactamente* a la instrucción de dos bytes ILOAD 0, excepto

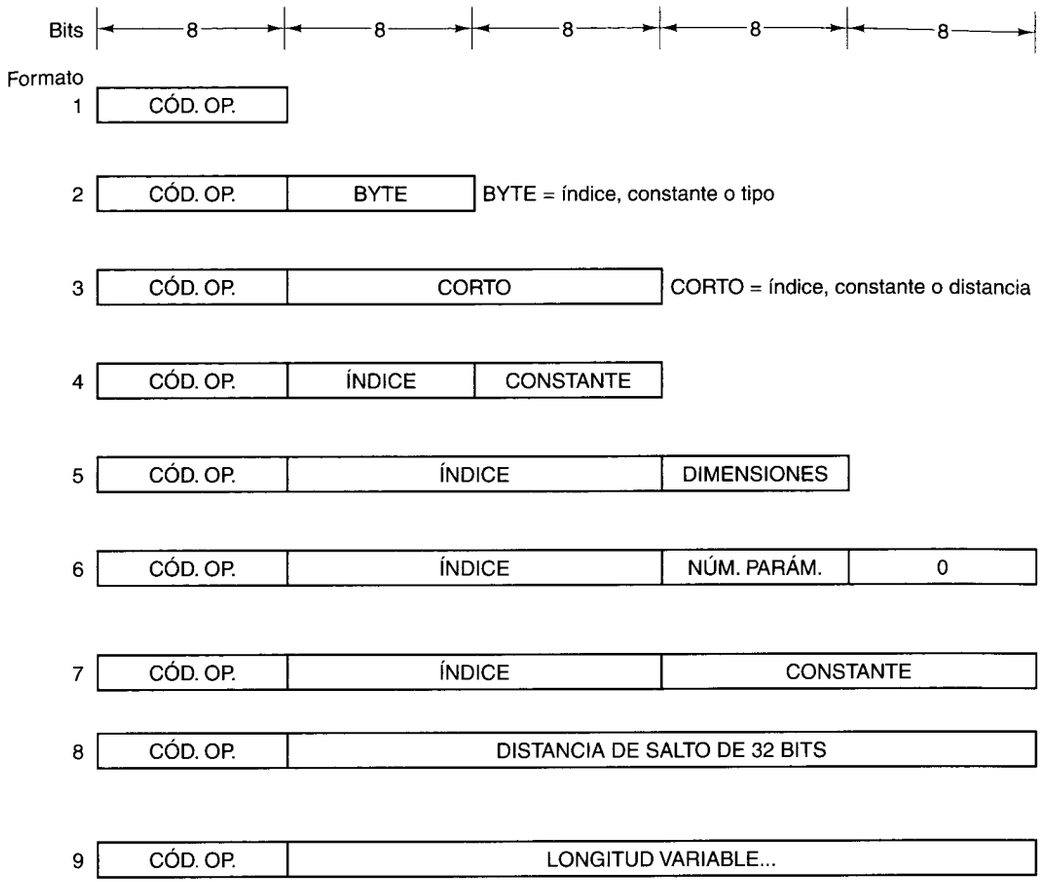


Figura 5-15. Los formatos de instrucción JVM.

que ocupa un byte en lugar de dos. Asimismo, ILOAD_1, ILOAD_2 e ILOAD_3 (códigos de operación 0x1B, 0x1C y 0x1D, respectivamente) almacenan las variables locales 1, 2 y 3 en la pila. Cabe señalar que la variable local 1, por ejemplo, se puede cargar de tres maneras: ILOAD_1, ILOAD 1 o WIDE ILOAD 1.

Existen variaciones similares de muchas instrucciones, como instrucciones especiales que equivalen exactamente a BIPUSH con constantes de 0, 1, 2, 3, 4 y 5, así como –1. También hay instrucciones especiales para guardar variables de la pila en las primeras cuatro palabras del espacio de variables locales.

Cabe señalar que estas variaciones no carecen de costo. Sólo pueden especificarse 256 instrucciones únicas con un byte. La decisión de dedicar cuatro de ellas a la carga de las primeras cuatro variables locales consume cuatro de las 256 instrucciones disponibles. Y si añadimos la instrucción ILOAD básica vemos que ya se ocuparon cinco instrucciones. Desde luego, el prefijo WIDE también consume uno de los 256 valores disponibles (y ni siquiera es una instrucción, sólo un prefijo), pero es más general porque se aplica a muchos otros códigos de operación.

Los diseñadores de la JVM usaron un método un poco diferente para especificar la carga de operandos desde la reserva de constantes, tal vez porque esperaban una diferencia en la distribución de las direcciones especificadas. Incluyeron dos versiones de la instrucción: LDC y LDC_W. La segunda forma, que se incluyó en la IJVM, es la forma general que puede traer cualquiera de las 65,536 palabras de la reserva de constantes. La primera forma requiere un índice de sólo un byte, pero sólo puede traer cualquiera de las primeras 256 palabras. Así, la obtención de cualquiera de las primeras 256 palabras puede efectuarse con una instrucción de dos bytes, mientras que la obtención de cualquier palabra se puede efectuar con una instrucción de tres bytes. Estas opciones consumen dos de los 256 códigos de operación. Cabe señalar que el conjunto de instrucciones habría sido más limpio y regular si los diseñadores hubieran escogido la misma técnica que con ILOAD, es decir, el uso del prefijo WIDE, en lugar de la instrucción LDC_W. Sin embargo, de hacerse así la obtención de constantes más allá de las primeras 256 habría requerido cuatro bytes en lugar de tres.

La técnica de combinar códigos de operación e índices en un solo byte y luego asignar los 256 bytes disponibles según la frecuencia de uso fue propuesta en 1978 por el autor, pero tuvieron que pasar más de dos décadas para que se popularizara (Tanenbaum, 1978).

5.4 DIRECCIONAMIENTO

Un diseño cuidadoso de los códigos de operación es una parte importante de la ISA. Sin embargo, una porción considerable de los bits de un programa se gastan en especificar de dónde provienen los operandos, más que las operaciones que se efectuarán con ellos. Considere una instrucción ADD que requiere la especificación de tres operandos: dos fuentes y un destino. (Se acostumbra usar el término *operando* para referirse a los tres, pero el destino es el lugar donde se coloca el resultado.) De alguna manera, la instrucción ADD debe indicar dónde están los operandos y dónde debe ponerse el resultado. Si las direcciones de memoria tienen 32 bits, la especificación ingenua de esta instrucción requiere tres direcciones de 32 bits además del código de operación. Las direcciones ocupan muchos más bits que el código de operación.

Se usan dos métodos generales para reducir el tamaño de la especificación. Primero, si un operando va a usarse varias veces, puede colocarse en un registro. El uso de un registro para una variable tiene dos ventajas: el acceso es más rápido y se requieren menos bits para especificar el operando. Si hay 32 registros, cualquiera de ellos puede especificarse con sólo cinco bits. Si la instrucción ADD puede efectuarse usando sólo operandos en registros, bastan 15 bits para especificar los tres operandos, en lugar de los 96 que se necesitarían si todos estuvieran en la memoria.

Desde luego, el uso de registros no es gratuito. De hecho, hay ocasiones en que su uso empeora el problema. Si primero hay que cargar en un registro un operando de la memoria, se ocuparán más bits de especificación que si sólo se especificara la posición en memoria. Primero se necesita una instrucción LOAD para colocar el operando en un registro. Esto no sólo requiere un código de operación, sino también la dirección de memoria completa y la especificación del registro objetivo. Por tanto, si un operando sólo se va a usar una vez, no vale la pena colocarlo en un registro.

Por fortuna, años de mediciones muestran que los operandos se reutilizan mucho. Por consiguiente, casi todas las arquitecturas recientes cuentan con un gran número de registros y casi todos los compiladores hacen hasta lo imposible por mantener variables locales en esos registros, lo que elimina muchas referencias a la memoria. Esto reduce tanto el tamaño de los programas como el tiempo de ejecución.

Un segundo método para reducir el tamaño de la especificación consiste en especificar uno o más operandos implícitamente. Hay varias técnicas para hacerlo. Una de ellas es utilizar una sola especificación para un operando fuente y para el destino. Mientras que una instrucción ADD general de tres direcciones podría usar la forma

```
DESTINO = FUENTE1 + FUENTE2
```

una instrucción de dos registros podría estar limitada a la forma

```
REGISTRO2 = REGISTRO2 + FUENTE1
```

Esta instrucción es destructiva en el sentido de que el contenido de REGISTRO2 no se conserva. Si el valor original se va a necesitar posteriormente, primero habrá que copiarlo en otro registro. El equilibrio aquí es que las instrucciones de dos direcciones son más cortas, pero son menos generales. Diferentes diseñadores toman diferentes decisiones. El Pentium II, por ejemplo, usa instrucciones de dos direcciones en el nivel ISA, mientras que el UltraSPARC II usa instrucciones de tres direcciones.

Habiendo reducido el número de operandos de nuestra instrucción ADD de tres a dos, no nos detengamos. Las primeras computadoras tenían sólo un registro llamado **acumulador**. La instrucción ADD, por ejemplo, siempre sumaba una palabra de la memoria al acumulador, por lo que sólo era necesario especificar un operando (el de la memoria). Esta técnica funcionaba bien con cálculos sencillos, pero si se requerían varios resultados intermedios el acumulador tenía que escribirse en la memoria y más adelante tenía que recuperarse. Es por esto que la técnica se ha dejado de usar.

Ahora hemos pasado de una ADD de tres direcciones a una de dos direcciones y luego a una con una sola dirección. ¿Qué falta? ¿Cero direcciones? Sí. En el capítulo 4 vimos cómo IJVM usa una pila. La instrucciones IADD de IJVM no tiene direcciones. Tanto la fuente como el destino están implícitos. Veremos el direccionamiento de pila un poco más adelante.

5.4.1 Modos de direccionamiento

Hasta aquí no nos hemos fijado mucho en la forma en que los bits de un campo de dirección se interpretan para encontrar el operando. Una posibilidad es que contienen la dirección en memoria del operando. Además del campo grande que se necesita para especificar la dirección en memoria completa, este método tiene la restricción adicional de que la dirección se debe determinar en el momento de la compilación. Hay otras posibilidades que permiten acortar las especificaciones y también determinar las direcciones dinámicamente.

En las secciones que siguen exploraremos algunas de estas formas, llamadas **modos de direccionamiento**.

5.4.2 Direccionamiento inmediato

La forma más sencilla de especificar un operando es que la parte de dirección de la instrucción realmente contenga el operando mismo en lugar de una dirección u otra información que describa la ubicación del operando. Semejante operando se llama **operando inmediato** porque se obtiene automáticamente de la memoria al mismo tiempo que se obtiene la instrucción misma; por tanto, está disponible inmediatamente para usarse. En la figura 5-16 se muestra una posible instrucción para cargar el registro R1 con la constante 4.

Figura 5-16. Instrucción inmediata para cargar 4 en el registro 1.

El direccionamiento inmediato tiene la virtud de no requerir una referencia extra a la memoria para obtener el operando. Su desventaja es que con esta técnica sólo se puede suministrar una constante. Además, el número de valores está limitado por el tamaño del campo. No obstante, muchas arquitecturas usan esta técnica para especificar constantes enteras pequeñas.

5.4.3 Direccionamiento directo

Un método para especificar un operando en la memoria es sencillamente dar su dirección completa. Este modo se llama **direccionamiento directo**. Al igual que el direccionamiento inmediato, el uso del direccionamiento directo tiene limitaciones: la instrucción siempre accesará a la misma localidad de memoria exactamente. Si bien el valor contenido en ella puede cambiar, la dirección no puede alterarse. Por ello, el direccionamiento directo sólo sirve para accesar variables globales cuya dirección se conoce en el momento de la compilación. Por otra parte, muchos programas tienen variables globales, por lo que este modo se usa ampliamente. Los detalles de cómo la computadora sabe cuáles direcciones son inmediatas y cuáles son directas se explicarán más adelante.

5.4.4 Direccionamiento por registro

El direccionamiento por registro es igual al directo en lo conceptual pero especifica un registro en lugar de una posición de memoria. Dada la importancia de los registros (por su acceso tan rápido y sus direcciones tan cortas) este modo de direccionamiento es el más común en la mayor parte de las computadoras. Muchos compiladores se esfuerzan por determinar a cuáles variables se accesará con mayor frecuencia (por ejemplo, el índice de un ciclo) y colocan estas variables en registros.

Este modo de direccionamiento se conoce simplemente como **modo de registro**. En las arquitecturas de carga/almacenamiento como el UltraSPARC II, casi todas las instrucciones

emplean este modo de direccionamiento exclusivamente. La única ocasión en que no se usa este modo de direccionamiento es cuando un operando se pasa de la memoria a un registro (instrucción LOAD) o de un registro a la memoria (instrucción STORE). Incluso en el caso de esas instrucciones, uno de los operandos es un registro: dónde está o dónde se colocará la palabra de memoria.

5.4.5 Direccionamiento indirecto por registro

En este modo, el operando que se especifica viene de la memoria o va a ella, pero su dirección no está fija en la instrucción, como en el direccionamiento directo. En vez de ello, la dirección está contenida en un registro. Cuando una dirección se usa de esta manera, se llama **apuntador**. Una gran ventaja del direccionamiento indirecto por registro es que puede hacer referencias a la memoria sin pagar el precio de tener una dirección de memoria completa en la instrucción. También permite usar diferentes palabras de la memoria en diferentes ejecuciones de la instrucción.

Para ver qué utilidad podría tener usar una palabra diferente en cada ejecución, imagine un ciclo que procesa los elementos de un arreglo unidimensional de enteros con 1024 elementos y calcula la suma de los elementos en el registro R1. Fuera del ciclo, puede hacerse que algún otro registro, digamos R2, apunte al primer elemento del arreglo, y que otro registro, digamos R3, apunte a la primera dirección más allá del arreglo. Con 1024 enteros de 4 bytes cada uno, si el arreglo comienza en *A*, la primera dirección más allá del arreglo será *A* + 4096. En la figura 5-17 se muestra un código de ensamblador típico para realizar este cálculo en una máquina de dos direcciones.

```
        MOV R1,#0           ; acumular la suma en R1, inicialmente 0
        MOV R2,#A           ; R2 = dirección del arreglo A
        MOV R3,#A+1024      ; R3 = dirección de la primera palabra después de A
CICLO:  ADD R1,(R2)         ; indirecto por registro R2 para obtener operando
        ADD R2,#4           ; incrementar R2 en una palabra (4bytes)
        CMP R2,R3           ; ¿ya terminamos?
        BLT LOOP            ; si R2 < R3, no hemos acabado; continuar
```

Figura 5-17. Programa genérico en ensamblador para calcular la suma de los elementos de un arreglo.

En este pequeño programa usamos varios modos de direccionamiento. Las primeras tres instrucciones usan el modo de registro para el primer operando (el destino) y el modo inmediato para el segundo operando (una constante indicada por el signo #). La segunda instrucción coloca la *dirección* de *A* en R2, no su contenido. Eso es lo que el signo # le dice al ensamblador. Asimismo, la tercera instrucción coloca en R3 la dirección de la primera palabra después del arreglo.

Lo interesante es que el cuerpo del ciclo mismo no contiene ninguna dirección de memoria. En la cuarta instrucción se usa el modo de registro y el indirecto por registro; en la quinta se usa el modo de registro y el inmediato, y en la sexta se usa dos veces el modo de registro.

La instrucción de ramificación BLT podría usar una dirección de memoria, pero lo más probable es que especifique la dirección a la que lleva la rama con una distancia de 8 bits relativa a la instrucción BLT misma. Al evitar totalmente las direcciones de memoria hemos producido un ciclo corto y rápido. Por cierto, este programa es realmente para el Pentium II, sólo que cambiamos los nombres de las instrucciones y los registros, así como la notación, para hacerlo más comprensible, ya que la sintaxis del lenguaje ensamblador estándar del Pentium II (MASM) es rarísima, una reliquia de la anterior vida de la máquina como un 8088.

Vale la pena señalar que, en teoría, hay otra forma de realizar este cálculo sin usar direccionamiento indirecto por registro. El ciclo podría haber contenido una instrucción que sumara *A* y R1, como

```
ADD R1,A
```

Luego, en cada iteración del ciclo la instrucción misma podría incrementarse en 4, de modo que después de una iteración diga

```
ADD R1,A+4
```

y así hasta terminar.

Un programa que se modifica a sí mismo de esta manera es un programa **automodificable**. La idea se le ocurrió al mismísimo John von Neumann y era razonable en las primeras computadoras, que no tenían direccionamiento indirecto por registro. Hoy día los programas automodificables se consideran de muy mal gusto y difíciles de entender; además, no pueden compartirse entre varios procesos al mismo tiempo. Otra desventaja es que ni siquiera funcionan correctamente en máquinas con caché de nivel 1 dividida, si la caché I no cuenta con circuitos para efectuar escrituras de vuelta (porque los diseñadores supusieron que los programas no se modifican ellos mismos).

5.4.6 Direccionamiento indexado

En muchos casos es útil poder hacer referencia a palabras de memoria que están a una distancia conocida de un registro. Ya vimos algunos ejemplos con IJVM en los que se hace referencia a variables locales indicando su distancia respecto a LV. Direccionar la memoria dando un registro (explícito o implícito) más una distancia constante se denomina **direccionamiento indexado**.

El acceso a variables locales en IJVM utiliza un apuntador a la memoria (LV) en un registro más una distancia pequeña en la instrucción misma, como se muestra en la figura 4-19(a). Sin embargo, también es posible hacerlo del otro modo: con el apuntador a la memoria en la instrucción y la distancia pequeña en el registro. Para ver cómo funciona esto, considere el cálculo siguiente. Tenemos dos arreglos unidimensionales *A* y *B*, de 1024 palabras cada uno, y queremos calcular A_i AND B_i para todos los pares y luego obtener el OR de estos 1024 productos booleanos para ver si hay al menos un par distinto de cero en el conjunto. Una estrategia sería colocar la dirección de *A* en un registro, la dirección de *B* en un segundo registro, y luego incrementarlas juntas, de forma análoga a como hicimos en la figura 5-17.

No cabe duda de que esto funcionaría, pero hay una forma mejor de hacerlo, como se muestra en la figura 5-18.

```
        MOV R1,#0       ; acumular el OR en R1, inicialmente 0
        MOV R2,#0       ; R2 = índice, i, del producto actual: A[i] AND B[i]
        MOV R3,#4096    ; R3 = primer valor del índice que no se usará
CICLO   MOV R4,A(R2)    ; R4 = A[i]
        AND R4,B(R2)    ; R4 = A[i] AND B[i]
        OR R1,R4        ; hacer OR de todos los productos booleanos y poner en R1
        ADD R2,#4       ; i = 1 + 4 (incremento en unidades de 1 palabra = 4 bytes)
        CMP R2,R3       ; ¿ya terminamos?
        BLT LOOP        ; si R2 < R3, no hemos terminado; continuar.
```

Figura 5-18. Programa genérico en ensamblador para calcular el OR de A_i y B_i con dos arreglos de 1024 elementos.

El funcionamiento del programa es sencillo. Necesitamos cuatro registros:

1. R1 – Contiene el OR acumulado de los productos booleanos
2. R2 – El índice *i* que se usa para procesar los arreglos
3. R3 – La constante 4096, que es el valor más bajo de *i* que no debemos usar
4. R4 – Un registro de borrador para contener cada producto a medida que se forma.

Después de inicializar los registros ingresamos en el ciclo de seis instrucciones. La instrucción que está en *CICLO* trae A_i y lo coloca en R4. Aquí, el cálculo de la fuente usa el modo indizado. Un registro, R2, y una constante, la dirección de *A*, se suman y la suma se usa para hacer referencia a la memoria. La suma de estas dos cantidades se envía a la memoria, pero no se guarda en ningún registro visible para el usuario. La notación

```
MOV R4,A(R2)
```

indica que el destino usa modo de registro con R4 como el registro y la fuente usa modo indexado, con *A* como la distancia y R2 como el registro. Si *A* tiene el valor de 124300, por ejemplo, la instrucción de máquina real para esto es probable que se parezca a la de la figura 5-19.

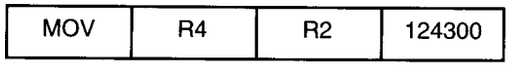


Figura 5-19. Una posible representación de MOV R4,A(R2).

En la primera iteración del ciclo, R2 es 0 (porque así se inicializó), de modo que la palabra de memoria direccionada es A_0, en la dirección 124300. Esta palabra se carga en R4. En la siguiente iteración del ciclo R2 es 4, así que la palabra de memoria a la que se accede es A_1, en 124304, etcétera.

Como prometimos antes, aquí la distancia en la instrucción misma es el apuntador a la memoria, y el valor que está en el registro es un entero pequeño que se incrementa durante el cálculo. Desde luego, esta forma requiere un campo de distancia en la instrucción lo bastante grande como para contener una dirección, así que es menos eficiente que la forma anterior; no obstante, en muchos casos es la mejor manera de hacerlo.

5.4.7 Direccionamiento basado indexado

Algunas máquinas tienen un modo de direccionamiento en el que la dirección en memoria se calcula sumando dos registros más una distancia (opcional). Este modo se conoce como **direccionamiento basado indexado**. Uno de los registros es la base y el otro es el índice. Un modo así habría sido útil aquí. Afuera del ciclo podríamos haber colocado la dirección de *A* en R5 y la dirección de *B* en R6. Luego podríamos haber sustituido la instrucción que está en *CICLO* y su sucesora por

```
CICLO:  MOV R4,(R2+R5)
        AND R4,(R2+R6)
```

Si hubiera un modo de direccionamiento para acceder a la memoria de forma indirecta usando la suma de dos registros sin una distancia, habría sido ideal. Como alternativa, incluso una instrucción con una distancia de 8 bits habría sido mejor que el código original, porque podríamos asignar 0 a ambas distancias. En cambio, si las distancias siempre son de 32 bits no habremos ganado nada al usar este modo. En la práctica, las máquinas que manejan este modo suelen tener una forma con una distancia de 8 bits o de 16 bits.

5.4.8 Direccionamiento de pila

Ya mencionamos que es preferible hacer las instrucciones de máquina lo más pequeñas posible. El límite para reducir la longitud de las direcciones es no tener direcciones. Como vimos en el capítulo 4, son posibles instrucciones con cero direcciones, como IADD, si se usan con una pila. En esta sección examinaremos con mayor detalle el direccionamiento de pila.

Notación polaca inversa

Una tradición antigua en matemáticas es colocar el operador entre los operandos, como en $x+y$, y no después de los operandos, como en $xy+$. La forma en la que el operador está "adentro", entre los operandos, se llama notación **infija**. La forma en la que el operador está después de los operandos se llama notación **posfija** o **notación polaca inversa**, por el lógico polaco J. Lukasiewicz (1958), quien estudió las propiedades de esta notación.

La notación polaca inversa tiene varias ventajas sobre la infija para expresar fórmulas algebraicas. Primera, cualquier fórmula se puede expresar sin paréntesis. Segunda, es cómoda para evaluar fórmulas en computadoras que tienen pilas. Tercera, los operadores infijos tienen precedencia, lo cual es arbitrario e indeseable. Por ejemplo, sabemos que $a \times b + c$ significa $(a \times b) + c$ y no $a \times (b + c)$ porque se definió arbitrariamente que la multiplicación tiene precedencia sobre la suma. Pero, ¿el desplazamiento a la izquierda tiene precedencia sobre el AND booleano? ¿Quién sabe? La notación polaca inversa elimina esta molestia.

Existen varios algoritmos para convertir fórmulas infijas a notación polaca inversa. La que se da a continuación es una adaptación de una idea de E. W. Dijkstra. Suponga que una fórmula se compone de los siguientes símbolos: variables, los operadores díadicos (de dos operandos) + – * /, y los paréntesis izquierdo y derecho. Para marcar el final de una fórmula, insertamos el símbolo ⊥ después del último símbolo y antes del primero.

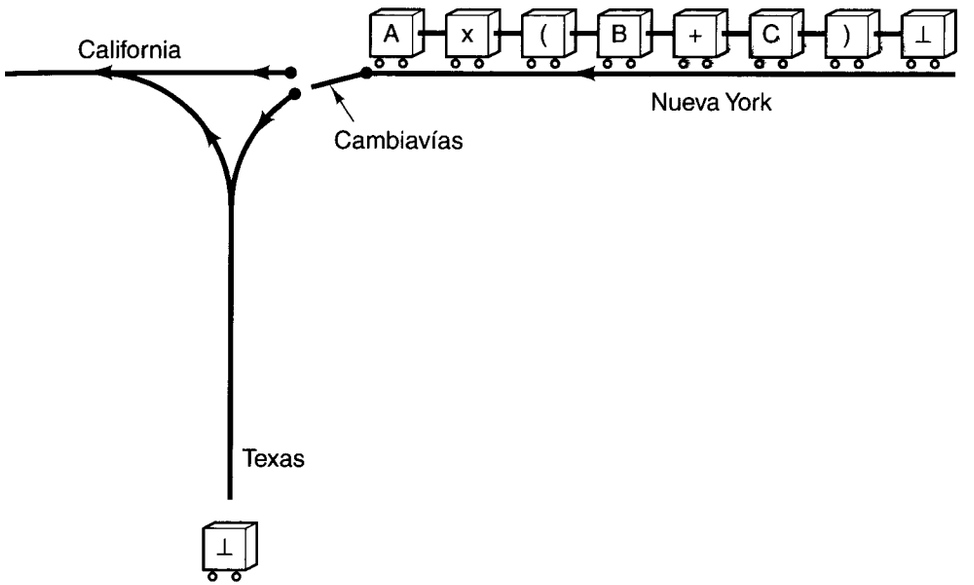


Figura 5-20. Cada vagón de ferrocarril representa un símbolo de la fórmula que se convertirá de notación infija a polaca inversa.

La figura 5-20 muestra una vía de ferrocarril de Nueva York a California, con una desviación a la mitad que lleva a Texas. Cada símbolo de la fórmula se representa con un vagón de ferrocarril. El tren se mueve hacia el poniente (a la izquierda). Cuando un vagón se acerca a un cambiavías, debe detenerse justo antes de llegar y preguntar si debe ir a California directamente o desviarse hacia Texas. Los vagones que contienen variables siempre se van directamente a California sin pasar por Texas. Los vagones que contienen cualquier otro símbolo deben indagar el contenido del vagón más cercano en la vía a Texas antes de ingresar al cambiavías.

La tabla de la figura 5-21 muestra lo que sucede, dependiendo del contenido del vagón más cercano en la vía a Texas y del vagón que está llegando al cambiavías. El primer ⊥ siempre va a Texas. Los números se refieren a las siguientes situaciones:

1. El vagón en el cambiavías se dirige a Texas.
2. El vagón más reciente en la línea a Texas se da vuelta y se dirige a California.
3. Tanto el vagón que está en el cambiavías como el último que tomó la vía a Texas son secuestrados y desaparecen (es decir, ambos se eliminan).
4. Parar. Los símbolos que están en California representan la fórmula en notación polaca inversa leídos de izquierda a derecha.
5. Parar. Ocurrió un error. La fórmula original no estaba balanceada correctamente.

Último vagón que llegó a la vía a Texas \ Vagón en el cambiavías	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Figura 5-21. Tabla de decisión del algoritmo para convertir notación infija a polaca inversa.

Después de realizar cada acción, se hace una nueva comparación entre el vagón que está en el cambiavías, que podría ser el mismo vagón que en la comparación anterior o bien el siguiente vagón, y el vagón que ahora es el último en tomar la vía a Texas. El proceso continúa hasta llegar al paso 4. Observe que la vía a Texas se está usando como pila, en la que la desviación de un vagón a Texas equivale a la operación de meter o apilar (*push*), y hacer que un vagón que ya está en la vía a Texas se dé vuelta y se dirija a California es una operación de sacar o desapilar (*pop*).

Notación infija	Notación polaca inversa
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C − D)	A B + C D − /
A × B / C	A B × C /
((A + B) × C + D) / (E + F = G)	A B + C × D + E F + G + /

Figura 5-22. Ejemplos de expresiones infijas y sus equivalentes en notación polaca inversa.

El orden de las variables es el mismo en la notación infija y en la polaca inversa, pero el orden de los operadores no siempre es el mismo. En la notación polaca inversa los operadores aparecen en el orden en que se ejecutarán realmente durante la evaluación de la expresión. La figura 5-22 da ejemplos de fórmulas infijas y sus equivalentes en notación polaca inversa.

Evaluación de fórmulas en notación polaca inversa

La notación polaca inversa es ideal para evaluar fórmulas en una computadora que tiene pila. La fórmula consiste en *n* símbolos, cada uno de los cuales es un operando o bien un operador. El algoritmo para evaluar una fórmula en notación polaca inversa empleando una pila es sencillo. Se recorre la cadena de notación polaca inversa de izquierda a derecha. Cuando se encuentra un operando, se mete en la pila. Cuando se encuentra un operador, se ejecuta la instrucción correspondiente.

La figura 5-23 muestra la evaluación de

$$(8 + 2 \times 5) / (1 + 3 \times 2 - 4)$$

en JVM. La fórmula en notación polaca inversa es

$$8\ 2\ 5 \times +\ 1\ 3\ 2 \times +\ 4 - /$$

En la figura introdujimos IMUL e IDIV como instrucciones de multiplicación y división, respectivamente. El número que está en el tope de la pila es el operando derecho, no el izquierdo. Este punto es importante para la división (y la resta) porque el orden de los operandos es significativo (a diferencia de la suma y la multiplicación). En otras palabras, IDIV se definió cuidadosamente de modo que si primero se empila el numerador, luego se empila el denominador y luego se efectúa la operación, se obtiene el resultado correcto. Observe lo fácil que es generar código (I)JVM a partir de notación polaca inversa: basta con recorrer la fórmula y producir una instrucción por símbolo. Si el símbolo es una constante o variable, se produce una instrucción que lo almacene en la pila. Si el símbolo es un operador, se produce una instrucción que ejecute la operación.

5.4.9 Modos de direccionamiento para instrucciones de ramificación

Hasta ahora hemos examinado sólo instrucciones que operan con datos. Las instrucciones de ramificación (y las llamadas a procedimientos) también necesitan modos de direccionamiento para especificar la dirección objetivo. Los modos que ya examinamos generalmente también funcionan para las ramificaciones. El direccionamiento directo es una posibilidad obvia: la dirección objetivo simplemente se incluye completa en la instrucción.

Sin embargo, otros modos de direccionamiento también tienen sentido. El direccionamiento indirecto por registro permite al programa calcular la dirección objetivo, colocarla en un registro, y luego saltar ahí. Este modo es el que más flexibilidad ofrece porque la dirección objetivo se calcula en el momento de la ejecución, pero también es la que crea más oportunidades de incluir errores que son casi imposibles de encontrar.

Otro modo razonable es el indexado, que salta una distancia conocida desde un registro. Sus propiedades son las mismas que la del modo indirecto por registro.

Paso	Cadena restante	Instrucción	Pila
1	825 × +132 × +4−/	BIPUSH 8	8
2	25×+132 ×+4−/	BIPUSH 2	8, 2
3	5×+132 ×+4−/	BIPUSH 5	8, 2, 5
4	× +132 ×+4−/	IMUL	8, 10
5	+132 ×+4−/	IADD	18
6	132 ×+4−/	BIPUSH 1	18, 1
7	32× +4−/	BIPUSH 3	18, 1, 3
8	2× +4−/	BIPUSH 2	18, 1, 3, 2
9	×+4−/	IMUL	18, 1, 6
10	+4−/	IADD	18, 7
11	4−/	BIPUSH 4	18, 7, 4
12	−/	ISUB	18, 3
13	/	IDIV	6

Figura 5-23. Uso de una pila para evaluar una fórmula en notación polaca inversa.

Otra opción es el direccionamiento relativo al PC. En este modo, la distancia (con signo) que está en la instrucción se suma al contador de programa para obtener la dirección objetivo. De hecho, esto no es más que el modo indexado usando PC como el registro.

5.4.10 Ortogonalidad de códigos de operación y modos de direccionamiento

Desde el punto de vista del software, las instrucciones y el direccionamiento deben tener una estructura regular, con un mínimo de formatos de instrucciones. Una estructura así permite al compilador producir fácilmente buen código. Todos los códigos de operación deben permitir todos los modos de direccionamiento si son lógicos. Además, todos los registros deberán estar disponibles para todos los modos de registro, incluidos el apuntador de marco (FP), el apuntador de pila (SP) y el contador de programa (PC).

Como ejemplo de diseño aseado para una máquina de tres direcciones, considere los formatos de instrucción de 32 bits de la figura 5-24. Se reconocen hasta 256 códigos de operación. En el formato 1, cada instrucción tiene dos registros fuente y un registro de destino. Todas las instrucciones aritméticas y lógicas usan este formato.

El campo de 8 bits no utilizado del final puede servir para diferenciar aún más las instrucciones. Por ejemplo, podría asignarse un solo código de operación para todas las operaciones de punto flotante, y el campo extra distinguiría entre ellas. Además, si el bit 23 está encendido, se usa el formato 2 y el segundo operando deja de ser un registro para convertirse en una constante inmediata de 13 bits con signo. Las instrucciones LOAD y STORE también pueden usar este formato para hacer referencia a la memoria en modo indexado.

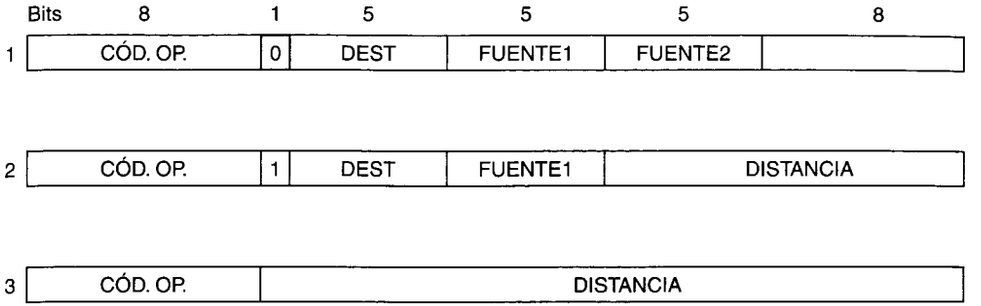


Figura 5-24. Diseño sencillo para los formatos de instrucción de una máquina de tres direcciones.

Se requieren unas pocas instrucciones adicionales, como ramificaciones condicionales, pero éstas podrían usar fácilmente el formato 3. Por ejemplo, se podría asignar un código de operación a cada ramificación (condicional), llamada a procedimiento, etc., lo que dejaría 24 bits para una distancia relativa al PC. Si suponemos que dicha distancia se cuenta en palabras, el alcance sería de ±32 MB. Además, podrían reservarse unos cuantos códigos de operación para las LOAD y STORE que necesitan las distancias largas del formato 3. Éstas no tendrían plena generalidad (por ejemplo, sólo se podría cargar o almacenar R0), pero no se usarían con mucha frecuencia.

Consideremos ahora un diseño para una máquina de dos direcciones que puede usar una palabra de memoria como uno de los operandos. El diseño se muestra en la figura 5-25. Una máquina así puede sumar una palabra de memoria a un registro, sumar un registro a una palabra de memoria, sumar un registro a un registro, o sumar una palabra de memoria a una palabra de memoria. Actualmente, los accesos a la memoria son relativamente costosos, por lo que este diseño no goza de popularidad, pero si los adelantos en las tecnologías de caché o de memoria hacen que los accesos a ésta sean baratos en el futuro, éste es un diseño para el cual es muy fácil y eficiente la compilación. La PDP-11 y la VAX fueron máquinas con mucho éxito que dominaron el mundo de las minicomputadoras durante dos décadas empleando diseños similares a éste.

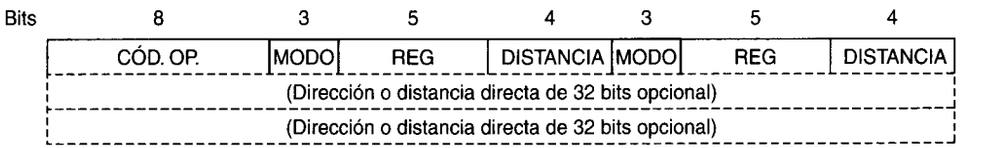


Figura 5-25. Diseño sencillo para los formatos de instrucción de una máquina de dos direcciones

En este diseño, tenemos otra vez un código de operación de 8 bits, pero ahora tenemos 12 bits para especificar la fuente y 12 bits para especificar el destino. Para cada operando, 3 bits dan el modo, 5 bits dan el registro y 4 bits dan la distancia. Con tres bits de modo podríamos manejar direccionamiento inmediato, directo, de registro, indirecto por registro, indexado y de pila, y tener espacio para dos modos futuros más. Éste es un diseño limpio y regular para el cual es fácil

compilar y muy flexible, sobre todo si el contador de programa, el apuntador a la pila y el apuntador a las variables locales están incluidos entre los registros generales y se puede acceder a ellos.

El único problema aquí es que con direccionamiento directo necesitamos más bits para la dirección. Lo que hicieron la PDP-11 y la VAX fue añadir una palabra extra a la instrucción, para la dirección de cada operando direccionado directamente. También podríamos usar uno de los dos modos de direccionamiento disponibles para un modo indexado con una distancia de 32 bits después de la instrucción. Así, en el peor caso, como una ADD de memoria a memoria en la que ambos operandos se direccionan directamente, o usando la forma indexada larga, la instrucción tendría 96 bits de largo y usaría tres ciclos de bus (uno para la instrucción, dos para datos). Por otra parte, casi todos los diseños RISC requerirían al menos 96 bits, y probablemente más, para sumar una palabra de memoria arbitraria a otra, y usaría al menos cuatro ciclos de bus.

Hay muchas posibles alternativas a la figura 5-25. En este diseño es posible ejecutar el enunciado

```
i = j;
```

en una instrucción de 32 bits, siempre que *i* y *j* se cuenten entre las primeras 16 variables locales. Por otra parte, para las variables después de la 16, tenemos que usar distancias de 32 bits. Una opción sería otro formato con una sola distancia de 8 bits en lugar de dos distancias de 4 bits, más una regla que diga que la fuente o el destino podrían usarla, pero no ambos. Las posibilidades y equilibrios son ilimitados, y los diseñadores de máquinas deben balancear muchos factores para obtener un buen resultado.

5.4.11 Modos de direccionamiento del Pentium II

Los modos de direccionamiento del Pentium II son altamente irregulares y difieren dependiendo de si una instrucción dada está en modo de 16 bits o en modo de 32 bits. En lo que sigue haremos caso omiso del modo de 16 bits; el modo de 32 bits ya es bastante malo. Los modos que se manejan incluyen inmediato, directo, de registro, indirecto por registro, indexado y un modo especial para direccionar elementos de un arreglo. El problema es que no todos los modos aplican a todas las instrucciones y no todos los registros se pueden usar en todos los modos. Esto hace que la tarea del escritor de compiladores sea mucho más difícil y que el código generado sea más deficiente.

El byte MODO de la figura 5-13 controla los modos de direccionamiento. Uno de los operandos se especifica con la combinación de los campos MOD y R/M. El otro siempre es un registro, y está dado por el valor del campo REG. Las 32 combinaciones que pueden especificarse con el campo MOD de 2 bits y el campo R/M de 3 bits se enumeran en la figura 5-26. Por ejemplo, si ambos campos son cero, el operando se lee de la dirección de memoria contenida en el registro EAX.

Las columnas 01 y 10 contienen modos en los que un registro se suma a una distancia de 8 o 32 bits que sigue a la instrucción. Si se selecciona una distancia de 8 bits, primero se extiende su signo a 32 bits antes de la suma. Por ejemplo, una instrucción ADD con R/M = 011, MOD = 01 y una distancia de 6 calcula la suma de EBX y 6 y lee la palabra de memoria que está en esa dirección para obtener uno de los operandos. EBX no se modifica.

La columna MOD = 11 permite escoger entre dos registros. Si la instrucción es de palabras, se escoge el primero; si la instrucción es de bytes, se escoge el segundo. Observe que la tabla no es del todo regular. Por ejemplo, no hay forma de direccionar indirectamente a través de EBP ni de sumar una distancia a ESP.

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + DIST8]	M[EAX + DIST32]	EAX o AL
001	M[ECX]	M[ECX + DIST8]	M[ECX + DIST32]	ECX o CL
010	M[EDX]	M[EDX + DIST8]	M[EDX + DIST32]	EDX o DL
011	M[EBX]	M[EBX + DIST8]	M[EBX + DIST32]	EBX o BL
100	SIB	SIB con DIST8	SIB con DIST32	ESP o AH
101	Directo	M[EBP + DIST8]	M[EBP + DIST32]	EBP o CH
110	M[ESI]	M[ESI + DIST8]	M[ESI + DIST32]	ESI o DH
111	M[EDI]	M[EDI + DIST8]	M[EDI + DIST32]	EDI o BH

Figura 5-26. Modos de direccionamiento de 32 bits del Pentium II. M[*x*] es la palabra de memoria que está en *x*.

En algunos modos un byte adicional, llamado **SIB (escala, índice, base)** sigue al byte MODO (véase la figura 5-13). El byte SIB especifica un factor de escala y dos registros. Cuando está presente un byte SIB, la dirección del operando se calcula multiplicando el registro índice por 1, 2, 4 u 8 (dependiendo de ESCALA), sumándolo al registro base, y por último tal vez sumándole un desplazamiento de 8 o 32 bits, dependiendo de MOD. Casi todos los registros pueden usarse como índice o como base.

Los modos SIB son útiles para accesar a elementos de un arreglo. Por ejemplo, considere el enunciado en Java

```
for (i = 0; i < n; i++) a[i] = 0;
```

donde *a* es un arreglo de enteros de 4 bytes local respecto al procedimiento en curso. Por lo regular, se usa EBP para apuntar a la base del marco de pila que contiene las variables locales y arreglos, como se muestra en la figura 5-27. El compilador podría mantener *i* en EAX. Para accesar a *a*[*i*], usaría un modo SIB en el que la dirección del operando es la suma de 4 × EAX, EBP y 8. Esta instrucción podría guardar un valor en *a*[*i*] con una sola instrucción.

¿Vale la pena este modo? Es difícil saberlo. No hay duda de que esta instrucción, si se usa correctamente, ahorra unos cuantos ciclos. La frecuencia con que se use depende del compilador y de la aplicación. El problema es que esta instrucción ocupa un área dentro del chip que podría haber tenido otro uso si la instrucción no estuviera presente. Por ejemplo, la caché nivel 1 podría haber sido más grande, o el chip podría haber sido más pequeño, lo que tal vez habría permitido una velocidad de reloj un poco más alta.

Éstos son los tipos de decisiones que los diseñadores enfrentan todo el tiempo. Por lo regular se efectúan simulaciones extensas antes de producir circuitos en silicio, pero esas simu-

laciones requieren tener una buena idea de la magnitud de la carga de trabajo. Podemos asegurar que los diseñadores del 8088 no incluyeron un navegador de Web en su conjunto de prueba. A pesar de ello, no pocos de los descendientes de ese producto se usan ahora primordialmente para explorar la Web, y es posible que las decisiones tomadas hace 20 años sean totalmente inadecuadas para las aplicaciones actuales. No obstante, en aras de la compatibilidad con modelos anteriores, una vez que se incorpora una característica, es imposible deshacerse de ella.

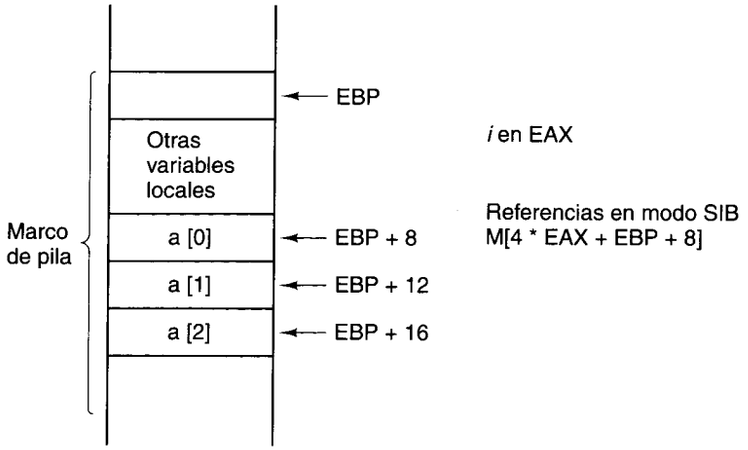


Figura 5-27. Acceso a $a[i]$.

5.4.12 Modos de direccionamiento del UltraSPARC II

En la ISA UltraSPARC todas las instrucciones usan direccionamiento inmediato o de registro con excepción de las que direccionan memoria. En el modo de registro, los 5 bits simplemente indican cuál registro usar. En el modo inmediato, una constante de 13 bits (con signo) proporciona los datos. No hay otros modos para las instrucciones aritméticas, lógicas y similares.

Tres clases de instrucciones direccionan memoria: las LOAD, las STORE y una instrucción de sincronización de multiprocesadores. Las instrucciones LOAD y STORE tienen dos modos para direccionar la memoria. El primer modo calcula la suma de dos registros y luego la usa para acceso indirecto. El otro es una indexación tradicional, con una distancia de 13 bits (con signo).

5.4.13 Modos de direccionamiento de la JVM

La JVM no tiene modos de direccionamiento generales en el sentido de que toda instrucción tiene unos cuantos bits que le indican cómo calcular la dirección (como en el Pentium II). En vez de ello, cada instrucción tiene asociado un modo de direccionamiento específico. Puesto que la JVM no tiene registros visibles, los modos de direccionamiento de registro e indirecto por registro ni siquiera son posibles. Unas cuantas instrucciones, como BIPUSH, usan direccionamiento inmediato. El único otro modo disponible es el modo indizado, empleado por ILOAD, ISTORE, LDC_W y varias otras instrucciones para especificar una variable

relativa a algún registro implícito, por lo regular LV o CPP. Las instrucciones de ramificación también usan el modo indexado, con PC como registro implícito.

5.4.14 Análisis de los modos de direccionamiento

Ya hemos estudiado una buena cantidad de modos de direccionamiento. Los que usan el Pentium II, el UltraSPARC II y JVM se resumen en la figura 5-28. Sin embargo, como ya explicamos, no todas las instrucciones pueden usar todos los modos.

Modo de direccionamiento	Pentium II	UltraSPARC II	JVM
Inmediato	×	×	×
Directo	×		
Registro	×	×	
Indirecto por registro	×		
Indexado	×	×	×
Basado-indexado		×	
Pila			×

Figura 5-28. Comparación de modos de direccionamiento.

En la práctica, no se necesitan muchos modos de direccionamiento para tener una ISA eficaz. Puesto que casi todo el código que se escribe en este nivel es generado por compiladores, el aspecto más importante de los modos de direccionamiento de una arquitectura es que las opciones sean pocas y claras, y que los costos (en términos de tiempo de ejecución y tamaño del código) puedan calcularse con facilidad. Lo que eso implica generalmente es que una máquina debe adoptar una postura extrema: o bien debe ofrecer todas las opciones posibles, o sólo debe ofrecer una. Cualquier término medio implica que el compilador enfrentará decisiones que tal vez no tenga los suficientes conocimientos o sofisticación para tomar.

Así, las arquitecturas más limpias por lo regular sólo tienen un número muy reducido de modos de direccionamiento, con límites estrictos para su uso. En la práctica, tener los modos inmediato, directo, de registro e indexado es suficiente para casi todas las aplicaciones. Además, todos los registros (incluidos el apuntador a variables locales, el apuntador a la pila y el contador de programa) deberán poder usarse cuando se requiera un registro. Los modos de direccionamiento más complejos tal vez reduzcan el número de instrucciones, pero a expensas de introducir secuencias de operaciones que no se pueden ejecutar fácilmente en paralelo con otras operaciones secuenciales.

Ya terminamos nuestro estudio de los diversos equilibrios que puede haber entre códigos de operación y direcciones, y las distintas formas de direccionamiento. Al estudiar una computadora nueva, es recomendable examinar las instrucciones y los modos de direccionamiento no sólo para ver cuáles están disponibles, sino también para entender por qué se tomaron esas decisiones y qué consecuencias habrían tenido otras decisiones.

5.5 TIPOS DE INSTRUCCIONES

Las instrucciones en el nivel ISA pueden dividirse aproximadamente en una media docena de grupos que son relativamente similares entre las distintas máquinas, aunque podrían diferir en los detalles. Además, toda computadora tiene unas cuantas instrucciones poco comunes, que se añadieron para mantener la compatibilidad con modelos previos o porque el arquitecto tuvo una idea brillante o tal vez porque una dependencia del gobierno le pagó al fabricante para que la incluyera. Trataremos de cubrir brevemente todas las categorías comunes a continuación, sin tratar de ser exhaustivos.

5.5.1 Instrucciones de movimiento de datos

Copiar datos de un lugar a otro es la más fundamental de todas las operaciones. Con copiar queremos decir crear un nuevo objeto con un patrón de bits idéntico al del original. Este uso de la palabra "movimiento" es un tanto diferente de su uso normal en español. Cuando decimos que las 14 toneladas de maíz que estaban en la Bodega 23 se movieron al molino, no queremos implicar que se creó una copia idéntica del maíz en el molino y que las 14 toneladas originales todavía están en la Bodega 23. Cuando decimos que el contenido de la localidad de memoria 2000 se trasladó o pasó a algún registro, siempre queremos decir que se creó en el registro una copia idéntica y que el original todavía está en la localidad 2000. Sería mejor llamar a las instrucciones de movimiento de datos instrucciones de "duplicación de datos", pero el término "traslado o movimiento de datos" ya está establecido.

Hay dos razones para copiar datos de un lugar a otro. Una es fundamental: la asignación de valores a variables. La asignación

A = B

se implementa copiando el valor que está en la dirección de memoria *B* en la posición *A* porque el programador dijo que se hiciera. La segunda razón para copiar datos es prepararlos para un acceso y uso eficientes. Como hemos visto, muchas instrucciones sólo pueden accesar a variables si están disponibles en un registro. Puesto que hay dos posibles fuentes para un dato (memoria o registro), y hay dos posibles destinos para un dato (memoria o registro), hay cuatro posibles tipos de copiado. Algunas computadoras tienen cuatro instrucciones, una para cada caso; otras tienen una instrucción para los cuatro casos. Otras más usan LOAD para copiar de la memoria a un registro, STORE para copiar de un registro a la memoria, MOVE para copiar de un registro a otro, y ninguna para copiar de una localidad de memoria a otra.

Las instrucciones de movimiento de datos deben indicar de alguna manera la cantidad de datos que deben copiarse. En algunas ISA existen instrucciones para mover cantidades variables de datos que van desde un bit hasta toda la memoria. En las máquinas con longitud de palabra fija la cantidad que se traslada suele ser exactamente una palabra. Para mover una cantidad mayor o menor se requiere una rutina en software que usa desplazamiento y fusión. Algunas ISA ofrecen capacidades adicionales para copiar menos de una palabra (generalmente en incrementos de bytes) y también varias palabras. Copiar múltiples palabras no es

fácil, sobre todo si el número máximo de palabras es grande, porque una operación así puede tardar mucho, y podría tenerse que interrumpir a la mitad. Algunas máquinas con longitud de palabra variable tienen instrucciones que especifican sólo las direcciones de origen y de destino pero no la cantidad. El movimiento continúa hasta que se encuentra un campo de fin de datos.

5.5.2 Operaciones diádicas

Las operaciones diádicas son las que combinan dos operandos para producir un resultado. Todas las ISA tienen instrucciones para sumar y restar enteros. La multiplicación y división de enteros también son estándar en muchos casos. Suponemos que no es necesario explicar por qué las computadoras están equipadas con instrucciones aritméticas.

Otro grupo de operaciones diádicas incluye las instrucciones booleanas. Aunque existen 16 funciones booleanas de dos variables, casi ninguna máquina, o ninguna, tiene instrucciones para las 16. Por lo regular están presentes AND, OR y NOT; a veces también se incluyen el OR EXCLUSIVO, el NOR y el NAND.

Un uso importante de AND es la extracción de bits de una palabra. Consideremos, por ejemplo, una máquina con palabras de 32 bits en la que se almacenan cuatro caracteres de 8 bits en cada palabra. Supongamos que es necesario separar el segundo carácter de los otros tres para imprimirlo; es decir, es necesario crear una palabra que contenga ese carácter en los 8 bits de la derecha (es decir, que esté **justificado a la derecha**), con ceros en los 24 bits de la izquierda.

Para extraer el carácter, se calcula el AND de la palabra que lo contiene y una constante, llamada **máscara**. El resultado de esta operación es que los bits no deseados se convierten en ceros –es decir, se enmascaran– como se muestra a continuación.

```
10110111 10111100 11011011 10001011  A
00000000 11111111 00000000 00000000  B (máscara)
00000000 10111100 00000000 00000000  A AND B
```

Después, el resultado se desplazaría 16 bits a la derecha para aislar el carácter en el extremo derecho de la palabra.

Un uso importante de OR es empaquetar bits en una palabra, donde "empaquetar" es lo contrario de "extraer". Si queremos modificar los 8 bits de la derecha de una palabra de 32 bits sin alterar los otros 24 bits, primero enmascaramos los 8 bits no deseados y luego introducimos el nuevo carácter con un OR, como se muestra a continuación.

```
10110111 10111100 11011011 10001011  A
11111111 11111111 11111111 00000000  B (máscara)
10110111 10111100 11011011 00000000  A AND B
00000000 00000000 00000000 01010111  C
10110111 10111100 11011011 01010111  (A AND B) OR C
```

La operación AND tiende a eliminar unos, porque nunca hay más unos en el resultado que en cualquiera de los operandos. La operación OR tiende a insertar unos, porque siempre hay al menos tantos unos en el resultado como en el operando que más unos tiene. El OR

EXCLUSIVO, en cambio, es simétrico, y en promedio no tiende a insertar ni eliminar unos. Esta simetría respecto a los unos y ceros a veces es útil, por ejemplo al generar números aleatorios.

Casi todas las computadoras actuales también manejan una serie de instrucciones de punto flotante, que corresponden aproximadamente a las operaciones aritméticas con enteros. Casi todas las máquinas ofrecen al menos dos tamaños de números de punto flotante, los más cortos para efectuar operaciones rápidas y los más largos para aquellas ocasiones en que se necesitan muchos dígitos de precisión. Si bien hay muchas variaciones de formatos de punto flotante posibles, se ha adoptado casi universalmente una sola norma: IEEE 754. Los números de punto flotante y la norma se describen en el Apéndice B.

5.5.3 Operaciones monádicas

Las operaciones monádicas tienen un operando y producen un resultado. Dado que se tiene que especificar una dirección menos que en las operaciones diádicas, las instrucciones a veces son más cortas, aunque en muchos casos también hay que especificar otra información.

Las instrucciones para desplazar o someter a rotación el contenido de una palabra o byte son muy útiles y a menudo se incluyen con variaciones. Los desplazamientos son operaciones en las que los bits se recorren hacia la izquierda o la derecha, y los bits que son expulsados por un extremo se pierden. Las rotaciones son desplazamientos en los que los bits que salen por un extremo se retroalimentan en el otro extremo. A continuación ilustramos la diferencia entre un desplazamiento y una rotación.

```
00000000 00000000 00000000 01110011  A
00000000 00000000 00000000 00011100  A desplazado 2 bits a la derecha
11000000 00000000 00000000 00011100  A girado 2 bits a la derecha
```

Son útiles los desplazamientos y rotaciones tanto a la derecha como a la izquierda. Si una palabra de n bits se gira k bits a la izquierda, el resultado es el mismo que si se hubiera girado a la derecha $n - k$ bits.

Los desplazamientos a la derecha a menudo se realizan con extensión del signo. Esto implica que las posiciones que se desocupan en el extremo izquierdo de la palabra se llenan con el bit de signo original, 0 o 1. Es como si el bit de signo se arrastrara a la derecha. Entre otras cosas, esto implica que un número negativo seguirá siendo negativo. A continuación ilustramos esta situación para desplazamientos de 2 bits a la derecha.

```
11111111 11111111 11111111 11110000  A
00111111 11111111 11111111 11111100  A desplazado sin extensión de signo
11111111 11111111 11111111 11111100  A desplazado con extensión de signo
```

Un uso importante del desplazamiento es la multiplicación y división por potencias de 2. Si un entero positivo se desplaza k bits a la izquierda, el resultado, siempre que no haya habido desbordamiento, es el número original multiplicado por 2^k. Si un entero positivo se desplaza a la derecha k bits, el resultado es el número original dividido entre 2^k.

Los desplazamientos pueden servir para agilizar ciertas operaciones aritméticas. Consideremos, por ejemplo, el cálculo de $18 \times n$ para algún entero positivo n. Puesto que $18 \times n = 16 \times n + 2 \times n$, podemos obtener $16 \times n$ desplazando 4 bits a la izquierda una copia de n. Podemos obtener $2 \times n$ desplazando n un bit a la izquierda. La suma de estos dos números es $18 \times n$. Realizamos la multiplicación con un movimiento, dos desplazamiento y una suma, lo cual muchas veces es más rápido que una multiplicación. Desde luego, el compilador sólo puede usar este truco cuando un factor es una constante.

El desplazamiento de números negativos, aun con extensión de signo, da resultados muy distintos. Consideremos, por ejemplo, el número –1 en complemento a unos. Si lo desplazamos un bit a la izquierda obtenemos –3. Otro desplazamiento de un bit a la izquierda produce –7:

```
11111111 11111111 11111111 11111110  –1 en complemento a unos
11111111 11111111 11111111 11111100  –1 desplazado un bit a la izquierda = –3
11111111 11111111 11111111 11111000  –1 desplazado dos bits a la izquierda = –7
```

El desplazamiento a la izquierda de números negativos en formato de complemento a unos no multiplica por 2. Sin embargo, el desplazamiento a la derecha sí simula la división correctamente.

Consideremos ahora una representación de –1 en formato de complemento a dos. Si la desplazamos 6 bits a la derecha con extensión de signo, obtenemos –1, lo cual es incorrecto porque la parte entera de –1/64 es 0:

```
11111111 11111111 11111111 11111111  –1 en complemento a dos
11111111 11111111 11111111 11111111  –1 desplazado 6 bits a la derecha = –1
```

En general, el desplazamiento a la derecha introduce errores porque trunca hacia abajo (hacia el entero más negativo), lo que es incorrecto en la aritmética de números enteros negativos. En cambio, el desplazamiento a la izquierda sí simula la multiplicación por 2.

Las operaciones de rotación son útiles para empacar y desempacar secuencias de bits de las palabras. Si queremos probar todos los bits de una palabra, girar la palabra de bit en bit en cualquier dirección coloca sucesivamente cada bit en el bit de signo, donde se le puede probar fácilmente, y también restaura la palabra a su valor original una vez que se han probado todos los bits. Las operaciones de rotación son más puras que las de desplazamiento porque no se pierde información: una operación de rotación arbitraria se puede anular con otra operación de rotación.

Ciertas operaciones diádicas ocurren tan a menudo con ciertos operandos que las ISA a veces cuentan con instrucciones monádicas para efectuarlas rápidamente. Poner cero en una palabra de memoria o un registro es muy común al inicializar un cálculo. Desde luego, mover un cero es un caso especial de las instrucciones generales de movimiento de datos. Por eficiencia, es común que se incluya una operación CLR (*Clear*, borrar), con una sola dirección, la posición que se quiere borrar (es decir, igualar a cero).

Al contar también es común sumar 1 a una palabra. Una forma monádica de la instrucción ADD es la operación INC, que suma 1. La operación NEG es otro ejemplo. Negar X en realidad es calcular $0 - X$, una resta diádica, pero por la frecuencia con que se usa a veces se proporciona una instrucción NEG aparte. Es importante señalar aquí la diferencia entre la operación aritmética NEG y la operación lógica NOT. La operación NEG produce el **inverso aditivo** de un número (el número que sumado al original da 0). La operación NOT simple-

mente invierte todos los bits individuales de la palabra. Las operaciones son muy similares y de hecho, en un sistema que usa la representación de complemento a uno, son idénticas. (En aritmética de complemento a dos la instrucción NEG se lleva a cabo invirtiendo primero todos los bits individuales y sumando luego 1.)

Las instrucciones diádicas y monádicas a menudo se agrupan por su uso, más que por el número de operandos que requieren. Un grupo abarca las operaciones aritméticas, incluida la negación. El otro incluye las operaciones lógicas y el desplazamiento, ya que estas categorías casi siempre se usan juntas para obtener datos.

5.5.4 Comparaciones y ramificaciones condicionales

Casi todos los programas requieren la capacidad para probar sus datos y alterar la secuencia de instrucciones a ejecutar con base en el resultado. Un ejemplo sencillo es la función raíz cuadrada, $\sqrt{x}$. Si x es negativo, el procedimiento produce un mensaje de error; si no, calcula la raíz cuadrada. Una función *sqrt* tiene que probar x y luego ramificar, dependiendo de si es negativo o no.

Un método común para hacerlo es proporcionar instrucciones de ramificación condicional que prueban alguna condición y saltan a una dirección de memoria dada si la condición se satisface. A veces un bit de la instrucción indica si la rama debe tomarse si la condición se cumple o si no se cumple, respectivamente. En muchos casos la dirección objetivo no es absoluta, sino relativa a la instrucción actual.

La condición que con más frecuencia se prueba es si un bit dado de la máquina es 0 o no. Si una instrucción prueba el bit de signo de un número y salta a *ETIQUETA* si es 1, los enunciados que comienzan en *ETIQUETA* se ejecutarán si el número es negativo, y los enunciados que siguen a la ramificación condicional se ejecutarán si es 0 o positivo.

Muchas máquinas tienen bits de código de condición que sirven para indicar condiciones específicas. Por ejemplo, podría haber un bit de desbordamiento que se pone en 1 cada vez que una operación aritmética produce un resultado incorrecto. Al probar este bit se verifica si hubo desbordamiento o no en la operación aritmética anterior, y si ocurrió podría tomarse una rama a una rutina de error.

Asimismo, algunos procesadores tienen un bit de acarreo que se enciende cuando hay un acarreo de salida del bit de la extrema izquierda; por ejemplo, cuando se suman dos números negativos. Un acarreo de salida del bit de la extrema izquierda es muy normal y no debe confundirse con un desbordamiento. Se requiere la prueba del bit de acarreo en la aritmética de precisión múltiple (es decir, cuando un entero se representa con dos o más palabras).

La prueba de cero es importante en ciclos y muchas otras cosas. Si todas las instrucciones de ramificación condicional probaran sólo un bit, para probar si una palabra es 0 necesitaríamos una prueba individual de cada bit, para constatar que ninguno es 1. A fin de evitar esta situación, muchas máquinas cuentan con una instrucción que prueba una palabra y toma la rama si es cero. Desde luego, esta solución simplemente pasa la responsabilidad a la microarquitectura. En la práctica, el hardware casi siempre tiene un registro cuyos bits se conectan todos a una compuerta OR, para dar un solo bit que indica si el registro contiene

algún bit 1. El bit *Z* de la figura 4-1 normalmente se calcularía haciendo OR con todos los bits de salida de la ALU e invirtiendo después el resultado.

También es importante comparar dos palabras o caracteres para ver si son iguales o, si no lo son, cuál es mayor. Un ejemplo de aplicación son los ordenamientos. Para realizar esta prueba se requieren tres direcciones: dos para los datos y una para la dirección a la que debe saltarse si la condición se cumple. Las computadoras cuyo formato de instrucciones permite tres direcciones por instrucción no tienen problemas, pero las otras deben hacer algo para superar esta deficiencia.

Una solución común es proporcionar una instrucción que realice una comparación y establezca uno o más bits de condición para registrar el resultado. Una instrucción subsecuente puede probar los bits de condición y ramificar si los dos valores comparados son iguales, o diferentes, o si el primero fue mayor, etc. Tanto el Pentium II como el UltraSPARC II adoptan este enfoque.

La comparación de dos números tiene algunos aspectos sutiles. Por ejemplo, la comparación no es tan sencilla como la resta. Si se compara un número positivo muy grande con un número negativo muy grande, la resta producirá un desbordamiento, porque el resultado de la resta no puede representarse. No obstante, la instrucción de comparación debe determinar si la prueba especificada se cumple o no, y devolver la respuesta correcta; no hay desbordamiento en las comparaciones.

Otra cuestión sutil relacionada con la comparación de números es decidir si se debe considerar que los números tienen signo o no. Los números binarios de tres bits pueden ordenarse de dos formas. De menor a mayor:

Sin signo	Con signo
000	100 (menor)
001	101
010	110
011	111
100	000
101	001
110	010
111	011 (mayor)

La columna de la izquierda muestra los enteros positivos del 0 al 7 en orden creciente. La columna de la derecha muestra los enteros con signo –4 a +3 en complemento a dos. La respuesta a la pregunta "¿Es 011 mayor que 100?" depende de si los números se consideran con signo o no. Casi todas las ISA cuentan con instrucciones que manejan ambos ordenamientos.

5.5.5 Instrucciones de llamada a procedimientos

Un procedimiento es un grupo de instrucciones que realiza alguna tarea y que puede invocarse (llamarse) desde varios puntos del programa. Es común usar el término **subrutina** en lugar

de procedimiento, sobre todo al referirse a programas en lenguaje ensamblador. En Java, el término que se usa es **método**. Una vez que el procedimiento ha terminado su trabajo, debe regresar al enunciado que sigue a la llamada. Por tanto, la dirección de retorno se debe transmitir al procedimiento o guardarse en algún lugar donde se le pueda encontrar cuando sea el momento de regresar.

La dirección de retorno se puede colocar en cualquiera de tres lugares: la memoria, un registro o la pila. Por mucho, la peor solución es colocarla en una sola localidad fija de memoria. Con esta técnica, si el procedimiento invoca a otro procedimiento, la segunda llamada hará que la dirección de retorno de la primera se pierda.

Una solución un poco mejor es hacer que la instrucción de llamada a procedimiento guarde la dirección de retorno en la primera palabra del procedimiento, y que la primera dirección ejecutable esté en la segunda palabra. Así, el procedimiento puede regresar tomando una rama indirecta a la primera palabra o, si el hardware coloca el código de operación de ramificación en la primera palabra junto con la dirección de retorno, tomando una rama directamente a ella. El procedimiento podría invocar otros procedimientos, porque cada procedimiento tiene espacio para una dirección de retorno. Si el procedimiento se invoca a sí mismo, este esquema falla, porque la primera dirección de retorno será destruida por la segunda llamada. La capacidad de un procedimiento para invocarse a sí mismo, llamada **recursión**, es en extremo importante tanto para los teóricos como para los programadores prácticos. Además, si el procedimiento *A* llama al procedimiento *B*, y *B* llama al procedimiento *C*, y *C* llama a *A* (recursión indirecta o encadenada), el esquema también falla.

Algo todavía mejor es hacer que la instrucción de llamada a procedimiento coloque la dirección de retorno en un registro, dejando la responsabilidad de guardarla en un lugar seguro al procedimiento. Si el procedimiento es recursivo, tendrá que poner la dirección de retorno en un lugar distinto cada vez que se invoque.

Lo mejor que puede hacer la instrucción de llamada a procedimiento con la dirección de retorno es almacenarla en una pila. Una vez que el procedimiento termina, saca la dirección de retorno de la pila y la coloca en el contador de programa. Si se cuenta con esta forma de llamada a procedimiento, la recursión no causa problemas especiales; la dirección de retorno se guarda automáticamente de una forma tal que se evita la destrucción de direcciones de retorno previas. Vimos esta forma de guardar la dirección de retorno en IJVM en la figura 4-12.

5.5.6 Control de ciclos

La necesidad de ejecutar un grupo de instrucciones un número fijo de veces surge con frecuencia y es por ello que algunas máquinas cuentan con instrucciones para facilitar esto. Todos los esquemas usan un contador que se incrementa o decrementa en alguna constante en cada iteración del ciclo. El contador también se prueba una vez en cada iteración. Si cierta condición se cumple, el ciclo termina.

Un método inicializa un contador afuera del ciclo y de inmediato comienza a ejecutar el código del ciclo. La última instrucción del ciclo actualiza el contador y, si la condición de

terminación todavía no se satisface, salta a la primera instrucción del ciclo. De lo contrario, el ciclo termina y se ejecuta la primera instrucción que sigue al ciclo. Esta forma de ciclo se describe como ciclo con prueba al final y se ilustra en C en la figura 5-29(a). (No podríamos usar Java aquí porque no tiene una instrucción goto.)

```
    i = 1;

L1: primer-enunciado;
    .
    .
    .
    último-enunciado;
    i = i + 1;
    if (i < n) goto L1;
```

(a)

```
    i = 1;
L1: if (i > n) goto L2;
    primer-enunciado;
    .
    .
    .
    último-enunciado
    i = i + 1;
    goto L1;
L2:
```

(b)

Figura 5-29. (a) Ciclo con prueba al final. (b) Ciclo con prueba al principio.

Los ciclos con prueba al final tienen la propiedad de que el ciclo siempre se ejecuta al menos una vez, incluso si n es 0 o menor que 0. Considere, por ejemplo, un programa que mantiene los expedientes de personal de una compañía. En cierto punto del programa, se lee información acerca de un empleado en particular. Se lee n, el número de hijos que el empleado tiene, y se ejecuta un ciclo n veces, una por cada hijo, leyendo el nombre del hijo, su sexo y su fecha de nacimiento para que la compañía pueda enviarle un regalo de cumpleaños, lo cual es una de las prestaciones de la compañía. Si el empleado no tiene hijos, n será 0 pero el ciclo se ejecutará una vez y dará resultados erróneos.

La figura 5-29(b) muestra otra forma de realizar la prueba que funciona correctamente incluso cuando n es 0 o menor que 0. Observe que la prueba es diferente en los dos casos, de modo que si una sola instrucción ISA realiza tanto el incremento como la prueba, los diseñadores se verán obligados a escoger un método o el otro.

Considere el código que se debe producir para el enunciado

```
for (i = 0; i < n; i++) {enunciados}
```

Si el compilador no tiene información acerca de n, deberá usar la estrategia de la figura 5-29(b) para manejar correctamente el caso en que $n \leq 0$. En cambio, si puede determinar que $n > 0$, digamos examinando la asignación a n, podría usar el código de la figura 5-29(a), que es mejor. El estándar de FORTRAN solía exigir que todos los ciclos se ejecutaran al menos una vez, a fin de poder generar siempre el código más eficiente de la figura 5-29(a). En 1977 ese defecto se corrigió cuando hasta la comunidad FORTRAN comenzó a darse cuenta de que tener un enunciado de ciclo con semántica extraña que a veces daba la respuesta correcta no era una buena idea, aunque ahorrara una instrucción de ramificación en cada ciclo. C y Java siempre lo han hecho correctamente.

5.5.7 Entrada/Salida

Ningún otro grupo de instrucciones exhibe tanta variedad de una máquina a otra como las instrucciones de E/S. En las computadoras personales actuales se usan tres esquemas de E/S distintos. Éstos son

1. E/S programada con espera activa.
2. E/S controlada por interrupciones.
3. E/S de acceso directo a memoria (DMA).

A continuación examinaremos los tres esquemas por turno.

El método de E/S más sencillo posible es la **E/S programada**, que suele usarse en los microprocesadores de bajo costo, como los sistemas incorporados, o en los sistemas que deben responder rápidamente a cambios externos (sistemas de tiempo real). Estas CPU por lo regular tienen una sola instrucción de entrada y una sola instrucción de salida. Cada una de estas instrucciones selecciona uno de los dispositivos de E/S. Se transfiere un solo carácter entre un registro fijo del procesador y el dispositivo de E/S seleccionado. El procesador debe ejecutar una sucesión explícita de instrucciones para cada uno de los caracteres leídos o escritos.

Como ejemplo sencillo de este método, considere una terminal con cuatro registros de un byte, como se muestra en la figura 5-30. Se usan dos registros para entrada, estado y datos, y dos para salida, estado y datos. Cada uno tiene una dirección única. Si se usa E/S con correspondencia en la memoria, los cuatro registros forman parte del espacio de direcciones de memoria de la computadora y se pueden leer y escribir usando instrucciones ordinarias. Otra posibilidad es contar con instrucciones de E/S especiales, digamos IN y OUT, para leerlos y escribir en ellos. En ambos casos, la E/S se efectúa transfiriendo datos e información de situación entre la CPU y estos registros.

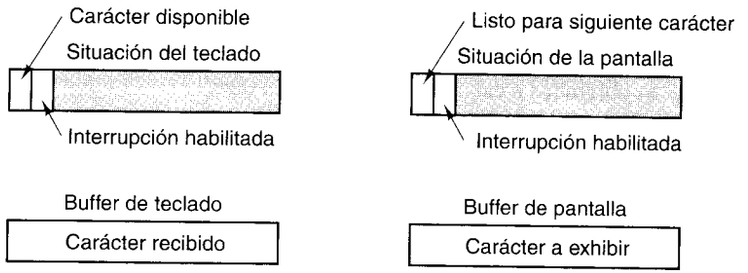


Figura 5-30. Registros de dispositivo para una terminal sencilla.

El registro de estado del teclado tiene dos bits que se usan y seis que no. El hardware pone en 1 el bit de la izquierda (7) cada vez que llega un carácter. Si antes el software encendió el bit 6, se genera una interrupción; si no, no (hablaremos de las interrupciones en breve). Al usar E/S programada para obtener entradas, la CPU normalmente está dando vueltas en un ciclo corto leyendo una y otra vez el registro de estado del teclado, esperando que se encienda

el bit 7. Cuando esto sucede, el software lee el registro buffer del teclado para obtener el carácter. La lectura del registro de datos del teclado hace que el bit CARÁCTER DISPONIBLE se ponga en 0.

La salida funciona de forma parecida. Para escribir un carácter en la pantalla, el software primero lee el registro de estado de la pantalla para ver si el bit READY vale 1. Si no, da vueltas en un ciclo hasta que el bit se enciende, lo que indica que el dispositivo está listo para aceptar un carácter. Tan pronto como la terminal está lista, el software escribe un carácter en el registro buffer de pantalla, lo que hace que se transmita a la pantalla, y también hace que el dispositivo apague el bit READY del registro de situación de la pantalla. Una vez que el carácter se exhibe y la terminal está lista para manejar el siguiente carácter, el controlador enciende automáticamente el bit READY.

Como ejemplo de E/S programada, considere el procedimiento Java de la figura 5-31. Este procedimiento se invoca con dos parámetros: un arreglo de caracteres que se enviará a la salida (buf[]) y el número de caracteres presentes en el arreglo (count), hasta 1K. El cuerpo del procedimiento es un ciclo que envía caracteres a la salida uno por uno. Para cada carácter, primero la CPU debe esperar hasta que el dispositivo está listo, y luego se envía el carácter a la salida. Los procedimientos *in* y *out* normalmente serían rutinas en lenguaje ensamblador para leer y escribir los registros de dispositivo especificados por el primer parámetro, de o en la variable especificada como segundo parámetro. La división entre 128 se deshace de los siete bits de orden bajo y deja el bit READY en el bit 0.

```
public static void output-buffer(int buf[ ], int count) {

  // Enviar un bloque de datos al dispositivo
  int status, i, ready;
  for (i = 0; i < count; i++) {
    do {
      status = in(display-status-reg);        // obtener situación
      ready = (status << 7) & 0x01;           // aislar bit ready
    } while (ready == 1);
    out(display_buffer_reg, buf[i];
  }
}
```

Figura 5-31. Ejemplo de E/S programada.

La principal desventaja de la E/S programada es que la CPU pasa casi todo su tiempo en un ciclo corto esperando que el dispositivo esté listo. Este enfoque se llama **espera activa**. Si la CPU no tiene nada más que hacer (por ejemplo, la CPU de una lavadora de ropa), la espera activa podría ser lo mejor (aunque incluso un controlador sencillo a menudo necesita monitorear varios sucesos concurrentes). Pero si hay otras cosas que hacer, como ejecutar otros programas, la espera activa es un desperdicio de recursos y se necesita un método de E/S distinto.

La forma de deshacerse de la espera activa es hacer que la CPU inicie el dispositivo de E/S y le diga que genere una interrupción cuando haya terminado. Mostraremos cómo se

hace esto con referencia a la figura 5-30. Si el software habilita el bit HABILITAR INTERRUPCIONES de un registro de dispositivo, el software puede solicitar que el hardware le envíe una señal cuando la E/S haya terminado. Estudiaremos las interrupciones con detalle en una sección posterior del capítulo cuando lleguemos al tema de control de flujo.

Vale la pena mencionar que en muchas computadoras la señal de interrupción se genera obteniendo el AND del bit HABILITAR INTERRUPCIONES y el bit READY. Si el software habilita las interrupciones primero (antes de iniciar E/S), ocurrirá una interrupción de inmediato, porque el bit READY será 1. Por ello, podría ser necesario iniciar primero el dispositivo e inmediatamente después habilitar las interrupciones. La escritura de un byte en el registro de situación no altera el bit READY, pues es sólo de lectura.

Aunque la E/S controlada por interrupciones es un gran adelanto en comparación con la E/S programada, le falta mucho para ser perfecta. El problema es que se requiere una interrupción por cada carácter transmitido. El procesamiento de interrupciones es costoso. Necesitamos una forma de deshacernos de la mayor parte de las interrupciones.

La solución está en regresar a la E/S programada, pero encargársela a alguien más. (La solución de muchos problemas consiste en hacer que otra persona efectúe el trabajo.) La figura 5-32 muestra cómo se logra esto. Hemos añadido un nuevo chip al sistema, un controlador de **acceso directo a la memoria** (**DMA**, *Direct Memory Access*), con acceso directo al bus.

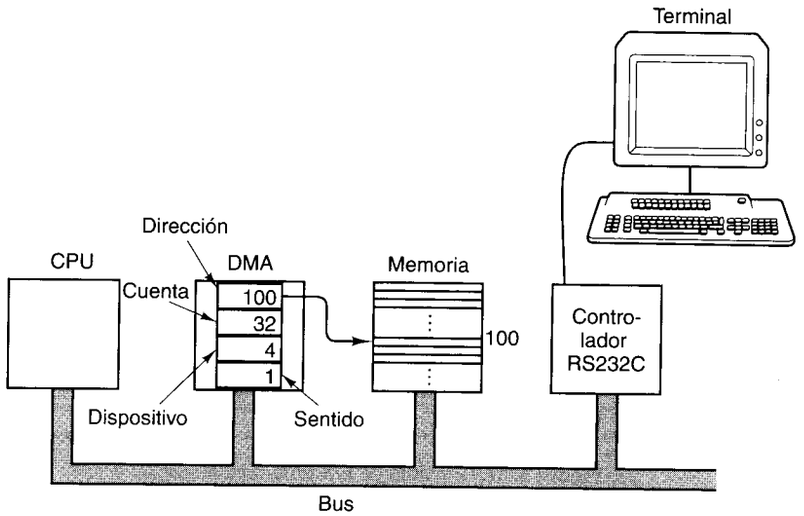


Figura 5-32. Sistema con controlador DMA.

El chip DMA contiene (al menos) cuatro registros, todos los cuales pueden ser cargados por software que se ejecuta en la CPU. El primero contiene la dirección de memoria que se leerá o en la que se escribirá. El segundo contiene el número de bytes (o palabras) que se transferirán. El tercero especifica el número de dispositivo o espacio de direcciones de E/S que se usará, con lo que se especifica qué dispositivo de E/S se desea. El cuarto indica si los datos se escribirán en el dispositivo de E/S o se leerán de él.

Para escribir un bloque de 32 bytes de la dirección de memoria 100 en una terminal (digamos, el dispositivo 4), la CPU escribe los números 100, 32 y 4 en los primeros tres registros de DMA, y luego el código de escribir (digamos, 1) en el cuarto, como se muestra en la figura 5-32. Una vez inicializado con estos valores, el controlador de DMA emite una solicitud de bus para leer el byte 100 de la memoria, del mismo modo como la CPU leería de la memoria. Una vez que obtiene el byte, el controlador de DMA emite una solicitud de E/S al dispositivo 4, para escribir el byte en él. Una vez completadas ambas operaciones, el controlador de DMA incrementa en 1 su registro de dirección y decrementa en 1 su registro de cuenta. Si el registro de cuenta sigue siendo mayor que 0, se lee otro byte de la memoria y se escribe en el dispositivo.

Cuando la cuenta por fin llega a 0, el controlador de DMA deja de transferir data y habilita la línea de interrupción en el chip de CPU. Con DMA, la CPU sólo tiene que inicializar unos cuantos registros. Una vez hecho eso, queda libre para hacer otras cosas hasta que se completa la transferencia, momento en el cual recibe una interrupción del controlador de DMA. Algunos controladores de DMA tienen dos, tres o más conjuntos de registros, lo que les permite controlar varias transferencias simultáneas.

Si bien el DMA alivia a la CPU de gran parte de la carga de E/S, el proceso no es totalmente gratuito. Si un dispositivo de alta velocidad, como un disco, se está controlando por DMA, se requerirán muchos ciclos de bus, tanto para las referencias a la memoria como para las referencias al dispositivo. Durante estos ciclos la CPU tendrá que esperar (el DMA siempre tiene precedencia sobre la CPU para usar el bus porque muchos dispositivos de E/S no pueden tolerar retrasos). El proceso por el que el controlador de DMA quita ciclos de bus a la CPU se denomina **robo de ciclos**. No obstante, la ganancia por no tener que manejar una interrupción por byte (o palabra) transferida rebasa por mucho la pérdida debida al robo de ciclos.

5.5.8 Las instrucciones del Pentium II

En esta sección y las dos que siguen examinaremos los conjuntos de instrucciones de nuestros tres ejemplos de máquinas, el Pentium II, el UltraSPARC II y el picoJava II. Cada una tiene un núcleo de instrucciones que los compiladores normalmente generan, más un grupo de instrucciones que pocas veces se usan, o que sólo el sistema operativo usa. En nuestra explicación nos concentraremos en las instrucciones comunes. Comenzaremos con el Pentium II.

El conjunto de instrucciones del Pentium II es una mezcla de instrucciones que tienen sentido en el modo de 32 bits y otras que se remontan a su anterior vida como un 8088. En la figura 5-33 mostramos algunas de las instrucciones para enteros que con más frecuencia usan los compiladores y programadores en la actualidad. La lista dista mucho de ser completa, pues no incluye instrucciones de punto flotante, instrucciones de control, y ni siquiera algunas de las instrucciones para enteros más exóticas (como usar un byte de 8 bits en AL para consultar una tabla). A pesar de todo, la lista nos da una buena idea de las capacidades del Pentium II.

Muchas de las instrucciones del Pentium II hacen referencia a uno o dos operandos, sea en registros o en la memoria. Por ejemplo, la instrucción ADD de dos operandos suma la fuente y el destino, y la instrucción INC de un operando incrementa su operando (le suma 1). Algunas de las instrucciones tienen variantes íntimamente relacionadas con ellas. Por ejemplo, las instrucciones de desplazamiento pueden desplazar a la derecha o a la izquierda y

pueden dar al bit de signo un tratamiento especial o no. Casi todas las instrucciones tienen varias codificaciones distintas, dependiendo de la naturaleza de los operandos.

En la figura 5-33, los campos SRC (*Source*) son fuentes de información y no se modifican. En contraste, los campos DST son destinos y la instrucción normalmente los modifica. Existen reglas acerca de qué está permitido y qué no como fuente o destino, las cuales varían de forma un tanto irregular de una instrucción a otra, pero no entraremos en detalles aquí. Muchas instrucciones tienen tres variantes, para operandos de 8, 16 y 32 bits, respectivamente. Éstas se distinguen por tener diferente código de operación y/o por un bit en la instrucción. En la lista de la figura 5-33 se hace hincapié en las instrucciones de 32 bits.

Por comodidad, hemos dividido las instrucciones en varios grupos. El primer grupo contiene instrucciones que trasladan datos dentro de la máquina, entre registros, la memoria y la pila. El segundo grupo efectúa operaciones aritméticas con y sin signo. Para la multiplicación y división, el producto o dividendo de 64 bits se guarda en EAX (parte de orden bajo) y EDX (parte de orden alto).

El tercer grupo efectúa operaciones aritméticas en decimal codificado en binario (BCD), tratando cada byte como dos **nibbles** de 4 bits. Cada nibble contiene un dígito decimal (0 a 9). Las combinaciones de bits 1010 a 1111 no se usan. Así, un entero de 16 bits puede contener un número decimal de 0 a 9999. Si bien esta forma de almacenamiento es poco eficiente, elimina la necesidad de convertir entradas decimales a binario y luego otra vez a decimal para enviarlos a la salida. Estas instrucciones sirven para realizar operaciones aritméticas con los números BCD, y se usan mucho en programas en COBOL.

Las instrucciones booleanas y de desplazamiento/rotación manipulan los bits de una palabra o byte de diversas maneras. Se proporcionan varias combinaciones.

Los dos grupos que siguen se ocupan de efectuar pruebas y comparaciones, para luego saltar con base en los resultados. Los resultados de las instrucciones de prueba y comparación se guardan en diversos bits del registro EFLAGS. Jxx representa un conjunto de instrucciones de saltos condicionales, dependiendo de los resultados de la comparación anterior (es decir, los bits de EFLAGS).

El Pentium II cuenta con varias instrucciones para cargar, almacenar, trasladar, comparar y examinar cadenas de caracteres o palabras. Estas instrucciones pueden ir precedidas por un byte especial llamado REP, que hace que se repitan hasta que se cumpla cierta condición, como que ECX, que se decrementa después de cada iteración, llegue a cero. Esto permite trasladar, comparar, etc., bloques arbitrarios de datos.

El último conjunto de instrucciones no puede clasificarse en ningún otro grupo. Incluyen conversiones, administración de pila, paro de la CPU, y E/S.

El Pentium II tiene varios **prefijos**, de los cuales ya mencionamos uno (*REP*). Cada uno de estos prefijos es un byte especial que puede anteponerse a casi todas las instrucciones, análogo a WIDE en IJVM. REP hace que la instrucción que sigue se repita hasta que ECX sea cero, como ya explicamos. REPZ y REPNZ ejecutan repetidamente la instrucción que sigue hasta que se enciende el código de condición *Z*, o hasta que se apaga, respectivamente. LOCK reserva el bus durante toda la instrucción para poder sincronizar multiprocesadores. Otros prefijos sirven para hacer que una instrucción opere en modo de 16 bits o en modo de 32 bits, lo que no sólo altera la longitud de los operandos, sino que también redefine totalmente los modos de direccionamiento. Por último, el Pentium II tiene un esquema de seg-

Traslados

MOV DST,SRC	Pasar SRC a DST
PUSH SRC	Meter SRC en la pila
POP DST	Sacar una palabra de la pila a DST
XCHG DS1,DS2	Intercambiar DS1 y DS2
LEA DST,SRC	Cargar dir. efectiva de SRC en DST
CMOV DST,SRC	Traslado condicional

Aritmética

ADD DST,SRC	Sumar SRC a DST
SUB DST,SRC	Restar DST de SRC
MUL SRC	Multiplicar EAX por SRC (sin signo)
IMUL SRC	Multiplicar EAX por SRC (con signo)
DIV SRC	Dividir EDX:EAX entre SRC (sin signo)
IDIV SRC	Dividir EDX:EAX entre SRC (con signo)
ADC DST,SRC	Sumar SRC a DST, luego sumar acarreo
SBB DST,SRC	Restar DST y acarreo a SRC
INC DST	Sumar 1 a DST
DEC DST	Restar 1 a DST
NEG DST	Negar DST (restarlo a 0)

Decimal codificado en binario

DAA	Ajuste decimal
DAS	Ajuste decimal para restar
AAA	Ajuste ASCII para sumar
AAS	Ajuste ASCII para restar
AAM	Ajuste ASCII para multiplicar
AAD	Ajuste ASCII para dividir

Booleanas

AND DST,SRC	AND booleano de SRC en DST
OR DST,SRC	OR booleano de SRC en DST
XOR DST,SRC	OR EXCLUSIVO booleano de SRC en DST
NOT DST	Sutituir DST por su complemento a uno

Desplazamiento/rotación

SAL/SAR DST,#	Desplazar DST izq/der # bits
SHL/SHR DST,#	Desplazam. lógico de DST # bits a izq/der
ROL/ROR DST,#	Girar DST # bits a la izq/der
RCL/RCR DST,#	Girar DST # bits pasando por acarreo

Probar/comparar

TST SRC1,SRC2	AND booleano de operandos, ajustar banderas
CMP SRC1,SRC2	Ajustar banderas con base en SRC1 _ SRC2

Transferencia de control

JMP ADDR	Saltar a la dirección ADDR
Jxx ADDR	Saltos condicionales con base en banderas
CALL ADDR	Llamar procedimiento que está en ADDR
RET	Regresar de procedimiento
IRET	Regresar de interrupción
LOOPxx	Ciclo hasta que se cumpla condición
INT ADDR	Iniciar interrupción por software
INTO	Interrumpir si se enciende bit de desbord.

Cadenas

LODS	Cargar cadena
STOS	Almacenar cadena
MOVS	Trasladar cadena
CMPS	Comparar dos cadenas
SCAS	Examinar cadenas

Códigos de condición

STC	Encender bit de acarreo en reg. EFLAGS
CLC	Apagar bit de acarreo en reg. EFLAGS
CMC	Complementar bit de acarreo en reg. EFLAGS
STD	Encender bit de sentido en reg. EFLAGS
CLD	Apagar bit de sentido en reg. EFLAGS
STI	Encender bit de interrupción en reg. EFLAGS
CLI	Apagar bit de interrupción en reg. EFLAGS
PUSHFD	Meter registro EFLAGS en la pila
POPFD	Sacar registro EFLAGS de la pila
LAHF	Cargar AH con el registro EFLAGS
SAHF	Guardar AH en el registro EFLAGS

Diversas

SWAP DST	Cambiar endianitud de DST
CWQ	Extender EAX a EDX:EAX para dividir
CWDE	Extender núm. de 16 bits en AX a EAX
ENTER SIZE,LV	Crear marco de pila con SIZE bytes
LEAVE	Deshacer marco de pila construido con ENTER
NOP	Ninguna operación
HLT	Parar
IN AL,PORT	Aceptar un byte del puerto PORT en AL
OUT PORT,AL	Enviar un byte de AL al puerto PORT
WAIT	Esperar interrupción

SRC = fuente
DST = destino
# = cuenta para despl./rot.
LV = núm. variables locales

Figura 5-33. Algunas de las instrucciones del Pentium II que manejan enteros.

mentación complejo con segmentos de código, datos, pila y extra, una reliquia del 8088. Se incluyen prefijos para obligar a las referencias a memoria a usar segmentos específicos, pero (por fortuna) no nos ocuparemos de éstos.

5.5.9 Las instrucciones del UltraSPARC II

Todas las instrucciones de modo de usuario del UltraSPARC II que manejan enteros y que un compilador podría generar se enumeran en la figura 5-34. No se presentan aquí las instrucciones de punto flotante, las de control (por ejemplo, gestión de caché y restablecimiento del sistema), las que hacen referencia a espacios de direcciones distintos del usuario, ni las obsoletas. El conjunto es sorprendentemente pequeño; el UltraSPARC II realmente es una computadora con conjunto de instrucciones reducido.

Las instrucciones LOAD y STORE son sencillas, con versiones para 1, 2, 4 y 8 bytes. Cuando se carga un número de menos de 64 bits en un registro (de 64 bits), el número puede extenderse con el signo o con cero. Hay instrucciones para ambas opciones.

El siguiente grupo es para aritmética. Las instrucciones que incluyen CC en su nombre ajustan los bits de código de condición NZVC; las demás no lo hacen. En las máquinas CISC, casi todas las instrucciones ajustan los códigos de condición, pero en una máquina RISC ello no es deseable porque restringe la libertad del compilador para cambiar las instrucciones de lugar cuando trata de llenar ranuras de retraso. Si el orden original de las instrucciones es A ... B ... C, y A establece los códigos de condición y B los prueba, el compilador no podrá insertar C entre A y B si C ajusta los códigos de condición. Por esta razón se proporcionan dos versiones de muchas instrucciones; el compilador normalmente usa la que no ajusta los códigos de condición, a menos que planee probarlos después. Hay instrucciones para multiplicación, división con signo y división sin signo.

La aritmética etiquetada es un formato especial con autoidentificación para números de 30 bits. Se puede usar para lenguajes como Smalltalk o Prolog, en los que las variables no tienen un tipo en el momento de la compilación y pueden cambiar de tipo durante la ejecución. Con números etiquetados, el compilador puede generar una instrucción ADD, y en el momento de la ejecución la máquina determina si necesita un ADD para enteros o uno para números de punto flotante.

El grupo de desplazamiento contiene un desplazamiento a la izquierda y dos a la derecha, cada uno con una versión de 32 bits y una versión extendida de 64 bits. En el caso de SLL, se desplazan los 64 bits, ya que esto sigue siendo compatible con el software viejo. Los desplazamientos se usan primordialmente para manipular bits. Casi todas las máquinas CISC tienen un número enorme de instrucciones de desplazamiento y rotación, casi todas ellas absolutamente inútiles. Pocos escritores de compiladores pasarán noches en vela lamentando su ausencia.

El grupo de instrucciones booleanas es análogo al grupo de instrucciones aritméticas: incluye AND, OR, OR EXCLUSIVO, ANDN, ORN y XNORN. Las últimas tres son de valor dudoso, pero pueden ejecutarse en un solo ciclo y casi no requieren hardware adicional, así que se incluyeron. Hasta los diseñadores de máquinas RISC ocasionalmente caen en la tentación.

El siguiente grupo de instrucciones contiene las transferencias de control. BPcc representa un conjunto de instrucciones que ramifican según las distintas condiciones, y especifi-

Cargas

LDSB ADDR,DST	Cargar byte con signo (8 bits)
LDUB ADDR,DST	Cargar byte sin signo (8 bits)
LDSH ADDR,DST	Cargar media palabra con signo (16 bits)
LDUH ADDR,DST	Cargar media palabra sin signo (16 bits)
LDSW ADDR,DST	Cargar palabra con signo (32 bits)
LDUW ADDR,DST	Cargar palabra sin signo (32 bits)
LDX ADDR,DST	Cargar extendido (64 bits)

Almacenamientos

STB SRC,ADDR	Almacenar byte (8 bits)
STH SRC,ADDR	Almacenar media palabra (16 bits)
STW SRC,ADDR	Almacenar palabra (32 bits)
STX SRC,ADDR	Almacenar extendido (64 bits)

Aritmética

ADD R1,S2,DST	Sumar
ADDCC "	Sumar y fijar icc
ADDC "	Sumar con acarreo
ADDCCC "	Sumar con acarreo y fijar icc
SUB R1,S2,DST	Restar
SUBCC "	Restar y fijar icc
SUBC "	Restar con acarreo
SUBCCC "	Restar con acarreo y fijar icc
MULX R1,S2,DST	Multiplicar
SDIVX R1,S2,DST	Dividir con signo
UDIVX R1,S2,DST	Dividir sin signo
TADCC R1,S2,DST	Suma etiquetada

Desplazamientos/rotaciones

SLL R1,S2,DST	Despl. lógico a la izq. (64 bits)
SLLX R1,S2,DST	Despl. lógico a la izq. extendido (64)
SRL R1,S2,DST	Despl. lógico a la der. (32 bits)
SRLX R1,S2,DST	Despl. lógico a la der. extendido (64)
SRA R1,S2,DST	Despl. aritmético a la der. (32 bits)
SRAX R1,S2,DST	Despl. aritmético a la der. ext. (64)

Booleanas

AND R1,S2,DST	AND booleano
ANDCC "	AND booleano y fijar icc
ANDN "	NAND booleano
ANDNCC "	NAND booleano y fijar icc
OR R1,S2,DST	OR booleano
ORCC "	OR booleano y fijar icc
ORN "	NOR booleano
ORNCC "	NOR booleano y fijar icc
XOR R1,S2,DST	XOR booleano
XORCC "	XOR booleano y fijar icc
XNOR "	XOR booleano
XNORCC "	XOR booleano y fijar icc

Transferencia de control

BPcc ADDR	Ramif. con predicción
BPr SRC,ADDR	Ramif. según registro
CALL ADDR	Llamar procedimiento
RETURN ADDR	Regresar de procedimiento
JMPL ADDR,DST	Saltar y vincular
SAVE R1,S2,DST	Adelantar ventana de registros
RESTORE "	Restaurar ventana de registros
Tcc CC,TRAP#	Trampa si se cumple condición
PREFETCH FCN	Preobtener datos de memoria
LDSTUB ADDR,R	Carga/almacenamiento atómico
MEMBAR MASK	Barrera de memoria

Diversas

SETHI CON,DST	Establecer bits 10 a 31
MOVcc CC,S2,DST	Trasladar si se cumple cond.
MOVr R1,S2,DST	Trasladar según registro
NOP	Ninguna operación
POPC S1,DST	Cuenta de población
RDCCR V,DST	Leer registro de cód. de cond.
WRCCR R1,S2,V	Escribir registro de cód. de cond
RDPC V,DST	Leer contador de programa

SRC = registro fuente
DST = registro de destino
R1 = registro fuente
S2 = fuente: registro o inmediato
ADDR = dirección en memoria

TRAP# = Número de trampa
FCN = código de función
MASK = tipo de operación
CON = constante
V = designador de registro

CC = se ajusta cód. de cond.
R = registro de destino

cc = condición
r = LZ,LEZ,Z,NZ,GZ,GEZ

Figura 5-34. Las instrucciones primarias de UltraSPARC II para enteros.

can en la instrucción si el compilador cree que se tomará la rama o no. BPr prueba un registro y ramifica según la condición detectada.

Se proporcionan dos mecanismos para invocar procedimientos. La instrucción CALL usa el formato 4 de la figura 5-14 con una distancia en *palabras* relativa al PC de 32 bits. Este valor basta para alcanzar cualquier instrucción que esté a menos de 2 gigabytes de la que invoca, en cualquier sentido. La instrucción CALL deposita la dirección de retorno en R15, que se convierte en R31 después de la llamada.

La otra forma de invocar un procedimiento es usar JMPL, que usa el formato 1a o 1b, y permite colocar la dirección de retorno en cualquier registro. Esta forma es útil cuando la dirección objetivo se calcula durante la ejecución.

SAVE y RESTORE manipulan la ventana de registros y el apuntador de la pila. Ambas causan una trampa cuando la ventana siguiente (anterior) no está disponible.

El último grupo contiene diversas instrucciones. SETHI es necesaria porque no hay forma de colocar un operando inmediato de 32 bits en un registro. La forma como se hace es usar SETHI para establecer los bits del 10 al 31 y luego hacer que la siguiente instrucción proporcione los demás bits empleando el formato inmediato.

La instrucción de cuenta de población es un misterio: cuenta el número de bits 1 en una palabra. Hay rumores de que es buena para simular explosiones de bombas, y que Los Alamos National Laboratory (que no escatima en gastos) ve con buenos ojos las máquinas que cuentan con ella. Las últimas tres instrucciones son para leer y escribir registros especiales.

Varias instrucciones CISC conocidas que están ausentes en esta lista se pueden simular usando G0 o un operando constante (formato 1b). En la figura 5-35 se dan unas cuantas de ellas. El ensamblador de UltraSPARC II las reconoce y los compiladores las generan con frecuencia. Muchas de ellas aprovechan el hecho de que G0 está conectado permanentemente a 0 y que escribir en él no tiene ningún efecto.

5.5.10 Las instrucciones del picoJava II

Ha llegado el momento de examinar el nivel ISA del picoJava II, que implementa el conjunto de instrucciones JVM completo, con sus 226 instrucciones, además de 115 instrucciones extra cuyo propósito es implementar C, C++ y el sistema operativo. Nos concentraremos primordialmente en las instrucciones JVM, ya que éstas son las únicas que produce el compilador de Java. La ISA JVM no tiene registros visibles para el usuario ni las demás características que la mayor parte de las CPU tienen en común. (El picoJava II tiene 64 registros en el chip para el tope de la pila, pero los programas de usuario no pueden verlos.) Casi todas las instrucciones de JVM meten palabras en la pila, operan con palabras que ya están en la pila, y obtienen palabras de la pila.

El hardware de picoJava II ejecuta directamente la mayor parte de las instrucciones JVM, pero unas cuantas están microprogramadas, y otras pocas saltan a través de una trampa a un manejador en software para su ejecución. Así, se requiere un sistema de tiempo de ejecución pequeño en el nivel ISA para que la máquina funcione, pero este sistema de tiempo de ejecución es mucho más pequeño que el intérprete JVM completo y no se invoca muy a menudo. Dicho sistema contiene código para interpretar unas cuantas instrucciones complejas, el cargador de clases, el verificador de códigos de byte, el gestor de líneas y el recolector de basura.

Instrucción	Cómo hacerlo
MOV SRC,DST	OR SRC con G0 y guardar el resultado en DST
CMP SRC1,SRC2	SUBCC SRC2 de SRC1 y guardar el resultado en G0
TST SRC	ORCC SRC1 con G0 y guardar el resultado en G0
NOT DST	XNOR DST con G0
NEG DST	SUB DST de G0 y guardar en DST
INC DST	ADD 1 a DST (operando inmediato)
DEC DST	SUB 1 de DST (operando inmediato)
CLR DST	OR G0 con G0 y guardar en DST
NOP	SETHI G0 a 0
RET	JMPL %17+8,%G0

Figura 5-35. Algunas instrucciones UltraSPARC II simuladas.

JVM tiene un conjunto de instrucciones relativamente pequeño y sencillo. El conjunto de instrucciones JVM se da en la figura 5-36 con excepción de algunas de las instrucciones anchas, rápidas y de forma corta. Aparte de esas excepciones, la lista está completa.

Las instrucciones JVM tienen tipo, así que hay diferentes instrucciones para efectuar la misma operación con diferentes tipos de datos. Por ejemplo, ILOAD mete un entero de 32 bits en la pila, mientras que ALOAD mete una referencia (apuntador) de 32 bits en la pila. Esta tipificación estricta no es necesaria para un funcionamiento correcto, ya que el resultado en ambos casos es que los 32 bits que están en la posición de memoria especificada se transfieren a la pila. La tipificación estricta se incluye en JVM para verificar durante la ejecución que un programa binario JVM no viola restricciones de seguridad al tratar de convertir subrepticiamente un entero en un apuntador y así acceder a memoria que no se supone deba tocar.

Repasemos ahora las instrucciones JVM. La primera instrucción de la lista es

typeLOAD IND8

En realidad, ésta no es una instrucción, sino una plantilla para generar instrucciones. JVM es regular, así que en lugar de enumerar las instrucciones individualmente, en algunos casos proporcionamos una regla para generarlas. En este caso, type representa una de las cuatro letras I, L, F o D, que corresponden a los tipos *integer, long, float* (punto flotante de 32 bits) y *double* (punto flotante de 64 bits), respectivamente. Así, hay cuatro instrucciones implícitas aquí, ILOAD, LLOAD, FLOAD y DLOAD, cada una de las cuales usa el índice de 8 bits, IND8, para encontrar una variable local, y mete en la pila un valor con la longitud y el tipo correctos. Ya vimos una de éstas (ILOAD) en IJVM, pero las otras funcionan de la misma

Cargas

typeLOAD IND8	Apilar variable local
typeALOAD	Apilar elemento de arreglo
BALOAD	Apilar byte de un arreglo
SALOAD	Apilar short de un arreglo
CALOAD	Apilar char de un arreglo
AALOAD	Apilar apuntador de un arreglo

Almacenamientos

typeSTORE IND8	Sacar valor y guardar en var local
typeASTORE	Sacar valor y guardar en arreglo
BASTORE	Sacar byte y guardar en arreglo
SASTORE	Sacar short y guardar en arreglo
CASTORE	Sacar char y guardar en arreglo
AASTORE	Sacar pointer y guardar en arreglo

Empilado

BIPUSH CON8	Apilar constante pequeña
SIPUSH CON16	Apilar constante de 16 bits
LDC IND8	Apilar constante de res. de ctes.
typeCONST_#	Apilar constante inmediata
ACONST_NULL	Apilar apuntador nulo

Aritmética

typeADD	Sumar
typeSUB	Restar
typeMUL	Multiplicar
typeDIV	Dividir
typeREM	Residuo
typeNEG	Negar

Booleanas/desplazamiento

ilAND	AND booleano
ilOR	OR booleano
ilXOR	OR EXCLUSIVO booleano
ilSHL	-Desplazamiento a la izq.
ilSHR	Desplazamiento a la der.
ilUSHR	Desplazam. a la der. sin signo

Conversión

x2y	Convertir x en y
i2c	Convertir integer en char
i2b	Convertir integer en byte

Gestión de pila

DUPxx	Seis instrucc. para duplicar
POP	Desapilar integer y desechar
POP2	Desapilar dos integer y desechar
SWAP	Intercambiar dos int. del tope

Comparación

IF_ICMPrel OFFSET16	Ramificación condicional
IF_ACMPEQ OFFSET16	Ramif. si 2 apunt. iguales
IF_ACMPNE OFFSET16	Ramif. si apunt. distintos
IFrel OFFSET16	Probar valor y ramif.
IFNULL OFFSET16	Ramif. si apunt. nulo
IFNONNULL OFFSET16	Ramif. si apunt. no nulo
LCMP	Comparar dos longs
FCMPL	Comparar 2 floats si <
FCMPG	Comparar 2 floats si >
DCMPL	Comparar doubles si <
DCMPG	Comparar doubles si >

Transferencia de control

INVOKEVIRTUAL IND16	Invocación de método
INVOKESTATIC IND16	Invocación de método
INVOKEINTERFACE ...	Invocación de método
INVOKESPECIAL IND16	Invocación de método
JSR OFFSET16	Invocar cláusula finally
typeRETURN	Devolver valor
ARETURN	Devolver apuntador
RETURN	Devolver void
RET IND8	Regresar de finally
GOTO OFFSET16	Ramif. incondicional

Arreglos

ANEWARRAY IND16	Crear arreglo de apunt.
NEWARRAY ATYPE	Crear arreglo de atype
MULTINEWARRAY IN16,D	Crear arreglo multidim.
ARRAYLENGTH	Obtener longitud de arreglo

Varias

IINC IND8,CON8	Incrementar variable local
WIDE	Prefijo ancho
NOP	Ninguna operación
GETFIELD IND16	Leer campo de objeto
PUTFIELD IND16	Escribir campo en objeto
GETSTATIC IND16	Obt. campo static de clase
NEW IND16	Crear objeto nuevo
INSTANCEOF OFFSET16	Determinar tipo de objeto
CHECKCAST IND16	Verif. tipo de objeto
ATHROW	Lanzar excepción
LOOKUPSWITCH ...	Ramif. multivías rala
TABLESWITCH ...	Ramif. multivías densa
MONITORENTER	Ingresar en monitor
MONITOREXIT	Salir de monitor

IND8/16 = índice de var. local type, x, y = I, L, F, D
CON8/16, D, ATYPE = constante OFFSET16 para ramif.

Figura 5-36. El conjunto de instrucciones JVM.

manera; la única diferencia radica en el número de palabras que se apilan y en el tipo del valor.

Además de estas cuatro instrucciones de carga, hay otras cuatro implícitas en typeALOAD, todas las cuales almacenan elementos de un arreglo en la pila. En los cuatro casos, la pila debe precargarse con un apuntador al arreglo y con el índice del elemento. Estas instrucciones desapilan el índice y el apuntador, realizan el cálculo necesario para localizar el elemento, y luego meten el elemento en la pila. El cálculo necesita conocer el tamaño de los elementos, que está implícito en el tipo. Estas instrucciones obtienen el índice del arreglo de la pila, por lo que la instrucción misma no tiene operando.

Las últimas cuatro instrucciones de este grupo también son para elementos de arreglo, pero de otros tipos: manejan byte (8 bits), short (16 bits), char (16 bits) y pointer (32 bits). Así, hay un total de 12 instrucciones LOAD en este grupo.

Las instrucciones typeSTORE funcionan igual que las instrucciones typeLOAD, pero en el otro sentido. Todas obtienen un elemento de la pila y lo guardan en una variable local. Una vez más, ya vimos una de éstas (ISTORE) en IJVM. En este caso también tenemos instrucciones para guardar elementos de un arreglo. En el tope de la pila hay un valor del tipo correcto. Debajo de él está el índice, y debajo de éste, el apuntador al arreglo. La operación desapila las tres cosas.

Las instrucciones de empilado meten valores en la pila. Ya vimos BIPUSH. SIPUSH hace lo mismo, sólo que con un número de 16 bits. LDC almacena en la pila un valor de la reserva de constantes. La siguiente instrucción representa todo un grupo de instrucciones inmediatas, de los cuatro tipos básicos (*integer, long, float y double*). Todas consisten únicamente en un código de operación y cada código almacena un valor específico en la pila. Por ejemplo, ICONST_0 (código 0x03) almacena una palabra 0 de 32 bits en la pila. Desde luego, puede hacerse la misma cosa con BIPUSH, pero usando dos bytes. Al optimar los casos más comunes, se reduce el tamaño de los programas JVM. Los valores que se manejan son los enteros –1, 0, 1, 2, 3, 4, 5, los longs 0 y 1, los floats 0.0, 1.0 y 2.0, y los doubles 0.0 y 1.0. De forma similar, CONST_NULL almacena un apuntador nulo en la pila. La combinación de códigos de operación con las direcciones más comunes en una instrucción de un solo byte reduce considerablemente el tamaño del programa, ahorra memoria y acorta el tiempo que toma transportar programas binarios en Java por Internet.

Las operaciones aritméticas son totalmente regulares, con seis operaciones para cada uno de los cuatro tipos básicos. Las tres operaciones booleanas y las tres de desplazamiento sólo están disponibles para integers y longs. Tener una instrucción para efectuar un AND bit por bit con números de punto flotante habría sido contrario a las estrictas reglas de tipificación de la JVM. La entrada x2y de la tabla representa una matriz 4×4 de instrucciones de conversión. Valores de cada tipo se pueden convertir a cada uno de los otros tipos (pero no al mismo tipo), así que hay 12 instrucciones aquí. Una instrucción representativa es I2F, que convierte un integer en un float.

El grupo de gestión de pila tiene instrucciones para duplicar el valor o los dos valores del tope, colocarlos en diversos lugares de la pila (no siempre en el tope). Otras instrucciones sacan valores de la pila, e intercambian los dos valores del tope.

Las instrucciones del grupo de comparación sacan uno o dos valores de la pila y los examinan. Cuando se sacan dos valores de la pila, se restan y se prueba el resultado. Cuando se saca un solo valor, se prueba su valor. El sufijo rel representa los operadores relacionales LT, LE, EQ, NE, GE y GT. Las instrucciones que tienen una distancia toman la rama si la prueba se satisface. Las demás meten un resultado de vuelta en la pila.

El siguiente grupo permite invocar procedimientos y devolver valores. Vimos versiones muy simplificadas de INVOKEVIRTUAL e IRETURN en IJVM. Las versiones completas tienen muchas más opciones, y hay varias instrucciones para cubrir muchos casos. Una explicación completa rebasaría el alcance de este libro. Si quiere conocer toda la historia, vea Lindholm y Yellin (1997).

El siguiente grupo contiene cuatro instrucciones para crear arreglos unidimensionales y multidimensionales y preguntar por sus longitudes. Los arreglos se guardan en el montículo en JVM y se recolectan dinámicamente como basura cuando ya no se necesitan.

El último grupo es una colección miscelánea, cada una con un propósito específico relacionado con alguna característica del lenguaje Java. Su explicación rebasa el alcance de este libro.

Hablemos ahora un poco acerca del nivel ISA del picoJava II. Se trata de una máquina big endian (aunque hay unas cuantas instrucciones que se puede hacer que funcionen como little endian). Las palabras son de 32 bits, aunque hay instrucciones para manejar cantidades de 8, 16 y 64 bits. La pila crece hacia abajo en la memoria, de direcciones altas a direcciones bajas, al contrario que en IJVM (la especificación de JVM permite las dos cosas).

El picoJava II se diseñó para ejecutar programas en C y C++ además de programas en Java. Sin embargo, para hacerlo se requiere un compilador que convierta C o C++ a la ISA del picoJava II. Una vez compilado a JVM un programa en C o C++, todas las optimaciones de hardware que describimos en el capítulo 4 para Java (como el plegado) también aplican a C o C++.

Para poder ejecutar programas en C y C++ en el picoJava II, se añadieron 115 instrucciones al nivel ISA. Casi todas tienen dos o más bytes de longitud y comienzan con uno de dos códigos de operación JVM reservados (0xFE y 0xFF) para indicar que sigue una instrucción extendida. A continuación presentaremos un breve resumen de las características ajenas a la JVM del picoJava II en el nivel ISA.

El picoJava II tiene 25 registros de 32 bits. Cuatro de ellos son funcionalmente equivalentes a los registros PC, LV, SP y CPP de IJVM. El registro OPLIM establece un límite para SP. Si SP rebasa OPLIM, ocurre una trampa. Esta característica sirve para poder representar la pila como una lista enlazada de trozos de pila, en lugar de un bloque contiguo de la memoria. El registro FRAME marca el final de las variables locales y apunta a la palabra que contiene el contador de programa del procedimiento invocador.

Hay otros registros, como la palabra de situación del programa, un registro que se mantiene al tanto de hasta dónde se ha llenado la caché de pila de 64 registros, y cuatro registros que sirven para gestión de hilos. También hay cuatro registros para trampas y puntos de ruptura, y cuatro registros que los programas en C y C++ usan para invocar procedimientos y devolver valores. Puesto que el picoJava II no tiene memoria virtual, dos registros acotan la parte de la memoria a la que el programa vigente puede accesar, a fin de limitar los daños que

podría causar un programa en C o C++ desbocado. Otros registros contienen el número de versión e información sobre la configuración del hardware.

Las instrucciones extendidas pertenecen a cinco categorías amplias. Primero están las instrucciones para leer los registros anteriores y escribir en ellos. Luego vienen las instrucciones para manipular apuntadores. Éstas permiten leer o escribir palabras arbitrarias en la memoria. Casi todas sacan una dirección de máquina de la pila y luego almacenan en la pila el byte, short, palabra, etc., que está en esa dirección. Tales instrucciones violan descaradamente la seguridad de tipos de Java, pero se necesitan para C y C++.

En tercer lugar están las instrucciones útiles para programas en C y C++, como las que invocan procedimientos y regresan de ellos sin usar las instrucciones JVM (de trabajo pesado). En cuarto lugar están las instrucciones que manejan el hardware, como por ejemplo la caché. Luego vienen varias instrucciones de diversa índole, como para diagnóstico al encender la máquina. Claro que los programas que usan estas instrucciones extra no se pueden transportar a otras plataformas JVM.

5.5.11 Comparación de conjuntos de instrucciones

Los tres conjuntos de instrucciones de ejemplo son muy distintos. El Pentium II es una máquina CISC clásica de dos direcciones y 32 bits, con una larga historia, modos de direccionamiento peculiares e irregulares, y muchas instrucciones que hacen referencia a la memoria. El UltraSPARC II es una máquina RISC moderna de tres direcciones y 64 bits, con una arquitectura de carga/almacenamiento, pocos modos de direccionamiento y un conjunto de instrucciones compacto y eficiente. La arquitectura JVM es una máquina de pila casi sin modos de direccionamiento, un conjunto de instrucciones altamente regular, y codificación de instrucciones muy densa.

Cada máquina es como es por una buena razón. El diseño del Pentium II estuvo determinado por tres factores principales:

1. Compatibilidad con modelos anteriores.
2. Compatibilidad con modelos anteriores.
3. Compatibilidad con modelos anteriores.

Dados los modernos avances tecnológicos, nadie diseñaría ahora una máquina tan irregular con tan pocos registros, todos distintos. Esto dificulta la escritura de compiladores. La falta de registros también obliga a los compiladores a verter constantemente variables en la memoria y luego volverlas a cargar: actividades muy costosas aun con dos o tres niveles de caché. Es un tributo a la calidad de los ingenieros de Intel que el Pentium II sea tan rápido, aun con las restricciones de esta ISA. Sin embargo, como vimos en el capítulo 4, la implementación es en extremo compleja y requiere dos veces más transistores que el picoJava II y casi 50% más que el UltraSPARC II.

El UltraSPARC II representa lo más moderno en diseños de ISA: cuenta con una ISA cabal de 64 bits (con un bus de 128 bits); tiene muchos registros, y un conjunto de instruccio-

nes que hace hincapié en las operaciones de tres registros, además de un pequeño grupo de instrucciones LOAD y STORE. Todas las instrucciones tienen el mismo tamaño, aunque el número de formatos ha crecido demasiado. No obstante, el conjunto de instrucciones se presta a una implementación directa y eficiente. Casi todos los nuevos diseños tienden a parecerse al UltraSPARC II, pero con menos formatos de instrucciones.

La JVM es harina de otro costal. La ISA se diseñó originalmente de modo que las applets (programas pequeños) se pudieran transmitir por Internet e interpretarse en software de forma eficiente en el destino. También es un diseño para un solo lenguaje. Esto llevó al uso de una pila e instrucciones cortas pero de longitud variable con una densidad elevada (un promedio de sólo 1.8 bytes por instrucción). Construir un dispositivo de hardware que realmente ejecutara una instrucción JVM a la vez, haciendo dos o tres referencias a la memoria en cada instrucción, sería un desastre en términos de desempeño. Sin embargo, al colocar una pila de 64 palabras en el chip y plegar muchas instrucciones para convertir sucesiones enteras en instrucciones RISC modernas de tres direcciones, el picoJava II logra tomar una ISA muy ineficiente y trabajar bien con ella.

El criterio definitivo aquí es que el corazón de una computadora moderna es una máquina RISC de carga/almacenamiento, de tres registros, con filas de procesamiento. El UltraSPARC II simplemente expone la máquina al usuario. El Pentium II la oculta tomando una ISA más vieja y dividiendo las instrucciones CISC en microoperaciones RISC que se pueden ejecutar de forma eficiente. El picoJava II también oculta la máquina RISC que está en su centro, pero lo hace combinando varias instrucciones ISA en una sola instrucción RISC, exactamente lo contrario de lo que el Pentium II hace.

En conclusión, si Ricitos de Oro se encontrara con algunas instrucciones de computadora en medio del bosque, vería que algunas son demasiado grandes (las instrucciones del Pentium II tienen que partirse); otras son demasiado pequeñas (las instrucciones del picoJava II tienen que combinarse) y unas más son justo del tamaño correcto (las instrucciones del UltraSPARC II se ejecutan tal cual). Ésta es la Teoría de los Tres Osos de la Computación.

5.6 FLUJO DE CONTROL

El término "flujo de control" se refiere al orden en que las instrucciones se ejecutan dinámicamente, es decir, durante la ejecución del programa. En general, si no hay ramificaciones y llamadas a procedimientos, las instrucciones que se ejecutan sucesivamente se traen de posiciones de memoria consecutivas. Las llamadas a procedimientos hacen que el flujo de control se altere, deteniendo el procedimiento que se está ejecutando e iniciando el procedimiento invocado. Las corrutinas están relacionadas con los procedimientos y causan alteraciones similares en el flujo de control; son útiles para simular procesos paralelos. Las trampas e interrupciones también hacen que se altere el flujo de control cuando se presentan condiciones especiales. Todos estos temas se tratarán en las secciones que siguen.

5.6.1 Flujo de control secuencial y ramificaciones

La mayor parte de las instrucciones no altera el flujo de control. Después de ejecutarse una instrucción, se trae la que le sigue en la memoria, y se ejecuta. Después de cada instrucción, el contador de programa se incrementa en una cantidad igual a la longitud de la instrucción. Si observamos este contador durante un intervalo largo en comparación con el tiempo medio de una instrucción, vemos que es una función aproximadamente lineal del tiempo, pues se incrementa en la longitud media de una instrucción en el tiempo medio de una instrucción. Dicho de otro modo, el orden dinámico en que el procesador realmente ejecuta las instrucciones es igual al orden en que aparecen en el listado del programa, como se muestra en la figura 5-37(a).

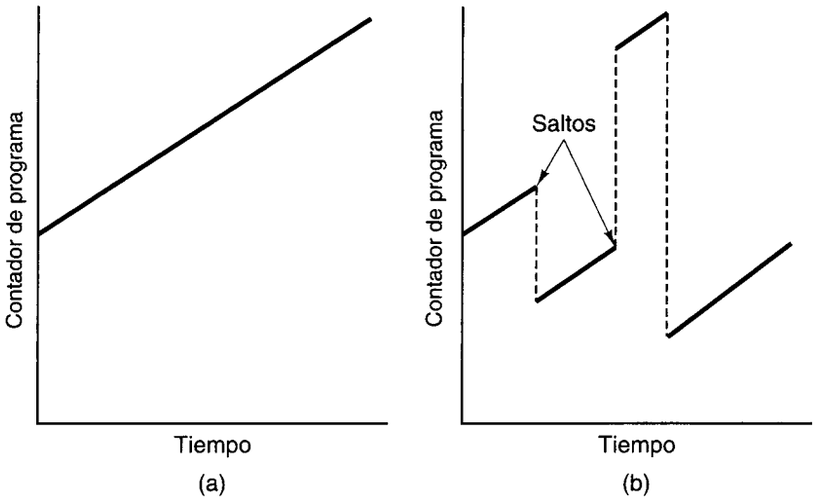


Figura 5-37. El contador de programa en función del tiempo (curva suavizada). (a) Sin ramificaciones. (b) Con ramificaciones.

Si un programa contiene ramificaciones, esta sencilla relación entre el orden en que las instrucciones aparecen en la memoria y el orden en que se ejecutan deja de ser real. Cuando hay ramificaciones, el contador de programa deja de ser una función monotónicamente creciente del tiempo, como se muestra en la figura 5-37(b). El resultado es que no es fácil visualizar la secuencia de ejecución de las instrucciones a partir del listado del programa. Cuando los programadores tienen dificultades para ver claramente el orden en que el procesador ejecutará las instrucciones, tienden a cometer errores. Esta observación llevó a Dijkstra (1986a) a escribir la carta, que tanta controversia causó, intitulada "El enunciado GOTO se considera perjudicial", en la que sugería evitar los enunciados goto. Esa carta dio origen a la revolución de la programación estructurada, uno de cuyos preceptos es la sustitución de enunciados goto por formas más estructuradas de control de flujo, como los ciclos while. Desde luego, la compilación de estos programas produce programas de nivel 2 que podrían contener muchas ramificaciones, porque la implementación de if, while y otras estructuras de control de alto nivel requiere ramificaciones.

5.6.2 Procedimientos

La técnica más importante para estructurar programas es el procedimiento. Desde un punto de vista, una llamada a un procedimiento altera el flujo de control como lo hace una ramificación pero, a diferencia de ésta, cuando el procedimiento termina su tarea devuelve el control al enunciado o instrucción que sigue a la llamada.

Desde otro punto de vista, el cuerpo de un procedimiento puede verse como la definición de una nueva instrucción en un nivel más alto. Desde esta perspectiva, una llamada a un procedimiento puede verse como una sola instrucción, aunque el procedimiento sea muy complicado. Para entender un fragmento de código que contiene una llamada a un procedimiento, basta con saber *qué* hace, no *cómo* lo hace.

Un tipo de procedimiento especialmente interesante es el **procedimiento recursivo**, es decir, un procedimiento que se llama a sí mismo, sea directamente o por medio de una cadena de procedimientos distintos. El estudio de procedimientos recursivos ayuda mucho a entender cómo se implementan las llamadas a procedimientos, y qué son realmente las variables locales. A continuación veremos un ejemplo de procedimiento recursivo.

Las "Torres de Hanoi" son un antiguo problema que tiene una solución recursiva sencilla. En cierto monasterio de Hanoi hay tres varillas de oro. Ensartadas en la primera había una serie de 64 discos de oro concéntricos, cada uno con un agujero en el centro para la varilla. Cada disco tiene un diámetro un poco menor que el del disco que está inmediatamente abajo. Las varillas segunda y tercera estaban vacías inicialmente. Los monjes del lugar están ocupados transfiriendo todos los discos a la varilla 3, un disco a la vez, sólo que en ningún momento un disco mayor puede descansar sobre uno más pequeño. Se dice que cuando los monjes terminen, el mundo llegará a su fin. Si quiere adquirir algo de experiencia práctica, puede usar discos de plástico y menos discos, pero cuando resuelva el problema no sucederá nada. Para lograr el efecto de fin-del-mundo se necesitan 64 discos, y de oro. La figura 5-38 muestra la configuración inicial para $n = 5$ discos.

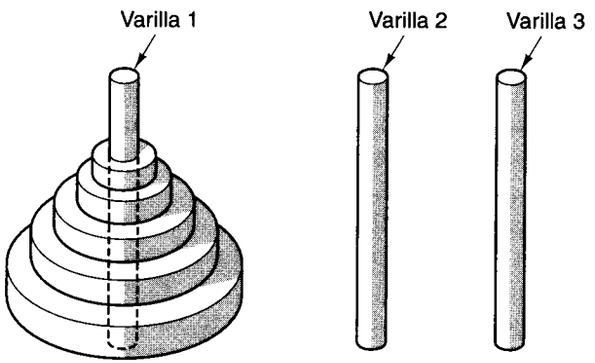


Figura 5-38. Configuración inicial del problema de las Torres de Hanoi con cinco discos.

La solución para transferir n discos de la varilla 1 a la 3 consiste en trasladar primero $n - 1$ discos de la varilla 1 a la 2, luego trasladar un disco de la varilla 1 a la 3, y por último trasladar $n - 1$ discos de la varilla 2 a la 3. Esta solución se ilustra en la figura 5-39.

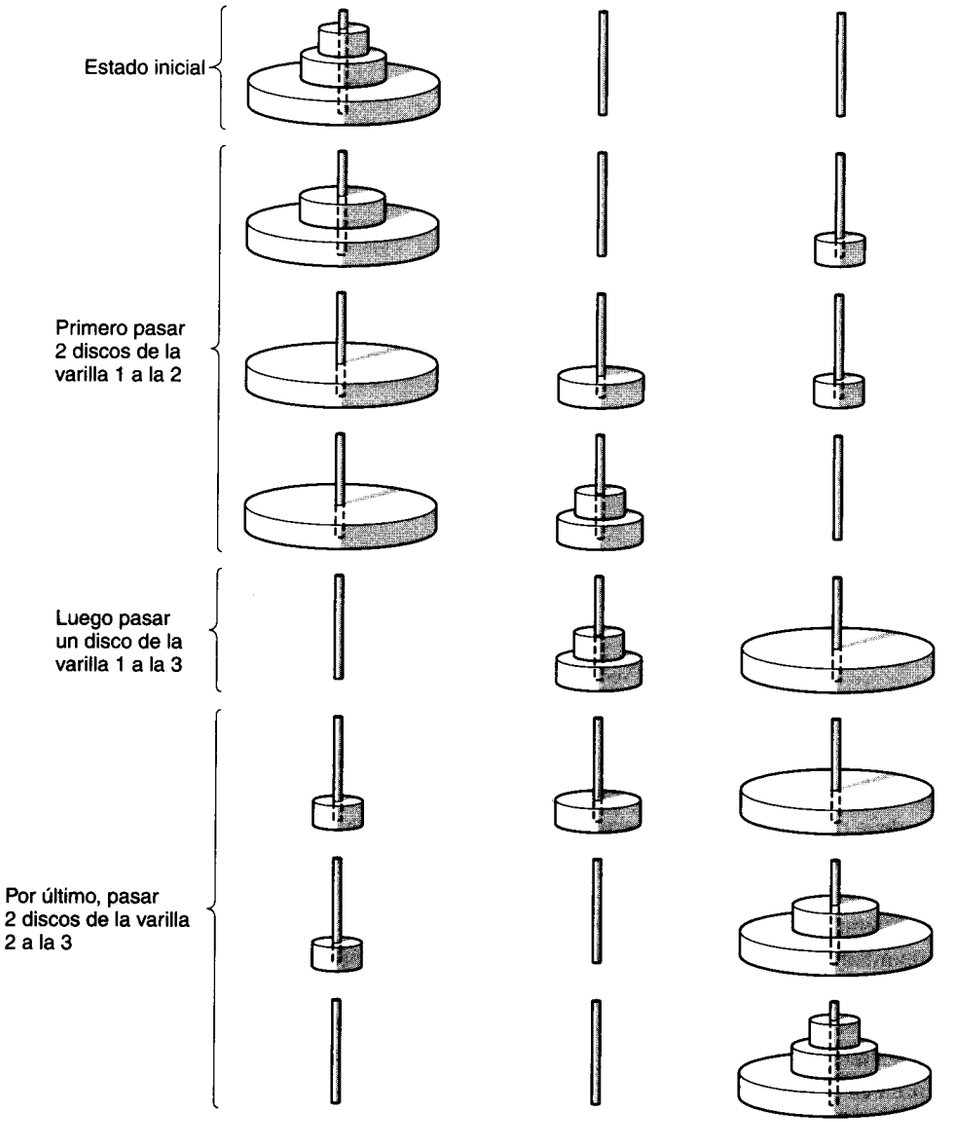


Figura 5-39. Pasos necesarios para resolver las Torres de Hanoi con tres discos.

Para resolver el problema necesitamos un procedimiento que traslade n discos de la varilla i a la varilla j. Cuando se invoque este procedimiento, con

```
torres(n,i, j)
```

se imprimirá la solución. El procedimiento primero prueba si $n = 1$. Si es así, la solución es trivial. Basta con pasar el disco de i a j. Si $n \neq 1$, la solución consiste en las tres partes que ya mencionamos, cada una de las cuales es una llamada a un procedimiento recursivo.

La solución completa se muestra en la figura 5-40. La llamada

```
torres(3, 1, 3)
```

para resolver el problema de la figura 5-39 genera tres llamadas más, a saber,

```
torres(2, 1, 2)
torres(1, 1, 3)
torres(2, 2, 3)
```

La primera y la tercera generan tres llamadas cada una, para un total de siete.

```
public void torres (int n, int i, int j) {
  int k;

    if (n==1)
    System.out. println("Pase un disco de " + i + " a " + j);
  else {
    k = 6 – i – j;
    torres(n – 1, i, k);
    torres(1, i, j);
    torres(n – 1, k, j);
 }
}
```

Figura 5-40. Procedimiento para resolver las Torres de Hanoi.

Para tener procedimientos recursivos necesitamos una pila en la cual guardar los parámetros y variables locales de cada invocación, como la que teníamos en IJVM. Cada vez que se invoca un procedimiento, se asigna un nuevo marco de pila para el procedimiento en el tope de la pila. El marco de creación más reciente es el marco vigente. En nuestros ejemplos la pila crece hacia arriba, de direcciones de memoria baja a direcciones altas, igual que en IJVM.

Además de apuntador de pila, que apunta al tope de la pila, suele ser conveniente tener un apuntador de marco, FP (*Frame Pointer*), que apunta a una posición fija dentro del marco. Podría apuntar al apuntador de enlace, como en IJVM, o a la primera variable local. La figura 5-41 muestra el marco de pila para una máquina con palabras de 32 bits. La llamada original a *torres* mete n, i y j en la pila y luego ejecuta una instrucción CALL que mete la dirección de retorno en la pila, en la dirección 1012. Al entrar, el procedimiento invocado guarda el valor viejo de FP (1000) en la pila en 1016 y luego adelanta el apuntador de pila a modo de asignar espacio para las variables locales. Ya que sólo hay una variable local de 32 bits (k), SP se incrementa en 4, a 1020. En la figura 5-41(a) se muestra la situación después de hacerse todas estas cosas.

Lo primero que un procedimiento debe hacer cuando se le invoca es guardar el FP anterior (para poder restaurarlo cuando se salga del procedimiento), copiar SP en FP, y tal vez

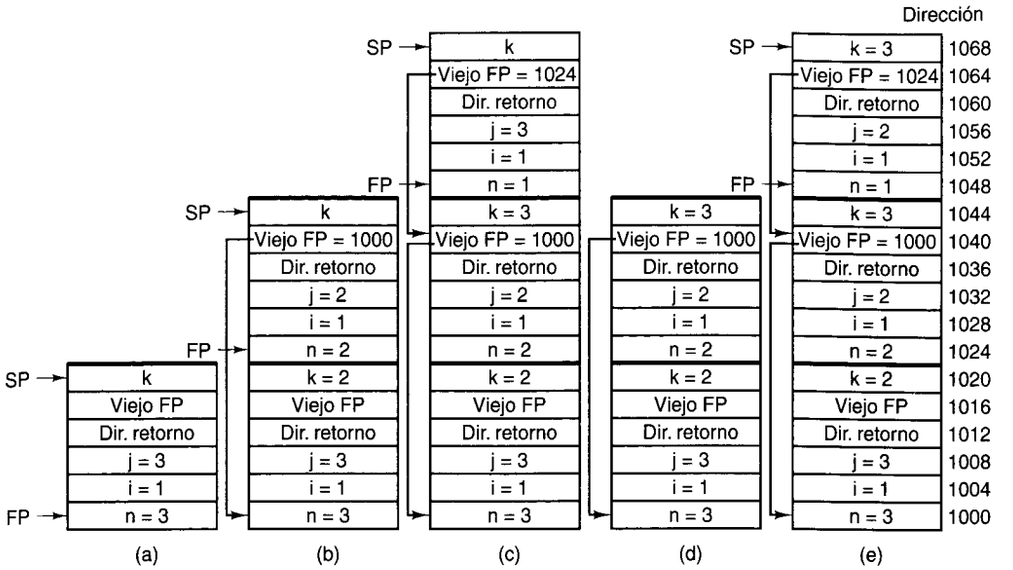


Figura 5-41. La pila en varios momentos durante la ejecución de la figura 5-40.

incrementarlo en una palabra, dependiendo de la parte del nuevo marco a la que FP apunta. En este ejemplo, FP apunta a la primera variable local, pero en IJVM, LV apuntaba al apuntador de enlace. Las diferentes máquinas manejan el apuntador de marco de forma un poco distinta; a veces lo colocan en la base del marco de pila, a veces hasta arriba y a veces a la mitad como en la figura 5-41. En este sentido, vale la pena comparar la figura 5-41 con la figura 4-12 para ver dos formas distintas de manejar el apuntador de enlace. Éstas no son las únicas formas. En todos los casos, la clave es la capacidad para efectuar después un retorno de procedimiento y restaurar la pila al estado que tenía justo antes de la llamada a procedimiento en curso.

El código que guarda el apuntador de marco viejo, ajusta el nuevo apuntador de marco y adelanta el apuntador de pila a fin de reservar espacio para las variables locales se llama **prólogo del procedimiento**. Cuando se sale del procedimiento, es preciso volver a limpiar la pila, es lo que se conoce como **epílogo del procedimiento**. Una de las características más importantes de cualquier computadora es cuán cortos y rápidos puede hacer al prólogo y al epílogo. Si son largos y lentos, las llamadas a procedimiento serán costosas. Los programadores que veneran la eficiencia aprenden a evitar la escritura de muchos procedimientos cortos y prefieren escribir programas grandes, monolíticos, no estructurados. Las instrucciones ENTER y LEAVE del Pentium II se diseñaron a modo de efectuar la mayor parte de las tareas del prólogo y epílogo de los procedimientos de manera eficiente. Desde luego, esas instrucciones tienen un modelo específico de la forma en que debe manejarse el apuntador de marco, y si el compilador tiene un modelo diferente, no pueden usarse.

Volvamos al problema de las Torres de Hanoi. Cada llamada de procedimiento añade un nuevo marco a la pila, y cada retorno de procedimiento elimina un marco de la pila. A fin de

ilustrar el uso de una pila al implementar procedimientos recursivos, seguiremos la pista a las llamadas, comenzando con

```
torres(3, 1, 3)
```

La figura 5-41(a) muestra la pila inmediatamente después de que se efectúa esta llamada. Lo primero que hace el procedimiento es probar si $n = 1$, y al descubrir que $n = 3$ sustituye k por su valor y efectúa la llamada

```
torres(2, 1, 2)
```

Una vez finalizada esa llamada, la pila está en el estado que se muestra en la figura 5-41(b) y el procedimiento vuelve a comenzar desde el principio (un procedimiento invocado siempre inicia en el principio). Esta vez la prueba de $n = 1$ falla también, por lo que se vuelve a sustituir k y se efectúa la llamada

```
torres(1, 1, 3)
```

La pila en ese momento tiene el aspecto que se muestra en la figura 5-41(c) y el contador de programa apunta al principio del procedimiento. Esta vez la prueba tiene éxito y se imprime una línea. Luego, el procedimiento regresa eliminando un marco de pila y restableciendo FP y SP a la figura 5-41(d). A continuación, el procedimiento continúa su ejecución en la dirección de retorno, que es la segunda llamada:

```
torres(1, 1, 2)
```

Esto añade un nuevo marco a la pila como se muestra en la figura 5-41(e). Se imprime otra línea; después del retorno se elimina un marco de la pila. Las llamadas a procedimientos continúan de esta manera hasta que termina de ejecutarse la llamada original y el marco de la figura 5-41(a) se elimina de la pila. Para entender mejor cómo funciona la recursión, se recomienda al lector simular la ejecución completa de

```
torres(3, 1, 3)
```

usando lápiz y papel.

5.6.3 Corrutinas

En la secuencia de invocación normal, hay una distinción clara entre el procedimiento que invoca y el procedimiento invocado. Considere un procedimiento *A*, que llama a un procedimiento *B* en la figura 5-42.

El procedimiento *B* calcula durante un rato y luego regresa a *A*. A primera vista podríamos pensar que esta situación es simétrica, porque ni *A* ni *B* son el programa principal; ambos son procedimientos. (El procedimiento *A* pudo haber sido llamado por el programa principal, pero eso no viene al caso.) Además, el control se transfiere primero de *A* a *B* –la llamada– y luego se transfiere de *B* a *A* –el retorno.

La asimetría surge del hecho de que cuando el control pasa de *A* a *B*, el procedimiento *B* comienza a ejecutarse al principio; cuando *B* regresa a *A*, la ejecución no inicia al principio de

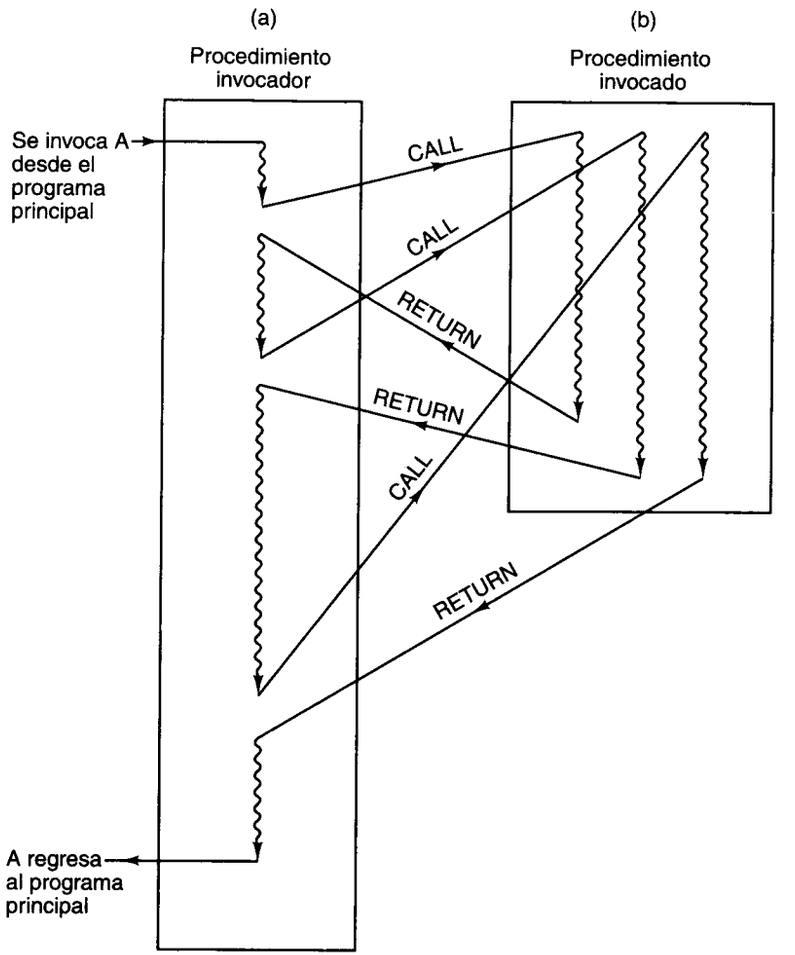


Figura 5-42. Cuando se invoca un procedimiento, la ejecución del procedimiento siempre inicia en el primer enunciado del procedimiento.

A, sino en el enunciado que sigue a la llamada. Si *A* trabaja durante un rato y luego vuelve a invocar a *B*, la ejecución inicia otra vez al principio de *B*, no en el enunciado que sigue al retorno anterior. Si, durante su ejecución, *A* llama a *B* muchas veces, *B* iniciará al principio en cada ocasión, mientras que *A* nunca vuelve a iniciar.

Esta diferencia se refleja en el método por el cual se transfiere el control entre *A* y *B*. Cuando *A* llama a *B*, usa la instrucción de llamada a procedimiento, la cual coloca la dirección de retorno (es decir, la dirección del enunciado que sigue a la llamada) en algún lugar útil; digamos, en el tope de la pila. Luego, la instrucción coloca la dirección de *B* en el contador de programa para completar la llamada. Cuando *B* regresa, no usa la instrucción de llamada, sino la de retorno, que simplemente saca la dirección de retorno de la pila y la coloca en el contador de programa.

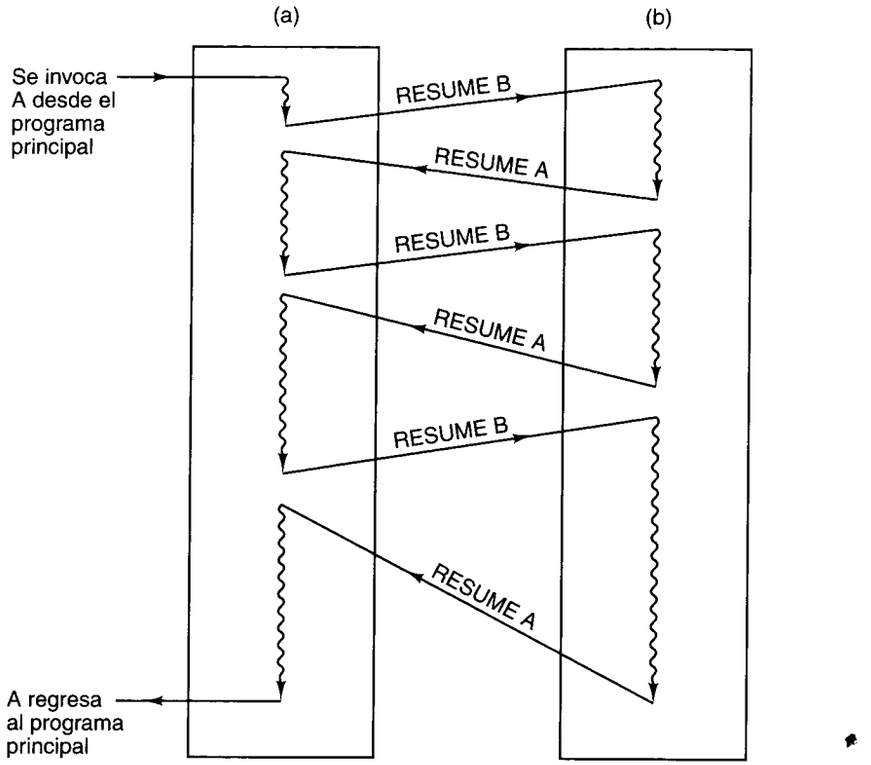


Figura 5-43. Cuando se reanuda una corrutina, la ejecución inicia en el enunciado donde se suspendió en la ocasión anterior, no al principio.

A veces resulta útil tener dos procedimientos, *A* y *B*, cada uno de los cuales invoca al otro como procedimiento, como se muestra en la figura 5-43. Cuando *B* regresa a *A*, salta al enunciado que sigue a la llamada a *B*, igual que antes. Cuando *A* transfiere el control a *B*, pasa no al principio (excepto la primera vez) sino al enunciado que sigue al "retorno" más reciente, es decir, la llamada más reciente de *A*. Dos procedimientos que funcionan de esta forma se llaman **corrutinas**.

Un uso común de las corrutinas es simular un procesamiento en paralelo con una sola CPU. Cada corrutina se ejecuta en seudoparalelo con las otras, como si tuviera su propia CPU. Este estilo de programación facilita la implementación de algunas aplicaciones, y también es útil para probar software que después se ejecutará en un multiprocesador.

Ni la llamada CALL normal ni la instrucción RETURN normal sirven para llamar corrutinas, ya que la dirección a la que se saltará proviene de la pila como en un retorno pero, a diferencia de un retorno, la llamada a la corrutina misma coloca una dirección de retorno en algún lugar para el regreso subsecuente a ella. Sería bueno contar con una instrucción que intercambiara el tope de la pila y el contador de programa. En detalle, esta instrucción primero sacaría la antigua dirección de retorno de la pila y la colocaría en un registro interno, luego metería el

contenido del contador de programa en la pila, y por último copiaría el registro interno en el contador de programa. Dado que se saca una palabra de la pila y se mete una palabra en ella, el apuntador de pila no cambia. Esta instrucción casi nunca existe, por lo que en la generalidad de los casos se tiene que simular con varias instrucciones.

5.6.4 Trampas

Una **trampa** es una especie de llamada a procedimiento automática iniciada por alguna condición causada por el programa, casi siempre una condición importante pero que raras veces ocurre. Un buen ejemplo es un desbordamiento. En muchas computadoras, si el resultado de una operación aritmética excede el número más grande que se puede representar, ocurre una trampa, lo que significa que el flujo de control se conmuta a alguna posición de memoria fija en lugar de continuar en sucesión. En esa posición fija hay una ramificación a un procedimiento llamado **manejador de trampas** que realiza alguna acción apropiada, como exhibir un mensaje de error. Si el resultado de una operación está dentro del intervalo, no ocurre ninguna trampa.

La característica fundamental de una trampa es que es iniciada por alguna condición especial causada por el programa mismo y detectada por el hardware o el microprograma. Un método alternativo para manejar el desbordamiento es tener un registro de un bit que se enciende cada vez que ocurre un desbordamiento. Un programador que quiera verificar si hubo o no desbordamiento deberá incluir una instrucción explícita "saltar si el bit de desbordamiento está encendido" después de cada instrucción aritmética. Esto es lento y desperdicia espacio. Las trampas ahorran tiempo y memoria en comparación con las verificaciones explícitas controladas por el programador.

La trampa podría implementarse con una prueba explícita efectuada por el microprograma (o el hardware). Si se detecta un desbordamiento, la dirección de la trampa se carga en el contador de programa. Lo que en un nivel es una trampa podría estar bajo el control del programa en un nivel más bajo. Hacer que el microprograma realice la prueba ahorra tiempo en comparación con una prueba del programador, porque se puede traslapar fácilmente con alguna otra cosa. También se ahorra memoria, porque sólo necesita ocurrir en un lugar, digamos el ciclo principal del microprograma, sin importar cuántas instrucciones aritméticas haya en el programa principal.

Algunas de las condiciones comunes que pueden causar trampas son el desbordamiento de punto flotante, el subdesbordamiento de punto flotante, el desbordamiento de enteros, una violación de la protección, un código de operación no definido, un desbordamiento de la pila, un intento por iniciar un dispositivo de E/S inexistente, un intento por traer una palabra de dirección impar, y una división entre cero.

5.6.5 Interrupciones

Las **interrupciones** son cambios en el flujo de control causados no por el programa en ejecución sino por otra cosa, casi siempre relacionada con E/S. Por ejemplo, un programa podría ordenar al disco que comience a transferir información, y que genere una interrupción apenas

haya terminado la transferencia. Al igual que la trampa, la interrupción detiene el programa en ejecución y transfiere el control a un manejador de interrupciones, el cual realiza alguna acción apropiada. Cuando el manejador de interrupciones termina, devuelve el control al programa interrumpido. El proceso interrumpido se debe reiniciar en el mismo estado exactamente en el que estaba cuando ocurrió la interrupción, lo que implica restaurar todos los registros internos a su estado previo a la interrupción.

La diferencia fundamental entre las trampas y las interrupciones es ésta: las *trampas* son sincrónicas con el programa y las *interrupciones* son asincrónicas. Si el programa se vuelve a ejecutar un millón de veces con las mismas entradas, las trampas volverán a ocurrir en el mismo punto en cada ocasión pero las interrupciones podrían variar, dependiendo, por ejemplo, del momento exacto en que una persona sentada ante una terminal pulsa el retorno de carro. La razón de que las trampas sean reproducibles y las interrupciones no lo sean es que las primeras son causadas directamente por el programa, mientras que las segundas son, si acaso, indirectamente causadas por el programa.

Para ver cómo funcionan en realidad las interrupciones, consideremos un ejemplo común: una computadora quiere enviar una línea de caracteres a una terminal. El software del sistema primero reúne en un buffer todos los caracteres que se van a escribir en la terminal, inicializa una variable global *apunt* que apunte al principio del buffer, y asigna a una segunda variable global *cuenta* el número de caracteres que se escribirán. Luego, el software verifica si la terminal está lista y, si así es, le envía el primer carácter (por ejemplo, usando registros como los de la figura 5-30). Después de iniciar la E/S, la CPU queda libre para ejecutar otro programa o hacer alguna otra cosa.

En algún momento, el carácter se exhibe en la pantalla. Ahora puede iniciarse la interrupción. En una forma simplificada, los pasos son los siguientes.

ACCIONES DEL HARDWARE

1. El controlador de dispositivo habilita una línea de interrupción en el bus del sistema para iniciar la secuencia de interrupción.
2. Tan pronto como la CPU está lista para manejar la interrupción, habilita una señal de reconocimiento de interrupción en el bus.
3. Cuando el controlador de dispositivo ve que su señal de interrupción ha sido reconocida, coloca un entero pequeño en las líneas de datos para identificarse. Este número se llama **vector de interrupción**.
4. La CPU quita el vector de interrupción del bus y lo guarda temporalmente.
5. Luego la CPU almacena el contador de programa y la palabra de estado del programa (PSW) en la pila.
6. La CPU localiza un nuevo contador de programa utilizando el vector de interrupción como índice de una tabla que está en la parte baja de la memoria. Si el contador de programa tiene 4 bytes, por ejemplo, el vector de interrupción *n* corresponde a la dirección $4n$. Este nuevo contador de programa apunta al inicio de la rutina de servicio de interrupción para el dispositivo que causó la interrupción. En muchos casos también se carga o modifica la PSW (por ejemplo, para inhabilitar interrupciones subsecuentes).

ACCIONES DEL SOFTWARE

7. Lo primero que hace la rutina de servicio de interrupción es guardar todos los registros para que puedan restaurarse después. Se pueden guardar en la pila o en una tabla del sistema.
8. En general, cada vector de interrupción es compartido por todos los dispositivos de un tipo dado, por lo que todavía no se sabe cuál terminal causó la interrupción. El número de terminal puede averiguarse leyendo algún registro de dispositivo.
9. Ahora puede leerse cualquier otra información que haya acerca de la interrupción, como códigos de situación.
10. Si ocurrió un error de E/S, se puede manejar aquí.
11. Se actualizan las variables globales *apunt* y *cuenta*. La primera se incrementa de modo que apunte al siguiente byte, y la segunda se decrementa para indicar que falta un byte menos que enviar a la salida. Si *cuenta* sigue siendo mayor que 0, hay más caracteres que exhibir. El carácter al que ahora apunta *apunt* se copia en el registro buffer de salida.
12. Si es necesario, se envía a la salida un código especial para indicar al dispositivo o al controlador de interrupciones que ya se procesó la interrupción.
13. Se restauran todos los registros que se guardaron.
14. Se ejecuta la instrucción RETURN FROM INTERRUPT (regresar de interrupción), que coloca la CPU otra vez en el modo y el estado en que estaba antes de la interrupción. La computadora continúa con lo que estaba haciendo.

Un concepto fundamental relacionado con las interrupciones es la **transparencia**. Cuando ocurre una interrupción se efectúan algunas acciones y se ejecuta cierto código, pero una vez que todo ha terminado la computadora deberá volver al mismo estado exactamente en el que estaba antes de la interrupción. Se dice que una rutina de interrupción que tiene esta propiedad es transparente. Si todas las interrupciones son transparentes es mucho más fácil entenderlas.

Si una computadora sólo tiene un dispositivo de E/S, las interrupciones siempre funcionan como acabamos de describirlas, y no tenemos más que decir al respecto. Sin embargo, una computadora grande podría tener muchos dispositivos de E/S, y varios podrían estar operando al mismo tiempo, tal vez para diferentes usuarios. Existe una probabilidad finita de que mientras se está ejecutando una rutina de interrupción, un segundo dispositivo de E/S quiera generar *su* interrupción.

Este problema se puede atacar de dos maneras. La primera es que todas las rutinas de interrupción inhabiliten las interrupciones subsecuentes como primer paso, aun antes de guardar los registros. Tal estrategia simplifica las cosas, ya que entonces las interrupciones se proce-

san en sucesión, pero puede causar problemas en el caso de dispositivos que no pueden tolerar mucho retraso. Por ejemplo, en una línea de comunicación de 9600 bps llegan caracteres cada 1042 μs, estemos o no listos para recibirlos. Si el primero todavía no se ha procesado cuando llega el segundo, se podrían perder datos.

Cuando una computadora tiene dispositivos de E/S para los que el tiempo es crítico, una mejor estrategia de diseño es asignar a cada dispositivo de E/S una prioridad, alta para los dispositivos críticos y baja para los más tolerantes de retrasos. La CPU deberá tener también prioridades, generalmente determinadas por un campo de la PSW. Cuando un dispositivo con prioridad *n* interrumpe, la rutina de interrupción también deberá ejecutarse con prioridad *n*.

Mientras una rutina de interrupción con prioridad *n* se está ejecutando, se hará caso omiso de cualquier intento de un dispositivo con más baja prioridad de causar una interrupción, hasta que la rutina termine y la CPU regrese para ejecutar un programa de usuario (prioridad 0). En cambio, las interrupciones de dispositivos con más alta prioridad se procesan sin retraso.

Ahora que las rutinas de interrupción mismas están sujetas a interrupciones, la mejor manera de organizar la administración es asegurarse de que todas las interrupciones sean transparentes. Consideremos un sencillo ejemplo de múltiples interrupciones. Una computadora tiene tres dispositivos de E/S, una impresora, un disco y una línea RS232, con prioridades 2, 4 y 5, respectivamente. En un principio ($t = 0$) se está ejecutando un programa de usuario. En $t = 10$ ocurre una interrupción de impresora. Se pone en marcha la rutina de servicio de interrupción (ISR, *Interrupt Service Routine*) de la impresora, como se muestra en la figura 5-44.

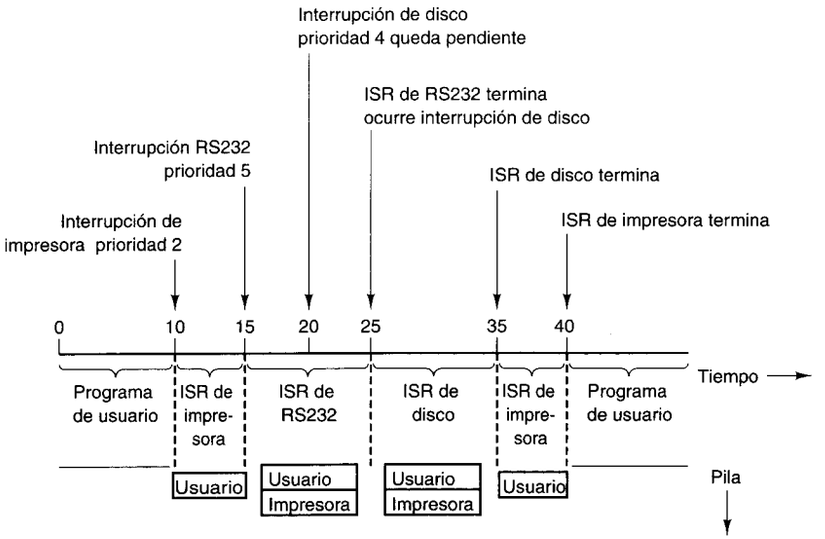


Figura 5-44. Secuencia temporal del ejemplo de múltiples interrupciones.

En $t = 15$ la línea RS232 necesita que la atiendan y genera una interrupción. Puesto que la línea RS232 tiene mayor prioridad (5) que la impresora (2), la interrupción ocurre. El estado de la máquina, que ahora está ejecutando la rutina de servicio de interrupción de la impresora, se almacena en la pila, y se inicia la rutina de servicio de interrupción de RS232.

Un poco después, en $t = 20$, el disco termina y quiere que lo atiendan. Sin embargo, su prioridad (4) es menor que la de la rutina de interrupción que se está ejecutando (5), por lo que el hardware de la CPU no reconoce la interrupción, y queda pendiente. En $t = 25$ termina la rutina de RS232, por lo que la máquina regresa al estado en el que estaba justo antes de que ocurriera la interrupción de RS232, es decir, ejecutando la rutina de servicio de interrupción de la impresora con prioridad 2. Tan pronto como la CPU cambia a prioridad 2, antes de que pueda ejecutarse siquiera una instrucción, se permite la entrada de la interrupción de disco con prioridad 4, y se ejecuta la rutina de servicio de disco. Cuando esta rutina termina, la rutina de la impresora puede continuar. Por último, en $t = 40$, todas las rutinas de servicio de interrupción han terminado y el programa de usuario continúa desde donde se quedó.

Desde el 8088, los chips de CPU de Intel han tenido dos niveles de interrupción (prioridades): enmascarables y no enmascarables. Las interrupciones no enmascarables por lo regular sólo se usan para avisar de posibles catástrofes, como errores de paridad de memoria. Todos los dispositivos de E/S usan la interrupción enmascarable.

Cuando un dispositivo de E/S emite una interrupción, la CPU usa el vector de interrupción como índice de una tabla de 256 entradas para encontrar la dirección de la rutina de servicio de interrupción. Las entradas de la tabla son descriptores de segmento de 8 bytes y la tabla puede iniciar en cualquier lugar de la memoria. Un registro global apunta a su principio.

Al tener un solo nivel de interrupción utilizable, la CPU no puede permitir que un dispositivo con mayor prioridad interrumpa una rutina de servicio de interrupción con prioridad intermedia y al mismo tiempo prohíba a un dispositivo con menor prioridad hacerlo. Para resolver este problema, las CPU de Intel ahora se usan normalmente con un controlador de interrupciones externo (por ejemplo, un 8259A). Cuando llega la primera interrupción, con una prioridad de *n*, la CPU se interrumpe. Si después llega una interrupción con mayor prioridad, el controlador de interrupciones interrumpe otra vez. Si la segunda interrupción es de menor prioridad, se detiene hasta que la primera termina. Para que este esquema funcione el controlador de interrupciones debe poder detectar el momento en que la rutina de servicio de interrupción en curso termina, por lo que la CPU debe enviarle un comando al terminar de procesar la interrupción en curso.

5.7 UN EJEMPLO DETALLADO: LAS TORRES DE HANOI

Ahora que hemos estudiado la ISA de tres máquinas, armemos todas las piezas y examinemos de cerca el mismo programa de ejemplo para las tres ISA. Nuestro ejemplo será el programa de las Torres de Hanoi. Ya dimos una versión en Java de este programa en la figura 5-40. En las secciones que siguen presentaremos programas en lenguaje ensamblador para el problema de las Torres de Hanoi.

Sin embargo, usaremos un pequeño truco. En vez de dar la traducción de la versión en Java, para el Pentium II y el UltraSPARC II daremos la traducción de una versión en C con objeto de evitar ciertos problemas de la E/S de Java. La única diferencia será la sustitución de la llamada a *println* en Java por el enunciado C estándar

```
printf("Pase un disco de %d a %d\n", i, j)
```

Para nuestros fines, la sintaxis de las cadenas de formato de *printf* no es importante (básicamente, la cadena se imprime literalmente excepto que %d significa imprimir el siguiente entero en formato decimal). Lo único importante aquí es que el procedimiento se invoque con tres parámetros: una cadena de formato y dos enteros.

La razón por la que usamos la versión C para el Pentium II y el UltraSPARC II es que la biblioteca de E/S de Java no está disponible en forma nativa para estas máquinas, mientras que la biblioteca de C sí lo está. Con la JVM usaremos la versión en Java. La diferencia es mínima, y sólo afecta el enunciado de impresión.

5.7.1 Las Torres de Hanoi en lenguaje ensamblador del Pentium II

La figura 5-45 presenta una posible traducción de la versión en C de las Torres de Hanoi para el Pentium II. En su mayor parte, el programa es relativamente sencillo. Se usa el registro EBP como apuntador de la pila. Las primeras dos palabras se usan para enlace, así que el primer parámetro real, *n* (o *N* aquí, ya que MASM no distingue entre mayúsculas y minúsculas), está en EBP + 8, seguido de *i* y *j* en EBP + 12 y EBP + 16, respectivamente. La variable local, *k*, está en EBP + 20.

Lo primero que hace el procedimiento es establecer el nuevo marco al final del anterior. Esto lo hace copiando ESP en el apuntador de marco, EBP. Luego compara *n* con 1, saltando a la cláusula else si $n > 1$. El código then mete tres valores en la pila: la dirección de la cadena de formato, *i* y *j*, y luego se llama a sí mismo.

Los parámetros se meten en orden inverso, lo cual es necesario para los programas en C a fin de colocar el apuntador a la cadena de formato en el tope de la pila. Puesto que *printf* tiene un número variable de parámetros, si los parámetros se metieran después *printf* no sabría qué tan abajo en la pila está la cadena de formato.

Después de la llamada, se suma 12 a ESP para eliminar los parámetros de la pila. Desde luego, los parámetros no se borran realmente de la memoria, pero el ajuste de ESP los vuelve inaccesibles a través de las operaciones de pila normales.

La cláusula else, que inicia en *L1*, es sencilla: primero calcula $6 - i - j$ y guarda este valor en *k*. Sean cuales sean los valores de *i* y *j*, la tercera varilla siempre es $6 - i - j$; guardarla en *k* ahorra el trabajo de recalcularla la segunda vez.

Luego, el procedimiento se llama a sí mismo tres veces, con diferentes parámetros en cada ocasión. Después de cada llamada, la pila se asea. Eso es todo.

Los procedimientos recursivos provocan confusión en mucha gente al principio, pero cuando se estudian en este nivel son de lo más sencillo. Lo único que sucede es que se meten los parámetros en la pila y se invoca el procedimiento.

5.7.2 Las Torres de Hanoi en lenguaje ensamblador del UltraSPARC II

Vamos a intentarlo otra vez, pero ahora para el UltraSPARC II. El código se presenta en la figura 5-46. Dado que el código del UltraSPARC II es muy difícil de entender, aun en comparación con otros códigos de ensamblador y aun después de mucha práctica, nos hemos tomado la libertad de definir unos cuantos símbolos al principio para hacerlo más claro.


```
        .586                        ; compilar para Pentium (no 8088, etc.)
.MODEL FLAT
PUBLIC _torres                      ; exportar 'torres'
EXTERN _printf:NEAR                 ; importar printf
.CODE
_torres: PUSH EBP                   ; guardar EBP (apuntador a marco)
        MOV EBP, ESP                ; fijar nuevo apunt. a marco arriba de ESP
        CMP [EBP+8], 1              ; si (n == 1)
        JNE L1                      ; ramificar si n no es 1
        MOV EAX, [EBP+16]           ;printf(" ...", i, j);
        PUSH EAX                    ; observe que los parámetros i, j y la cadena
        MOV EAX, [EBP+12]           ; de formato se empilan en orden inverso.
        PUSH EAX                    ; Ésta es la convención de llamada de C
        PUSH OFFSET FLAT:format     ; offset flat se refiere a la direc. de format
        CALL_pintf                  ; llamar a printf
        ADD ESP, 12                 ; quitar parámetros de la pila
        JMP Done                    ; ya terminamos
L1:     MOV EAX, 6                  ; iniciar k = 6 – i – j
        SUB EAX, [EBP+12]           ; EAX = 6 – i
        SUB EAX, [EBP+16]           ; EAX = 6 – i – j
        MOV [EBP+20], EAX           ; k = EAX
        PUSH EAX                    ; iniciar torres(n – 1, i, k)
        MOV EAX, [EBP+12]           ; EAX = i
        PUSH EAX                    ; apilar i
        MOV EAX, [EBP+8]            ; EAX = n
        DEC EAX                     ; EAX = n – 1
        PUSH EAX                    ; apilar n – 1
        CALL _torres                ; llamar a torres(n –1 , i, 6 – i – j)
        ADD ESP, 12                 ; quitar parámetros de la pila
        MOV EAX, [EBP+16]           ; iniciar torres(1, i, j)
        PUSH EAX                    ; apilar j
        MOV EAX, [EBP+12]           ; EAX = i
        PUSH EAX                    ; apilar i
        PUSH 1                      ; apilar 1
        CALL _torres                ; llamar torres(1, i, j)
        ADD ESP, 12                 ; quitar parámetros de la pila
        MOV EAX, [EBP+12]           ; iniciar torres(n – 1, 6 – i – j, i)
        PUSH EAX                    ; apilar i
        MOV EAX, [EBP+20]           ; apilar 20
        PUSH EAX                    ; apilar k
        MOV EAX, [EBP+8]            ; EAX = n
        DEC EAX                     ; EAX = n – 1
        PUSH EAX                    ; apilar n – 1
        CALL _torres                ; llamar torres(n – 1, 6 – i – j, i)
        ADD ESP, 12                 ; ajustar apuntador de pila
Done:   LEAVE                       ; prepararse para salir
        RET 0                       ; regresar al invocador
.DATA
format  DB "Pase un disco de %d a %d\n"   ; cadena de formato
END
```

Figura 5-45. Las Torres de Hanoi para el Pentium II.

Para que funcione, el programa se tiene que procesar con otro programa llamado *cpp*, el preprocesador de C, antes de ensamblarlo. Además, hemos usado minúsculas aquí porque el ensamblador de UltraSPARC II lo exige (por si algún lector desea introducir el programa y ejecutarlo).

En cuanto al algoritmo, la versión para UltraSPARC II es idéntica a la versión para el Pentium II. Ambas prueban n al principio y saltan a la cláusula else si $n > 1$. La complejidad de la versión UltraSPARC II se debe principalmente a ciertas propiedades de la ISA.

Por principio, el UltraSPARC II tiene que pasar la dirección de la cadena de formato a *printf*, pero la máquina no puede limitarse a poner la dirección en el registro que contiene el parámetro de salida porque no hay forma de colocar una constante de 32 bits en un registro con una sola instrucción. Se requieren dos instrucciones para hacerlo, *SETH* y OR.

Otra cosa que debemos notar es que no es necesario ajustar la pila después de la llamada porque la instrucción RESTORE del final del procedimiento ajusta la ventana de registro. La capacidad para colocar los parámetros de salida en registros y no tener que ir a la memoria aumenta de manera considerable la rapidez si la profundidad de las llamadas no es excesiva, pero en general es probable que la ventaja de contar con el mecanismo de ventanas de registros no justifique el aumento en la complejidad que implica.

Observe ahora la instrucción NOP que sigue a la ramificación a *Done*. Se trata de una ranura de retraso. Esta instrucción siempre se ejecutará, aunque sigue a una instrucción de ramificación condicional. El problema es que el UltraSPARC II tiene un conducto profundo, y para cuando el hardware descubre que enfrenta una ramificación, la siguiente instrucción prácticamente ya terminó de ejecutarse. Bienvenido al Mundo Maravilloso de la Programación RISC.

Esta característica tan divertida también se extiende a las llamadas a procedimientos. Observe la primera llamada a *torres* en la cláusula else: coloca $n - 1$ en %o0 y en %o1, pero efectúa la llamada a *torres* antes de colocar el último parámetro. En el Pentium II, primero se pasan los parámetros y luego se hace la llamada. Aquí, primero se pasan algunos parámetros, luego se hace la llamada, y luego se pasa el último parámetro. Para cuando la máquina se da cuenta de que enfrenta una instrucción *CALL*, la instrucción que le sigue ya está en un nivel tan profundo del conducto que tiene que ejecutarse. Entonces, ¿por qué no usar la ranura de retraso para pasar el último parámetro? Incluso si la primera instrucción del procedimiento invocado usa ese parámetro, el valor estará en su lugar a tiempo.

Por último, en *Done*, vemos que la instrucción RET también tiene una ranura de retraso, la cual aprovechamos para la instrucción RESTORE, que incrementa CWP a fin de restaurar la ventana de registros a la posición que el invocador espera.

5.7.3 Las Torres de Hanoi en lenguaje ensamblador de la JVM

El código de ensamblador JVM para la versión en Java de las Torres de Hanoi se da en la figura 5-47. La solución es relativamente sencilla, con excepción de la E/S. Este programa se generó con el compilador de Java, se desensambló a lenguaje ensamblador simbólico, se editó para hacerlo más comprensible y se le añadieron comentarios.


```
#define N %i0                   /* N es el parámetro de entrada 0 */
#define I %i1                   /* I es el parámetro de entrada 1 */
#define J %i2                   /* J es el parámetro de entrada 2 */
#define K %l0                   /* K es la variable local 0 */
#define Param0 %o0              /* Param0 es el parámetro de salida 0 */
#define Param1 %o1              /* Param1 es el parámetro de salida 1 */
#define Param2 %o2              /* Param2 es el parámetro de salida 2 */
#define Scrach %l1              /* por cierto, cpp usa la convención de comentarios de C */
        .proc 04
        .global torres

torres: save %sp, -112, %sp
        cmp N, 1                ! si (n == 1)
        bne Else                ! si (n != 1), ir a Else

        sethi %hi(format), Param0       ! printf("Pase un disco de %d a %d\n", i, j)
        or Param0, %lo(format), Param0  ! Param0 = dirección de cadena de formato
        mov I, Param1           ! Param1 = i
        call printf             ! llamar printf ANTES de preparar parámetro 2 (j)
        mov J, Param2           ! usar ranura de retraso para preparar parámetro 2
        b Done                  ! ya terminamos
        nop                     ! llenar ranura de retraso

Else:   mov 6, K                ! iniciar k = 6 – i – j
        sub K, J, K             ! k = 6 – j
        sub K, I, K             ! k = 6 – i – j

        add N, -1, Scratch      ! iniciar torres(n – 1, i, k)
        mov Scratch, Param0     ! Scratch = N – 1
        mov I, Param1           ! parámetro 1 = i
        call torres             ! llamar torres ANTES DE preparar parámetro 2 (k)
        mov K, Param2           ! usar ranura de retraso para preparar parámetro 2

        mov 1, Param0           ! iniciar torres(1, i, j)
        mov I, Param1           ! parámetro 1 = i
        call torres             ! llamar torres ANTES DE preparar parámetro 2 (j)
        mov J, Param2           ! parámetro 2 = j

        mov Scratch, Param0     ! iniciar torres(n – 1, k, j)
        mov K, Param1           ! parámetro 1 = k
        call torres             ! llamar torres ANTES DE preparar parámetro 2 (j)
        movJ, Param2            ! parámetro 2 = j

Done:   ret                     ! regresar
        restore                 ! usar ranura de retraso para restaurar ventanas

format; .asciz "Pase un disco de %d a %d\n"
```

Figura 5-46. Las Torres de Hanoi para el UltraSPARC II.

El compilador JVM mantiene los tres parámetros, *n*, *i* y *j* en las variables locales 0, 1 y 2, respectivamente. La variable local, *k*, se guarda en la 3. Podemos acceder a estas cuatro variables locales usando el código de operación de un byte correspondiente, como ILOAD_0. El resultado es que la versión binaria de este programa en JVM es muy corta (67 bytes).

Lo primero que el programa hace es meter *n* y la constante 1 en la pila, y luego usa IF_ICMPNE para compararlos. IF_ICMPNE es la prueba inversa de nuestra instrucción IF_ICMPEQ de IJVM: desempila dos operandos y ramifica si son diferentes.

Si los operandos son iguales, la ejecución continúa con la siguiente instrucción. Las 13 instrucciones que siguen asignan espacio para un buffer de cadena y construyen en él una cadena que al final se pasa a *println* para imprimirla. Una vez terminada la impresión, el procedimiento regresa.

En pocas palabras, lo que estas 13 instrucciones están haciendo es asignar espacio para un buffer de cadena en el montículo y llenarlo. La instrucción GETSTATIC usa un índice para obtener la palabra 13 de la reserva de constantes, la cual contiene un apuntador a un descriptor del buffer de cadena. La instrucción NEW usa este descriptor para asignar espacio para el buffer de cadena en el montículo. Las siguientes 11 instrucciones concatenan las dos cadenas y dos enteros para formar una sola cadena en este buffer y pasarla a *println*, que la imprime.

Si *n* no es 1, el control pasa a *L1*. Aquí se calcula *k* directamente usando aritmética de pila. Luego se efectúan las tres llamadas, una tras otra.

JVM cuenta con diversas instrucciones para invocar procedimientos. Aquí el compilador usó tres distintas. Todas ellas tienen un operando de dos bytes que funciona como índice de la reserva de constantes para encontrar un apuntador a un descriptor que contiene toda la información acerca del procedimiento que se invocará. Las constantes como #10, #11, etc., son índices para la reserva de constantes.

Las diferencias entre los diferentes tipos de llamadas a procedimientos tienen que ver con diferentes optimaciones. Se usa la instrucción INVOKESTATIC para invocar métodos estáticos escritos por el usuario, como *torres*. En contraste, INVOKESPECIAL sirve para invocar métodos de inicialización, métodos privados y métodos de la superclase de la clase actual. Por último, INVOKEVIRTUAL se usa para llamadas internas (de biblioteca).

5.8 EL IA-64 DE INTEL

Intel está llegando rápidamente a un punto en el que se ha exprimido hasta la última gota de jugo que puede sacarse a la ISA IA-32 y a la línea de procesadores Pentium II. Los nuevos modelos todavía pueden sacar ventaja de los adelantos en la tecnología de fabricación, lo que significa transistores más pequeños (y por ende velocidades de reloj más altas). Sin embargo, encontrar nuevos trucos para hacer más rápida la implementación es cada vez más difícil, y las restricciones impuestas por la ISA IA-32 son un obstáculo cada día mayor.

La única solución real consiste en abandonar la IA-32 como línea principal de desarrollo y cambiar a una ISA totalmente distinta. De hecho, esto es lo que Intel piensa hacer. La nueva arquitectura, desarrollada en forma conjunta por Intel y Hewlett-Packard, se llama **IA-64**, y es una máquina de 64 bits de principio a fin. En los próximos años se espera la aparición de toda una serie de CPU que implementen esta arquitectura. La primera implementación, cuyo


```
        ILOAD_0          // local 0 = n; apilar n
        ICONST_1         // apilar 1
        IF_ICMPNE L1     // si (n != 1), ir a L1

        GETSTATIC #13    // n == 1; este código maneja el enunc. println
        NEW #7           // asignar buffer para la cadena a construir
        DUP              // duplicar apuntador al buffer
        LDC #2           // apilar apunt. a cadena "pase un disco de "
        INVOKESPECIAL #10 // copiar la cadena en el buffer
        ILOAD_1          // apilar i
        INVOKEVIRTUAL #11 // convertir i en cadena y anexar a nuevo buffer
        LDC #1           // apilar apuntador a cadena " a "
        INVOKEVIRTUAL #12 // anexar esta cadena al buffer
        ILOAD_2          // apilar j
        INVOKEVIRTUAL #11 // convertir j en cadena y anexar a buffer
        INVOKEVIRTUAL #15 // conversión en cadena
        INVOKEVIRTUAL #14 // invocar println
        RETURN           // regresar de torres

L1:     BIPUSH 6         // parte Else: calcular k = 6 – i – j
        ILOAD_1          // local 1 = i; apilar i
        ISUB             // tope de pila = 6 – i
        ILOAD_2          // local 2 = j; apilar j
        ISUB             // tope de pila = 6 – i – j
        ISTORE_3         // local 3 = k = 6 – i – j; la pila está vacía

        ILOAD_0          // iniciar trabajos de torres(n – 1, i, k); apilar n
        ICONST_1         // apilar 1
        ISUB             // tope de pila = n – 1
        ILOAD_1          // apilar i
        ILOAD_3          // apilar k
        INVOKESTATIC #16 // invocar torres(n – 1, 1, k)

        ICONST_1         // iniciar trabajos de torres(1, i, k); apilar 1
        ILOAD_1          // apilar i
        ILOAD_2          // apilar j
        INVOKESTATIC #16 // invocar torres(1, i, j)

        ILOAD_0          // iniciar trabajos de torres(n – 1, k, j); apilar n
        ICONST_1         // apilar 1
        ISUB             // tope de pila = n – 1
        ILOAD_3          // apilar k
        ILOAD_2          // apilar j
        INVOKESTATIC #16 // invocar torres(n – 1, k, j)
        RETURN           // regresar de torres
```

Figura 5-47. Las Torres de Hanoi para JVM.

nombre en código es **Merced** (según el río californiano favorito de alguien), es una CPU de extremo superior, pero tarde o temprano aparecerá toda una gama de CPU, desde servidores de extremo superior hasta modelos económicos para escritorio.

Dada la importancia que para la industria de las computadoras tiene todo lo que Intel hace, vale la pena dedicar una mirada a la arquitectura IA-64 (y por implicación al Merced).

Sin embargo, dejando eso a un lado, los investigadores en este campo ya conocen bien las ideas fundamentales en que se apoya la IA-64, y bien podrían aparecer en otros diseños. De hecho, algunas de ellas ya están presentes en diversas formas en sistemas (experimentales). En las secciones que siguen examinaremos la naturaleza de los problemas que Intel enfrenta, el modelo que la IA-64 adoptó para resolverlos, el funcionamiento de algunas de las ideas fundamentales, y lo que podría suceder.

5.8.1 El problema del Pentium II

La raíz de todos los problemas es el hecho ineludible de que la IA-32 es una ISA antigua cuyas propiedades son totalmente inadecuadas para la tecnología actual. Es una ISA CISC con instrucciones de longitud variable y una multitud de formatos distintos que son difíciles de decodificar rápidamente sobre la marcha. La tecnología actual funciona de manera óptima con las ISA RISC que tienen una sola longitud de instrucción y un código de operación de longitud fija fácil de decodificar. Las instrucciones de la IA-32 pueden descomponerse en microoperaciones tipo RISC en el momento de la ejecución, pero ello requiere hardware (área de chip), consume tiempo y aumenta la complejidad del diseño. Ése es el primer *strike*.*

También, la IA-32 es una ISA de dos direcciones orientada hacia la memoria. Casi todas las instrucciones hacen referencia a la memoria, y pocos programadores dudan en hacer referencias a la memoria a diestra y siniestra. La tecnología actual favorece a las ISA de carga/almacenamiento que sólo hacen referencia a la memoria para obtener los operandos y colocarlos en registros, pero por lo demás realizan todos sus cálculos con instrucciones de tres direcciones que manejan registros. Y en vista del aumento mucho mayor de las velocidades de reloj de las CPU en comparación con las de memoria, el problema va a empeorar con el tiempo. Éste es el segundo *strike*.

Además, la IA-32 tiene un conjunto de registros pequeño e irregular. Esto no sólo hace que los compiladores parezcan contorsionistas, sino que el número tan reducido de registros de propósito general (cuatro o seis, dependiendo de cómo se cuenten ESI y EDI) obliga a verter a la memoria continuamente los resultados intermedios, lo que genera referencias a la memoria adicionales aun cuando ni siquiera se necesitan lógicamente. Ése es el tercer *strike*. La IA-32 está *out*.

Pasemos ahora a la segunda entrada (*inning*). La escasez de registros causa muchas dependencias, sobre todo dependencias WAR innecesarias porque los resultados tienen que colocarse en algún lado y no hay registros extra disponibles. Para subsanar la falta de registros, la implementación tiene que efectuar cambios de nombres internamente –un parche horrible– a registros secretos dentro del buffer de reordenamiento. Para evitar bloqueos con demasiada frecuencia cuando hay fallos de caché, las instrucciones tienen que ejecutarse en desorden. Sin embargo, la semántica de la IA-32 especifica interrupciones precisas, por lo que las instrucciones en desorden tienen que retirarse en orden. Todo esto requiere una gran cantidad de hardware muy complejo. Cuarto *strike*.

Para poder efectuar todo este trabajo rápidamente se requiere una fila de procesamiento muy grande (de 12 etapas). A su vez, la fila de procesamiento implica que las instrucciones ingresan en él 11 ciclos antes de que terminen. Por ello, es indispensable una predicción de ramificaciones muy exacta para asegurarse de que las instrucciones correctas ingresen en el

*El autor está usando aquí la terminología de un juego de béisbol. *(N. del T.)*

conducto. Dado que una predicción errónea obliga a vaciar la fila de procesamiento, lo cual es una operación muy costosa, los errores de predicción no tienen que ser muy frecuentes para que el desempeño sufra una degradación sustancial. Quinto *strike*.

Para aliviar los problemas que causan las predicciones erróneas, el procesador necesita efectuar ejecución especulativa, con todos los problemas que conlleva, sobre todo cuando referencias a la memoria en la rama equivocada causan una excepción. Sexto *strike*.

No vamos a jugar el partido de béisbol completo aquí, pero a estas alturas ya debe ser obvio que existe un problema real. Y ni siquiera hemos mencionado el hecho de que las direcciones de 32 bits de la IA-32 limitan los programas individuales a 4 GB de memoria, lo cual es un problema cada vez más grave en los servidores de extremo superior.

En síntesis, la situación de la IA-32 puede compararse favorablemente con el estado de la mecánica celeste justo antes de Copérnico. La teoría que entonces dominaba la astronomía era que la Tierra estaba fija e inmóvil en el espacio y que los planetas se movían a su alrededor en círculos con epiciclos. Sin embargo, a medida que mejoraban las observaciones y era posible detectar más claramente desviaciones respecto a este modelo, se añadieron epiciclos a los epiciclos hasta que el modelo completo se vino abajo por su propia complejidad interna.

Intel está en un aprieto comparable. Una fracción enorme de todos los transistores del Pentium II se dedica a descomponer instrucciones CISC, averiguar qué puede hacerse en paralelo, resolver conflictos, hacer predicciones, reparar las consecuencias de predicciones equivocadas y otras tareas contables, con lo que queda una cantidad sorprendentemente reducida de transistores para efectuar el trabajo real que el usuario solicitó. La conclusión a la que está siendo llevado inexorablemente Intel es la única conclusión razonable: tirar todo (la IA-32) a la basura y comenzar otra vez desde cero (IA-64).

5.8.2 El modelo IA-64: Computación con instrucciones explícitamente paralelas

El punto de partida para la IA-64 fue un procesador RISC de 64 bits del extremo superior (del cual el UltraSPARC II es uno de muchos ejemplos actuales). Puesto que el IA-64 se diseñó junto con Hewlett-Packard, es casi seguro que la arquitectura PA-RISC de HP haya sido la base real, pero todas las máquinas RISC de vanguardia son lo bastante parecidas como para que la descripción que dimos del UltraSPARC II aplique en muchos aspectos. El Merced es un procesador de modo dual, que puede ejecutar programas tanto IA-32 como IA-64, pero en lo que sigue nos concentraremos únicamente en la parte IA-64.

La arquitectura IA-64 es una arquitectura de carga/almacenamiento con direcciones de 64 bits y registros de 64 bits de anchura. Hay 64 registros generales que los programas IA-64 pueden usar (con registros adicionales para los programas IA-32). Todas las instrucciones tienen el mismo formato fijo: un código de operación, dos campos de registro fuente de 6 bits, un campo de registro de destino de 6 bits y otro campo de 6 bits que explicaremos más adelante. Casi todas las instrucciones aceptan dos operandos en registros, realizan algún cálculo con ellos, y colocan el resultado en el registro de destino. Hay muchas unidades funcionales que realizan diferentes operaciones en paralelo. Hasta aquí, nada fuera de lo común. Casi todas las máquinas RISC tienen una arquitectura muy parecida.

Lo que es inusitado es la idea de un haz de instrucciones relacionadas. Las instrucciones vienen en grupos de tres, llamados **haces**, como se muestra en la figura 5-48. Cada haz de 128 bits contiene tres instrucciones de formato fijo de 40 bits y una plantilla de 8 bits. Los haces pueden encadenarse mediante un bit de fin de haz, por lo que puede haber más de tres instrucciones presentes en un haz. La plantilla contiene información acerca de cuáles instrucciones se pueden ejecutar en paralelo. Este esquema, más la abundancia de registros, permite al compilador aislar bloques de instrucciones y decirle a la CPU que se pueden ejecutar en paralelo. Así, el compilador debe reordenar las instrucciones, detectar dependencias, asegurarse de que haya unidades funcionales disponibles, etc., en lugar de que lo haga el hardware. La idea es que al exponer el funcionamiento interno de la máquina y pedir a los escritores de compiladores que se aseguren de que cada haz consta de instrucciones compatibles, la tarea de planificar las instrucciones RISC se traslada del hardware (en el momento de la ejecución) al compilador (en el momento de la compilación). Por esta razón, el modelo se llama **computación con instrucciones explícitamente paralelas** (**EPIC**, *Explicitly Parallel Instruction Computing*).

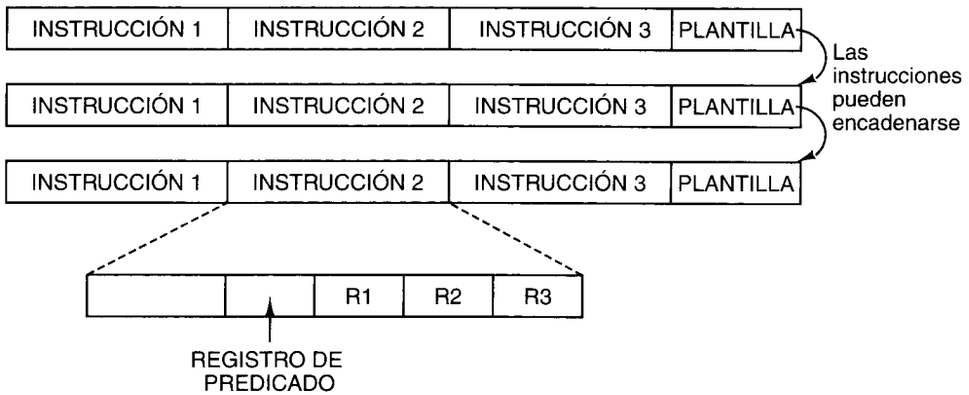


Figura 5-48. La IA-64 se basa en haces de tres instrucciones.

Planificar las instrucciones en el momento de la compilación tiene varias ventajas. Primera, puesto que ahora el compilador está haciendo todo el trabajo, el hardware puede ser mucho más sencillo, lo que ahorra millones de transistores para otras funciones útiles, como cachés de nivel 1 más grandes. Segunda, para un programa dado la planificación sólo tiene que realizarse una vez, en el momento de la compilación. Tercera, puesto que el compilador efectúa todo el trabajo, un proveedor de software puede usar un compilador que dedique horas a optimar su programa y así beneficiar a todos los usuarios cada vez que se ejecute el programa.

La idea de haces de instrucciones puede servir para crear toda una familia de procesadores. En las CPU de extremo inferior se podría emitir un haz cada ciclo de reloj. La CPU podría esperar hasta que todas las instrucciones hayan terminado antes de emitir el siguiente haz. En las CPU del extremo alto podría ser posible emitir varios haces durante el mismo ciclo de reloj, como en los diseños superescalares actuales. Además, en esas CPU, el procesador podría comenzar a emitir instrucciones de un nuevo haz antes de que terminen todas las instrucciones del haz anterior. Desde luego, tendría que verificar si los registros y la unidad funcional requeridos están disponibles, pero no tiene que verificar si otras instrucciones de su propio haz están en conflicto con la instrucción actual porque el compilador ya garantizó que no sucedería.

5.8.3 Predicación

Otra característica importante de IA-64 es el novedoso modo en que maneja las ramificaciones condicionales. Si hubiera alguna forma de deshacerse de casi todas ellas, las CPU podrían ser mucho más sencillas y rápidas. A primera vista podría parecer que es imposible deshacerse de las ramificaciones condicionales porque los programas están llenos de enunciados if. No obstante, la IA-64 usa una técnica llamada **predicación** que puede reducir su número considerablemente (August *et al.*, 1998; Hwu, 1998). A continuación la describiremos brevemente.

En las máquinas actuales, todas las instrucciones son incondicionales en el sentido de que cuando la CPU llega a una instrucción simplemente la ejecuta. No hay ningún debate interno del tipo de "Ejecutar o no ejecutar, ¡he ahí el dilema!". En cambio, en una arquitectura predicativa, las instrucciones contienen condiciones (predicados) que indican cuándo deben ejecutarse y cuándo no. Este cambio de paradigma, de instrucciones incondicionales a instrucciones predicativas, es lo que permite deshacerse de (muchas) ramificaciones condicionales. En lugar de tener que escoger entre una sucesión de instrucciones incondicionales y otra sucesión de instrucciones incondicionales, todas las instrucciones se fusionan en una sola sucesión de instrucciones predicativas, empleando diferentes predicados para diferentes instrucciones.

Para ver cómo funciona la predicación, comencemos con el sencillo ejemplo de la figura 5-49, que muestra la **ejecución condicional**, una precursora de la predicación. En la figura 5-49(a) vemos un enunciado if. En la figura 5-49(b) vemos su traducción a tres instrucciones: una comparación, una ramificación condicional y una instrucción de traslado.

```
if (R1 == 0)                CMP R1,0                CMOVZ R2,R3,R1
    R2 = R3;                BNE L1
                            MOV R2,R3
                        L1:
```

(a) (b) (c)

Figura 5-49. (a) Un enunciado if. (b) Código de ensamblador genérico para (a). (c) Una instrucción condicional.

En la figura 5-49(c) nos deshacemos de la ramificación condicional empleando una instrucción nueva, CMOVZ, que es un traslado condicional. Lo que esta instrucción hace es verificar si el tercer registro, R1, es cero. Si lo es, se copia R3 en R2; si no, no se hace nada.

Una vez que tenemos una instrucción que puede copiar datos cuando algún registro es cero, no es difícil diseñar una instrucción que pueda copiar datos cuando algún registro no

sea cero, digamos CMOVN. Si contamos con estas dos instrucciones, ya no nos falta mucho para lograr una ejecución plenamente condicional. Imagine un enunciado if que tiene varias asignaciones en la parte then y varias asignaciones en la parte else. El enunciado completo se puede traducir en código que pone en cero algún registro si la condición no se cumple y le asigna otro valor si la condición se cumple. Después de la preparación de los registros, las asignaciones de la parte then se pueden compilar a una secuencia de instrucciones CMOVN, y las de la parte else, a una secuencia de instrucciones CMOVZ.

Todas estas instrucciones (la preparación de registros, las CMOVN y las CMOVZ) forman un solo bloque básico sin ramificaciones condicionales. Hasta es posible reordenar las instrucciones, sea que lo haga el compilador (que incluso podría colocar las asignaciones antes de la prueba) o durante la ejecución. El único requisito es que la condición tiene que conocerse en el momento en que hay que retirar las instrucciones condicionales (cerca del final de la fila de procesamiento). En la figura 5-50 se da un ejemplo sencillo con una parte then y una parte else.

```
if (R1 == 0) {          CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;            BNE L1            CMOVZ R4,R5,R1
    R4 = R5;            MOV R2,R3         CMOVN R6,R7,R1
} else {                MOV R4,R5         CMOVN R8,R9,R1
    R6 = R7;            BR L2
    R8 = R9;        L1: MOV R6,R7
}                       MOV R8,R9
                    L2:
     (a)                    (b)                  (c)
```

Figura 5-50. (a) Un enunciado if. (b) Código de ensamblador genérico para (a). (c) Ejecución condicional.

Aunque aquí sólo hemos mostrado instrucciones condicionales muy sencillas (tomadas del Pentium II, de hecho), en la IA-64 todas las instrucciones son predicativas. Esto implica que se puede hacer condicional la ejecución de cada instrucción. El campo de 6 bits extra que mencionamos antes selecciona uno de los 64 registros de predicado de un bit. Así, un enunciado if se compila para dar código que pone en 1 uno de los registros de predicado si la condición se cumple, y lo pone en 0 si no se cumple. De forma simultánea y automática, el código asigna el valor opuesto a otro registro de predicado. Mediante la predicación, las instrucciones de máquina que forman las cláusulas then y else se fusionan en un solo flujo de instrucciones, de las cuales las primeras usan el predicado y las segundas usan su inverso.

Aunque sencillo, el ejemplo de la figura 5-51 ilustra la idea en la que se basa el uso de la predicación para eliminar ramificaciones. La instrucción CMPEQ compara dos registros y pone en 1 el registro de predicado P4 si son iguales, y en 0 si son diferentes; además, asigna el valor opuesto a un registro apareado, digamos P5. Ahora las instrucciones de las partes if y then se pueden poner una tras otra, cada una condicionada a algún registro de predicado (que se indica entre paréntesis angulares). Se puede colocar aquí cualquier código, siempre que cada instrucción tenga el predicado correcto.


```
if (R1 == R2)           CMP R1,R2          CMPEQ R1,R2,P4
    R3 = R4 + R5;       BNE L1             <P4> ADD R3,R4,R5
else                    MOX R3,R4          <P5> SUB R6,R4,R5
    R6 = R4 – R5        ADD R3,R5
                        BR L2
                    L1: MOV R6,R4
                        SUB R6,R5
                    L2:
     (a)                    (b)                  (c)
```

Figura 5-51. (a) Un enunciado if. (b) Código de ensamblador genérico para (a). (c) Ejecución predicativa.

En la IA-64, esta idea se lleva al extremo, con instrucciones de comparación para ajustar los registros de predicado además de instrucciones aritméticas y de otro tipo cuya ejecución depende de algún registro de predicado. Las instrucciones predicativas se pueden insertar en la fila de procesamiento una tras otra, sin paros ni problemas. Es por ello que son tan útiles.

La forma en que funciona realmente la predicación en la IA-64 es que todas las instrucciones se ejecutan realmente. Al final de la fila de procesamiento, cuando llega el momento de retirar una instrucción, se verifica si el predicado es 1; en tal caso, la instrucción se retira normalmente y sus resultados se escriben en el registro de destino. Si el predicado es 0, no se escribe el resultado y la instrucción no tiene efectos. La predicación se explica con mayor detalle en (Dulong, 1998).

5.8.4 Cargas especulativas

Otra característica de la IA-64 que acelera la ejecución es la presencia de instrucciones LOAD especulativas. Si un LOAD es especulativo y falla, no causa una excepción, sino que se detiene y enciende un bit asociado al registro que se iba a cargar, para marcar ese registro como no válido. Éste es el bit venenoso que introdujimos en el capítulo 4. Si posteriormente se usa el registro envenenado, la excepción ocurre en ese momento; de lo contrario, nunca ocurre.

La forma en que se usa normalmente la especulación es que el compilador eleva las LOAD a posiciones anteriores a los puntos en que se necesitan. Al iniciarse antes, pueden completarse antes de que se necesiten los resultados. En el lugar en que el compilador necesita usar el registro recién cargado, inserta una instrucción CHECK. Si el valor está ahí, CHECK actúa como NOP y la ejecución continúa de inmediato. Si el valor todavía no ha llegado, la siguiente instrucción deberá parar. Si ocurrió una excepción y el bit venenoso está encendido, la excepción pendiente ocurrirá en ese punto.

En síntesis, una máquina que implementa la arquitectura IA-64 obtiene su velocidad de varias fuentes. En lo fundamental, se trata de una máquina RISC de vanguardia con filas de procesamiento, de carga/almacenamiento, y de tres direcciones. Además, la IA-64 tiene un modelo de paralelismo explícito que exige al compilador determinar cuáles instrucciones se pueden ejecutar al mismo tiempo sin conflictos y agruparlas en haces. Así, la CPU podrá programar ciegamente un haz sin tener que romperse la cabeza pensando. Luego, la predica-

ción permite fusionar en un solo flujo los enunciados de ambas ramas de un enunciado if, con lo que se elimina la ramificación condicional y por ende la predicción correspondiente. Por último, las instrucciones LOAD especulativas permiten traer los operandos con anticipación, sin incurrir en un castigo si resulta que en realidad no se necesitaban.

5.8.5 Los pies en la tierra

Si todas estas cosas funcionan como se espera, Merced en verdad será una máquina potente. No obstante, debemos expresar ciertas reservas. En primer lugar, nunca se ha construido una máquina tan avanzada, al menos no para un mercado masivo. La historia muestra que incluso los planes mejor fraguados a veces se frustran por diversas razones.

En segundo lugar, va a ser necesario escribir compiladores para la IA-64. Escribir un buen compilador para esta arquitectura no será fácil. Además, durante los últimos 30 años el Santo Grial de las investigaciones en programación paralela ha sido la extracción automática de paralelismo de programas secuenciales. Estos trabajos no han tenido mucho éxito. Si un programa no tiene mucho paralelismo inherente, o si el compilador no puede extraerlo, todos los haces de la IA-64 serán cortos, y no se ganará mucho.

En tercer lugar, todo lo que dijimos en las secciones anteriores supone que el sistema operativo es cabalmente de 64 bits. Windows 95 y Windows 98 ciertamente no lo son y probablemente nunca lo serán. Esto implica que todo mundo va a tener que cambiar a Windows NT o a alguna versión de UNIX. Semejante transición podría ser dolorosa. Después de todo, a 10 años de la introducción del 386 de 32 bits, Windows 95 aún contenía una buena cantidad de código para 16 bits, y nunca usó el hardware de segmentación que ha estado incluido en todas las CPU de Intel desde hace más de una década (veremos la segmentación en el capítulo 6). No sabemos cuánto tardarán los sistemas operativos en convertirse en sistemas de 64 bits cabales.

En cuarto lugar, mucha gente va a juzgar a la IA-64 por lo bien que ejecute viejos juegos de 16 bits para MS-DOS. Cuando salió el Pentium Pro, fue criticado en la prensa porque no ejecutaba los programas viejos de 16 bits más rápidamente que el Pentium normal. Puesto que los programas viejos de 16 bits (o incluso de 32 bits) ni siquiera usarán las nuevas características de las CPU IA-64, todos los registros de predicado del mundo de nada servirán. Para observar mejoras, la gente tendrá que comprar modernizaciones de todo su software, compiladas con los nuevos compiladores en modo IA-64. Muchos de esos usuarios se resistirán a efectuar el desembolso.

Por último, podría haber alternativas de otros fabricantes (incluido Intel) que también ofrezcan alto desempeño empleando arquitecturas RISC más convencionales, tal vez con más instrucciones condicionales en una forma más débil de predicación. Hasta las versiones más rápidas de la línea Pentium II misma podrían ser importantes rivales de la IA-64. Es probable que pasen muchos años antes de que la IA-64 alcance el tipo de dominio del mercado que la IA-32 tiene ahora, y podría no suceder nunca.

5.9 RESUMEN

El nivel de arquitectura del conjunto de instrucciones es lo que la mayoría de las personas conoce como "lenguaje de máquina". En este nivel, la máquina tiene una memoria orientada a bytes o palabras que consiste en algunas decenas de megabytes, e instrucciones como MOVE, ADD y BEQ.

Casi todas las computadoras modernas tienen una memoria organizada en forma de una secuencia de bytes, los cuales se agrupan en palabras de 4 u 8 bytes cada una. También es normal contar con entre 8 y 32 registros, cada uno de los cuales puede contener una palabra. En algunas máquinas (como el Pentium II), las referencias a palabras de memoria no tienen que estar alineadas respecto a las fronteras naturales de la memoria, mientras que en otras (como el UltraSPARC II) es obligatorio.

Las instrucciones generalmente tienen uno, dos o tres operandos, que se direccionan empleando modos de direccionamiento directo, de registro, indexado, etc. Algunas máquinas tienen un gran número de modos de direccionamiento complejos. Por lo regular se cuenta con instrucciones para trasladar datos, efectuar operaciones diádicas o monádicas que incluyen operaciones aritméticas y booleanas, ramificaciones, llamadas a procedimientos, y ciclos, y a veces para E/S. Una instrucción representativa traslada una palabra de la memoria a un registro (o viceversa), suma, resta, multiplica o divide dos registros o un registro y una palabra de memoria, o compara dos valores que están en registros o en la memoria. No es inusitado que una computadora tenga más de 200 instrucciones en su repertorio.

El flujo de control en el nivel 2 se implementa utilizando diversas primitivas que incluyen ramificaciones, llamadas a procedimientos, llamadas a corrutinas, trampas e interrupciones. Las ramificaciones sirven para terminar una secuencia de instrucciones e iniciar una nueva. Los procedimientos se usan como mecanismo de abstracción, a fin de aislar una parte del programa en una unidad e invocarla desde varios puntos. Las corrutinas permiten a dos hilos de control operar simultáneamente. Las trampas sirven para avisar de situaciones excepcionales, como un desbordamiento aritmético. Las interrupciones permiten efectuar E/S en paralelo con los cálculos principales, ya que la CPU recibe una señal tan pronto como se completa la E/S.

Examinamos una buena solución recursiva del divertido problema de las Torres de Hanoi.

Por último, la arquitectura IA-64 usa el modelo de computación EPIC que facilita la explotación del paralelismo por parte de los programas. Además, usa predicación y cargas especulativas para aumentar su velocidad. Esto podría representar un adelanto importante respecto al Pentium II, pero relega una buena parte del trabajo de paralelización al compilador.

PROBLEMAS

1. En el Pentium II las instrucciones pueden contener cualquier número de bytes, par o impar. En el UltraSPARC II todas las instrucciones contienen un número entero de palabras, es decir, un número par de bytes. Cite una ventaja del esquema del Pentium II.

2. Diseñe un código de operación expansible que permita codificar lo siguiente en una instrucción de 36 bits:

7 instrucciones con dos direcciones de 15 bits y un número de registro de 3 bits.
500 instrucciones con una dirección de 15 bits y un número de registro de 3 bits.
50 instrucciones sin direcciones ni registros.

3. ¿Es posible diseñar un código de operación expansible que permita codificar lo siguiente en una instrucción de 12 bits? La especificación de un registro requiere 3 bits.

4 instrucciones con 3 registros
255 instrucciones con un registro
16 instrucciones con cero registros.

4. Cierta máquina tiene instrucciones de 16 bits y direcciones de 6 bits. Algunas instrucciones tienen una dirección y otras tienen dos. Si hay n instrucciones de dos direcciones, ¿cuántas instrucciones de una dirección puede haber como máximo?

5. Dados los valores de memoria que siguen y una máquina de una dirección con acumulador, ¿qué valores cargan en el acumulador las instrucciones siguientes?

la palabra 20 contiene 40
la palabra 30 contiene 50
la palabra 40 contiene 60
la palabra 50 contiene 70

a. LOAD IMMEDIATE 20
b. LOAD DIRECT 20
c. LOAD INDIRECT 20
d. LOAD IMMEDIATE 30
e. LOAD DIRECT 30
f. LOAD INDIRECT 30

6. Compare las máquinas de 0, 1, 2 y 3 direcciones escribiendo programas que calculen

$$X = (A + B \times C) / (D - E \times F)$$

para cada una de las cuatro máquinas. Las instrucciones que se pueden usar son:

0 direcciones	1 dirección	2 direcciones		3 direcciones	
PUSH M	LOAD M	MOV	(X = Y)	MOV	(X = Y)
POP M	STORE M	ADD	(X = X+Y)	ADD	(X = Y+Z)
ADD	ADD M	SUB	(X = X–Y)	SUB	(X = Y–Z)
SUB	SUB M	MUL	(X = X*Y)	MUL	(X = Y*Z)
MUL	MUL M	DIV	(X = X/Y)	DIV	(X = Y/Z)
DIV	DIV M				

M es una dirección de memoria de 16 bits, y X, Y y Z son direcciones de 16 bits o bien registros de 4 bits. La máquina de cero direcciones usa una pila, la máquina de una dirección usa un acumulador, y las otras dos tienen 16 registros e instrucciones que operan con todas las combinaciones posibles de localidades de memoria y registros. SUB X,Y resta Y a X, y SUB X,Y,Z resta Z a Y y coloca el resultado en X. Suponiendo códigos de operación e instrucciones cuyas longitudes son múltiplos de 4 bits, ¿cuántos bits necesita cada máquina para calcular X?

7. Diseñe un mecanismo de direccionamiento que permita especificar en un campo de 6 bits un conjunto arbitrario de 64 direcciones, no necesariamente contiguas, de un espacio de direcciones grande.

8. Sugiera una desventaja del código automodificante que no se haya mencionado en el texto.

9. Convierta las siguientes fórmulas de notación infija a notación polaca inversa.

a. A+B+C+D+E
b. (A+B)×(C+D)+E
c. (A×B)+(C×D)+E
d. (A+B)×(((C–D×E) / F) / G)×H

10. Convierta las siguientes fórmulas de notación polaca inversa a notación infija.

a. AB+C+D×
b. AB/CD/+
c. ABCDE+××/
d. ABCDE×F/+G–H/×+

11. ¿Cuáles de los siguientes pares de fórmulas en notación polaca inversa son matemáticamente equivalentes?

a. AB+C+ y ABC++
b. AB–C– y ABC––
c. AB×C+ y ABC+×

12. Escriba tres fórmulas en notación polaca inversa que no puedan convertirse a notación infija.

13. Convierta las siguientes fórmulas booleanas infijas a notación polaca inversa.

a. (A AND B) OR C
b. (A OR B) AND (A OR C)
c. (A AND B) OR (C AND D)

14. Convierta la siguiente fórmula infija a notación polaca inversa y genere código JVM para evaluarla.

$$(2 \times 3 + 4) - (4 / 2 + 1)$$

15. La instrucción de lenguaje ensamblador

```
MOV REG,ADDR
```

significa "cargar un registro de la memoria" en el Pentium II. En cambio, si queremos cargar un registro de la memoria en el UltraSPARC II escribimos

```
LOAD ADDR,REG
```

¿Por qué es diferente el orden de los operandos?

16. ¿Cuántos registros tiene la máquina cuyos formatos de instrucción se dan en la figura 5-24?

17. En la figura 5-24, el bit 23 sirve para distinguir el uso del formato 1 y el formato 2, pero ningún bit indica el uso del formato 3. ¿Cómo sabe el hardware cuándo debe usarlo?

18. En programación es común que un programa necesite determinar dónde está una variable X respecto al intervalo de A a B. Si se contara con una instrucción de tres direcciones con operandos A, B y X, ¿cuántos bits de código de condición tendría que ajustar esta instrucción?

19. El Pentium II tiene un bit de código de condición que sigue la pista al acarreo de salida del bit 3 después de una operación aritmética. ¿De qué sirve?

20. El UltraSPARC II no tiene ninguna instrucción para cargar un número de 32 bits en un registro. En vez de ello, normalmente se usa una secuencia de dos instrucciones, SETHI y ADD. ¿Hay más de una forma de cargar un número de 32 bits en un registro? Comente su respuesta.

21. Uno de sus amigos irrumpe en su habitación a las 3 A.M., jadeando, para contarle a usted la brillante idea que se le acaba de ocurrir: una instrucción con dos códigos de operación. ¿A dónde debe enviar a su amigo, a la oficina de patentes o al psiquiatra?

22. Las pruebas de la forma

```
if (n == 0) ...
if (i > j) ...
if (k < 4) ...
```

son comunes en programación. Genere una instrucción que realice tales pruebas de forma eficiente. ¿Qué campos contiene su instrucción?

23. Para el número binario de 16 bits 1001 0101 1100 0011, muestre el efecto de:

a. Un desplazamiento de 4 bits a la derecha rellenando con ceros.
b. Un desplazamiento de 4 bits a la derecha extendiendo el signo.
c. Un desplazamiento de 4 bits a la izquierda.
d. Una rotación de 4 bits a la izquierda.
e. Una rotación de 4 bits a la derecha.

24. ¿Cómo puede borrar una palabra de memoria en una instrucción que no cuenta con una instrucción CLR?

25. Calcule la expresión booleana (*A* AND *B*) OR *C* para

A = 1101 0000 1010 1101
B = 1111 1111 0000 1111
C = 0000 0000 0010 0000

26. Diseñe una forma de intercambiar dos variables *A* y *B* sin usar una tercera variable o registro. *Sugerencia*: piense en la operación OR EXCLUSIVO.

27. En cierta computadora es posible trasladar un número de un registro a otro, desplazar cada uno a la izquierda diferentes números de bits, y sumar los resultados en menos tiempo del que tarda una multiplicación. ¿En qué condiciones es útil esta sucesión de instrucciones para calcular "constante × variable"?

28. Las diferentes máquinas tienen diferentes densidades de instrucciones (número de bytes requeridos para realizar cierto cálculo). Traduzca cada uno de los siguientes fragmentos de código en Java a lenguaje ensamblador de Pentium II, lenguaje ensamblador de UltraSPARC II, y JVM. Luego determine cuántos bytes requiere la expresión en cada máquina. Suponga que *i* y *j* son variables locales en la memoria, pero por lo demás haga los supuestos más optimistas en todos los casos.

a. i = 3;
b. i = j;
c. i = j – 1;

29. Las instrucciones de ciclos que vimos en el texto servían para manejar ciclos for. Diseñe una instrucción que sirva para manejar ciclos while comunes.

30. Suponga que los monjes de Hanoi pueden mover un disco por minuto (no tienen prisa por terminar el trabajo porque las oportunidades de empleo para personas con esta habilidad en particular son limitadas en Hanoi). ¿Cuánto tardarán en resolver el problema con los 64 discos? Exprese su resultado en años.

31. ¿Por qué los dispositivos de E/S colocan el vector de interrupción en el bus? Sería posible guardar esa información en una tabla en la memoria en vez de colocarla en el bus?

32. Una computadora usa DMA para leer de su disco. El disco tiene 64 sectores de 512 bytes en cada pista. El tiempo de rotación del disco es de 16 ms. El bus tiene 16 bits de anchura, y las transferencias de bus tardan 500 ns cada una. Una instrucción de CPU representativa requiere dos ciclos de bus. ¿Cuánto frena el DMA a la CPU?

33. ¿Por qué se asignan prioridades a las rutinas de servicio de interrupción pero no a los procedimientos normales?

34. La arquitectura IA-64 contiene un número inusitadamente alto de registros (64). ¿La decisión de tener tantos registros tiene que ver con el uso de la predicación? Si es así, ¿en qué sentido? Si no, ¿por qué hay tantos?

35. En el texto se explica el concepto de instrucciones LOAD especulativas. Sin embargo, no se mencionan instrucciones STORE especulativas. ¿Por qué no? ¿Son en lo esencial iguales a las instrucciones LOAD especulativas o hay alguna otra razón para no mencionarlas?

36. Cuando se desea conectar dos redes de área local, se inserta entre ellas una computadora llamada puente, conectada a ambas redes. Cada paquete que se transmite por cualquiera de las redes causa una interrupción en el puente, para que éste vea si tiene que reenviar el paquete o no. Suponga que se requieren 250 μs por paquete para manejar la interrupción e inspeccionar el paquete, pero su reenvío, si es necesario, corre por cuenta de hardware de DMA y es "gratuito" para la CPU. Si todos los paquetes son de 1K bytes, ¿qué tasa de datos máxima puede tolerarse en cada red sin que el puente pierda paquetes?

37. En la figura 5-41 el apuntador de marco apunta a la primera variable local. ¿Qué información necesita el programa para regresar de un procedimiento?

38. Escriba una subrutina en lenguaje ensamblador que convierta un entero binario con signo a ASCII.

39. Escriba una subrutina en lenguaje ensamblador que convierta una fórmula infija a notación polaca inversa.

40. El de las Torres de Hanoi no es el único procedimiento recursivo favorito de los computólogos. Otro gran favorito es $n!$, donde $n! = n(n-1)!$ sujeto a la condición limitante de que $0! = 1$. Escriba un procedimiento en su lenguaje ensamblador favorito para calcular $n!$

41. Si no está convencido de que la recursión a veces es indispensable, trate de programar las Torres de Hanoi sin usar recursión y sin simular la solución recursiva manteniendo una pila en un arreglo. Conviene advertir que probablemente no logre encontrar la solución.

6

EL NIVEL DE MÁQUINA DE SISTEMA OPERATIVO

El tema de este libro es que una computadora moderna consiste en una serie de niveles, cada uno de los cuales añade funcionalidad al nivel que está abajo. Ya vimos el nivel de lógica digital, el nivel de microarquitectura y el nivel de conjunto de instrucciones. Ha llegado el momento de ascender al siguiente nivel, el ámbito del sistema operativo.

Un **sistema operativo** es un programa que, desde el punto de vista del programador, añade varias instrucciones y funciones nuevas, más allá de lo que el nivel ISA proporciona. Normalmente, el sistema operativo se implementa casi totalmente en software, pero no existe una razón teórica para no colocarlo en hardware, como se hace normalmente con los microprogramas (cuando están presentes). Usaremos el acrónimo **OSM** (*Operating System Machine*) para referirnos al nivel que el sistema operativo implementa, el nivel de **máquina de sistema operativo**, que se muestra en la figura 6-1.

Aunque tanto el nivel OSM como el nivel ISA son abstractos (en el sentido de que no son el verdadero nivel de hardware), existe una diferencia importante entre ellos. El conjunto de instrucciones del nivel OSM es el conjunto completo de instrucciones que pueden usar los programadores de aplicaciones; contiene casi todas las instrucciones del nivel ISA, así como el conjunto de instrucciones nuevas que el sistema operativo añade. Estas nuevas instrucciones se denominan **llamadas al sistema**. Una llamada al sistema invoca un servicio predefinido del sistema operativo, o sea, una de sus instrucciones. Una llamada al sistema típica solicita leer datos de un archivo. Usaremos el tipo de letra helvética en minúsculas para distinguir las llamadas al sistema.

El nivel OSM siempre se interpreta. Cuando un programa de usuario ejecuta una instrucción OSM, como leer datos de un archivo, el sistema operativo ejecuta esta instrucción paso

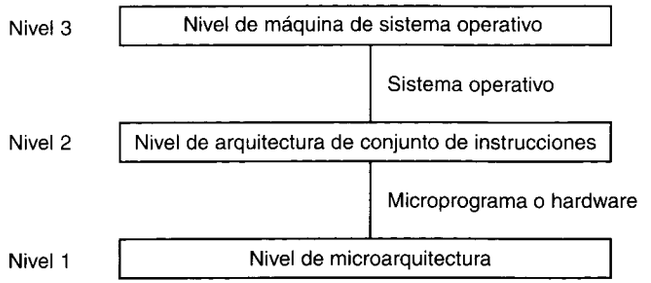


Figura 6-1. Ubicación del nivel de máquina de sistema operativo.

por paso, igual que un microprograma ejecutaría una instrucción ADD paso por paso. Sin embargo, cuando un programa ejecuta una instrucción del nivel ISA, el nivel de microarquitectura subyacente se encarga de llevarla a cabo directamente, sin ayuda del sistema operativo.

En este libro sólo podemos proporcionar una muy breve introducción al tema de los sistemas operativos. Nos concentraremos en tres importantes temas. El primero es la memoria virtual, una técnica que muchos sistemas operativos usan para aparentar que la máquina tiene más memoria de la que en realidad tiene. El segundo es la E/S de archivos, un concepto de nivel más alto que las instrucciones de E/S que estudiamos en el capítulo anterior. El tercer y último tema es el de procesamiento en paralelo: cómo varios procesos pueden ejecutarse, comunicarse y sincronizarse. El concepto de proceso es muy importante, y lo describiremos con detalle más adelante. Por ahora, podemos pensar en un proceso como un programa en ejecución y toda su información de estado (memoria, registros, contador de programa, situación de E/S, etc.). Después de explicar estos principios en general, veremos cómo se aplican a los sistemas operativos de dos de nuestras máquinas de ejemplo, el Pentium II (Windows NT) y el UltraSPARC II (UNIX). Puesto que el picoJava II normalmente se usa para sistemas incorporados, no tiene un sistema operativo con todas las de la ley.

6.1 MEMORIA VIRTUAL

En los albores de la computación, las memorias eran pequeñas y costosas. El IBM 650, la principal computadora científica de su época (fines de la década de los cincuenta), sólo tenía 2000 palabras de memoria. Uno de los primeros compiladores de ALGOL 60 se escribió para una máquina que sólo tenía 1024 palabras de memoria. Uno de los primeros sistemas de tiempo compartido trabajaba de forma muy satisfactoria en una PDP-1 con un tamaño total de memoria de 4096 palabras de 18 bits para el sistema operativo y los programas de usuario combinados. En esos días el programador dedicaba gran parte de su tiempo a lograr que sus programas cupieran en la diminuta memoria. En muchos casos era necesario usar un algoritmo que trabajaba mucho más despacio que otro algoritmo mejor, simplemente porque el algoritmo mejor era demasiado grande; es decir, un programa que usaba el algoritmo mejor no cabía en la memoria de la computadora.

La solución tradicional a este problema era usar memoria secundaria, como un disco. El programador dividía el programa en varios fragmentos, llamados **superposiciones**, cada uno de los cuales cabía en la memoria. Para ejecutar el programa, se leía la primera superposición y se ejecutaba durante un rato. Cuando terminaba, leía la siguiente superposición y la invocaba, y así. El programador tenía la responsabilidad de dividir el programa en superposiciones, decidir en qué lugar de la memoria secundaria se debía guardar cada superposición, encargarse del traslado de superposiciones entre la memoria principal y la memoria secundaria, y en general administrar el proceso de superponer sin ayuda de la computadora.

Aunque se usó ampliamente durante muchos años, esta técnica implicaba mucho trabajo invertido en la gestión de superposiciones. En 1961 un grupo de investigadores de Manchester, Inglaterra, propuso un método para realizar automáticamente el proceso de superponer, sin que el programador siquiera supiera qué estaba ocurriendo (Fotheringham, 1961). Este método, ahora conocido como **memoria virtual**, tenía la ventaja obvia de liberar al programador de una gran cantidad de molestas tareas de contabilidad, y se usó por primera vez en los años sesenta en varias computadoras, casi todas asociadas a proyectos de investigación sobre diseño de sistemas de cómputo. Para cuando comenzó la década de los setenta casi todas las computadoras contaban con memoria virtual. Ahora hasta las computadoras de un solo chip, incluidos el Pentium II y el UltraSPARC II, tienen sistemas de memoria virtual muy sofisticados, que examinaremos en una sección posterior del capítulo.

6.1.1 Paginación

La idea sugerida por el grupo de Manchester fue la de separar los conceptos de espacio de direcciones y posiciones de memoria. Consideremos, por ejemplo, una computadora representativa de esa época, con un campo de dirección de 16 bits en sus instrucciones y 4096 palabras de memoria. Un programa para esta computadora podía direccionar 65536 palabras de memoria. La razón es que existen 65536 (2^{16}) direcciones de 16 bits, cada una de las cuales corresponde a una palabra de memoria distinta. Cabe señalar que el número de palabras direccionables depende únicamente del número de bits que tiene una dirección y nada tiene que ver con el número de palabras de memoria con que se cuenta realmente. El **espacio de direcciones** de esta computadora consiste en los números 0, 1, 2, . . . , 65535, porque ése es el conjunto de posibles direcciones. Sin embargo, la computadora bien podría tener menos de 65535 palabras de memoria.

Antes de que se inventara la memoria virtual, la gente habría distinguido entre las direcciones por debajo de 4096 y aquellas iguales o mayores que 4096. Aunque pocas veces se decía explícitamente, estas dos partes se consideraban como el espacio de direcciones útiles y el espacio de direcciones inútiles, respectivamente (las direcciones mayores que 4095 eran inútiles porque no correspondían a posiciones de memoria reales). La gente no distinguía entre espacio de direcciones y direcciones de memoria, porque el hardware exigía una correspondencia uno a uno entre ellos.

La idea de separar el espacio de direcciones y las direcciones de memoria es la siguiente. En cualquier momento dado, es posible acceder directamente a 4096 palabras de memoria,

pero no es necesario que correspondan a las direcciones de memoria 0 a 4095. Podríamos, por ejemplo, "decirle" a la computadora que, en adelante, cada vez que se haga referencia a la dirección 4096, se usará la palabra de memoria que está en la dirección 0. Cada vez que se haga referencia a la dirección 4097, se usará la palabra de memoria que está en la dirección 1; cada vez que se haga referencia a la dirección 8191, se usará la palabra de memoria que está en la dirección 4095, y así. En otras palabras, habremos definido una correspondencia o "mapeo" del espacio de direcciones a las direcciones de memoria reales, como se muestra en la figura 6-2.

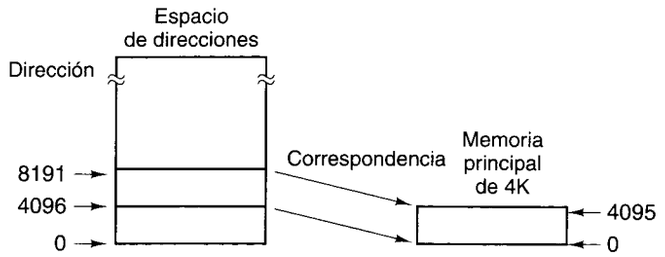


Figura 6-2. Mapeo en el que las direcciones virtuales 4096 a 8191 se hacen corresponder con las direcciones 0 a 4095 de la memoria principal.

En términos de hacer corresponder las direcciones del espacio de direcciones con las posiciones de memoria reales, una máquina de 4K sin memoria virtual simplemente tiene una correspondencia fija entre las direcciones 0 a 4095 y las 4096 palabras de memoria. Una pregunta interesante es: "¿Qué sucede si un programa ramifica a una dirección entre 8192 y 12287?" En una máquina sin memoria virtual, el programa causaría una trampa de error que imprimiría un mensaje debidamente grosero como "Referencia a memoria inexistente" y terminaría el programa. En una máquina con memoria virtual, ocurriría la siguiente sucesión de pasos:

1. El contenido de la memoria principal se guardaría en el disco.
2. Se localizarían en el disco las palabras 8192 a 12287.
3. Se cargarían en la memoria principal las palabras 8192 a 12287.
4. El mapa de direcciones se modificaría para hacer corresponder las direcciones 8192 a 12287 con las posiciones de memoria 0 a 4095.
5. La ejecución continuaría como si nada fuera de lo normal hubiera ocurrido.

Esta técnica de superposición automática se denomina **paginación**, y los fragmentos de programa que se leen del disco se llaman **páginas**.

Puede usarse una forma más sofisticada de hacer corresponder las direcciones del espacio de direcciones con las direcciones de memoria real. Para destacar la diferencia, llamaremos a las direcciones a las que el programa puede hacer referencia **espacio de direcciones virtual**, y a las direcciones de memoria reales, en hardware, **espacio de direcciones físico**. Un **mapa de memoria** o **tabla de páginas** relaciona las direcciones virtuales con las direcciones físicas. Suponemos que hay suficiente espacio en el disco para guardar todo el espacio de direcciones virtual (o al menos la porción de ese espacio que se está usando).

Los programas se escriben como si hubiera suficiente memoria principal para todo el espacio de direcciones virtual, aunque no sea así. Los programas pueden cargar valores de, o guardarlos en, cualquier palabra del espacio de direcciones virtual, o saltar a cualquier instrucción situada en cualquier punto del espacio de direcciones virtual, sin tener en cuenta el hecho de que en realidad no hay suficiente memoria física. De hecho, el programador puede escribir programas sin siquiera saber si existe o no memoria virtual. La computadora simplemente parece tener una memoria grande.

Este punto es crucial y se contrastará posteriormente con la segmentación, en la que el programador debe tener conocimiento de la existencia de segmentos. Hacemos hincapié una vez más en que la paginación ofrece al programador la ilusión de una memoria principal grande, continua y lineal, del mismo tamaño que el espacio de direcciones virtual. En realidad, la memoria principal disponible podría ser menor (o mayor) que el espacio de direcciones virtual. La simulación de esta memoria grande por paginación no puede ser detectada por el programa (a menos que se efectúen pruebas de cronometría). Cada vez que se hace referencia a una dirección, parece estar presente la palabra de instrucción o datos correcta. Puesto que el programador puede programar como si no existiera la paginación, se dice que el mecanismo de paginación es **transparente**.

No es la primera vez que nos topamos con la idea de que un programador pueda usar alguna característica inexistente sin preocuparse por su funcionamiento. El conjunto de instrucciones del nivel ISA a menudo incluye una instrucción MUL, a pesar de que la microarquitectura subyacente no cuenta con un dispositivo de multiplicación en el hardware. La ilusión de que la máquina puede multiplicar casi siempre se mantiene con microcódigo. Así mismo, la máquina virtual que el sistema operativo proporciona puede dar la ilusión de que todas las direcciones virtuales están respaldadas por memoria real, aunque no sea así. Sólo los escritores de sistemas operativos (y estudiantes de tales sistemas) tienen que saber cómo se mantiene la ilusión.

6.1.2 Implementación de la paginación

Un requisito indispensable para una memoria virtual es un disco en el cual pueda guardarse todo el programa y todos los datos. En lo conceptual, es más sencillo pensar que la copia del programa que está en el disco es el original y que los fragmentos que se traen a la memoria principal de vez en cuando son copias, y no lo contrario. Desde luego, es importante mantener actualizado el original. Cuando se modifica la copia que está en la memoria principal, los cambios deben reflejarse en el original (tarde o temprano).

El espacio de direcciones se divide en varias páginas de tamaño uniforme. Hoy día son comunes tamaños de página entre 512 y 64K bytes por página, aunque de vez en cuando se usan tamaños de hasta 4 MB. El tamaño de página siempre es una potencia de 2. El espacio de direcciones físico se divide en fragmentos de la misma manera, y cada uno es del mismo tamaño de una página, de modo que cada fragmento de la memoria principal pueda contener exactamente una página. Estos fragmentos de la memoria principal en los que entran las

páginas se llaman **marcos de página**. En la figura 6-2 la memoria principal contiene sólo un marco de página. En los diseños prácticos por lo regular contiene miles de marcos.

La figura 6-3(a) ilustra una posible forma de dividir los primeros 64K de un espacio de direcciones virtual: en páginas de 4K. (Tome nota de que estamos hablando de 64K y 4K de direcciones aquí. Una dirección podría ser un byte pero también podría ser una palabra en una computadora en la que palabras consecutivas tienen direcciones consecutivas.) La memoria virtual de la figura 6-3 se implementaría con una tabla de páginas que tiene tantas entradas como páginas caben en el espacio de direcciones virtual. Para simplificar, sólo hemos mostrado aquí las primeras 16 entradas. Cuando el programa trata de hacer referencia a una palabra que está en los primeros 64K de su espacio de direcciones virtual, sea para traer instrucciones, traer datos o guardar datos, primero genera una dirección virtual entre 0 y 65532 (suponiendo que las direcciones de palabra deben ser divisibles entre 4). Se puede usar indexación, direccionamiento indirecto o cualquier otra de las técnicas acostumbradas para generar la dirección.

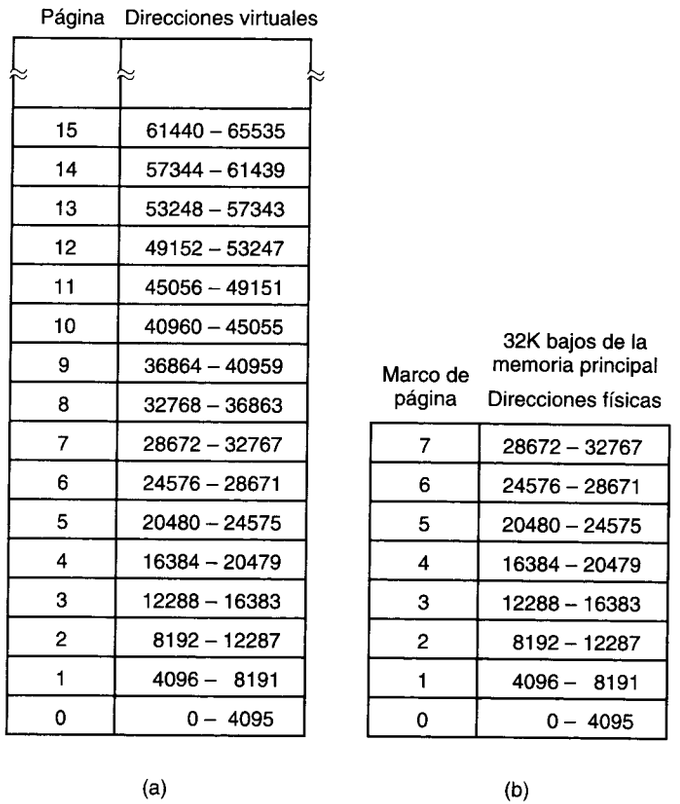


Figura 6-3. (a) Los primeros 64K del espacio de direcciones virtual divididos en 16 páginas, cada una de las cuales tiene 4K. (b) Memoria principal de 32K dividida en ocho marcos de página de 4K cada uno.

La figura 6-3(b) muestra una memoria física que consiste en ocho marcos de página de 4K. Esta memoria podría estar limitada a 32K porque (1) eso es todo lo que la máquina tiene (un procesador incorporado en una lavadora de ropa o un horno de microondas tal vez no necesite más), o (2) el resto de la memoria se asignó a otros programas.

Considere ahora cómo puede hacerse corresponder una dirección virtual de 32 bits con una dirección física en la memoria principal. Después de todo, lo único que la memoria entiende son direcciones de la memoria principal, no direcciones virtuales, así que eso es lo que debemos darle. Toda computadora con memoria virtual tiene un dispositivo para efectuar el mapeo de virtual a físico, llamado **unidad de gestión de memoria** (**MMU**, *Memory Management Unit*). La MMU podría estar en el chip de la CPU o en un chip aparte que funcione en estrecha colaboración con la CPU. Puesto que nuestra MMU de ejemplo establece una correspondencia entre una dirección virtual de 32 bits y una dirección física de 15 bits, necesita un registro de entrada de 32 bits y un registro de salida de 15 bits.

Para ver cómo funciona la MMU, considere el ejemplo de la figura 6-4. Cuando la MMU recibe una dirección virtual de 32 bits, la divide en un número de página virtual, de 20 bits, y una distancia dentro de la página, de 12 bits (porque en nuestro ejemplo las páginas son de 4K). El número de página virtual se usa como índice de la tabla de páginas para encontrar la entrada que corresponde a la página a la que se hizo referencia. En la figura 6-4 el número de página virtual es 3, por lo que se selecciona la entrada 3 de la tabla de páginas, como se muestra.

Lo primero que la MMU hace con la entrada de la tabla de páginas es verificar si la página a la que se hizo referencia está actualmente en la memoria principal. Después de todo, con 2^{20} páginas virtuales y sólo 8 marcos de página, no todas las páginas virtuales pueden estar en la memoria a la vez. La MMU efectúa esta verificación examinando el **bit presente/ausente** de la entrada de la tabla de páginas. En nuestro ejemplo el bit es 1, lo que implica que la página actualmente está en la memoria.

El siguiente paso es tomar el valor de marco de página de la entrada seleccionada (6 en este caso) y copiarlo en los tres bits superiores del registro de salida de 15 bits. Se requieren tres bits porque hay ocho marcos de página en la memoria física. Al mismo tiempo que se realiza esta operación, los 12 bits de orden bajo de la dirección virtual (el campo de distancia dentro de la página) se copian en los 12 bits de orden bajo del registro de salida, como se muestra. Esta dirección de 15 bits se envía entonces a caché o a la memoria para buscarla.

La figura 6-5 muestra una posible correspondencia entre páginas virtuales y marcos de página físicos. La página virtual 0 está en el marco de página 1. La página virtual 1 está en el marco de página 0. La página virtual 2 no está en la memoria principal. La página virtual 3 está en el marco de página 2. La página virtual 4 no está en la memoria principal. La página virtual 5 está en el marco de página 6, etcétera.

6.1.3 Paginación por demanda y modelo de conjunto de trabajo

En la explicación anterior se supuso que la página virtual a la que se hizo referencia estaba en la memoria principal. Sin embargo, tal supuesto no siempre se cumplirá porque no hay suficiente espacio en la memoria principal para todas las páginas virtuales. Cuando se hace una referencia a una dirección que está en una página que no está presente en la memoria principal, ocurre un **fallo de página**. En tal caso, el sistema operativo necesita leer la página requerida del disco, introducir en la tabla de páginas su nueva ubicación en la memoria física, y luego repetir la instrucción que causó el fallo.

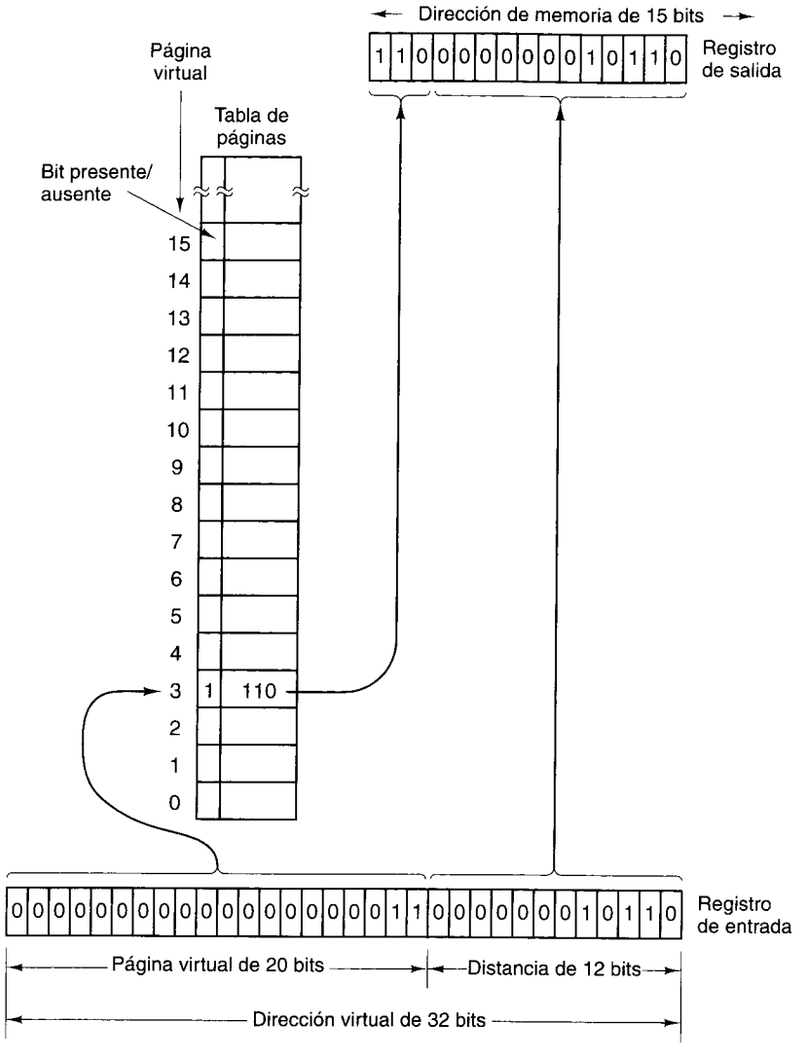


Figura 6-4. Formación de una dirección de memoria principal a partir de una dirección virtual.

Es posible iniciar la ejecución de un programa en una máquina con memoria virtual aunque ninguna parte del programa esté en la memoria principal. Simplemente hay que ajustar la tabla de páginas de modo que indique que todas y cada una de las páginas virtuales está en la memoria secundaria y no en la memoria principal. Cuando la CPU trate de traer la primera instrucción, de inmediato causará un fallo de página, lo que hará que la página que contiene la primera instrucción se cargue en la memoria y se introduzca en la tabla de páginas. Luego podrá iniciarse la primera instrucción. Si dicha instrucción tiene dos direcciones,

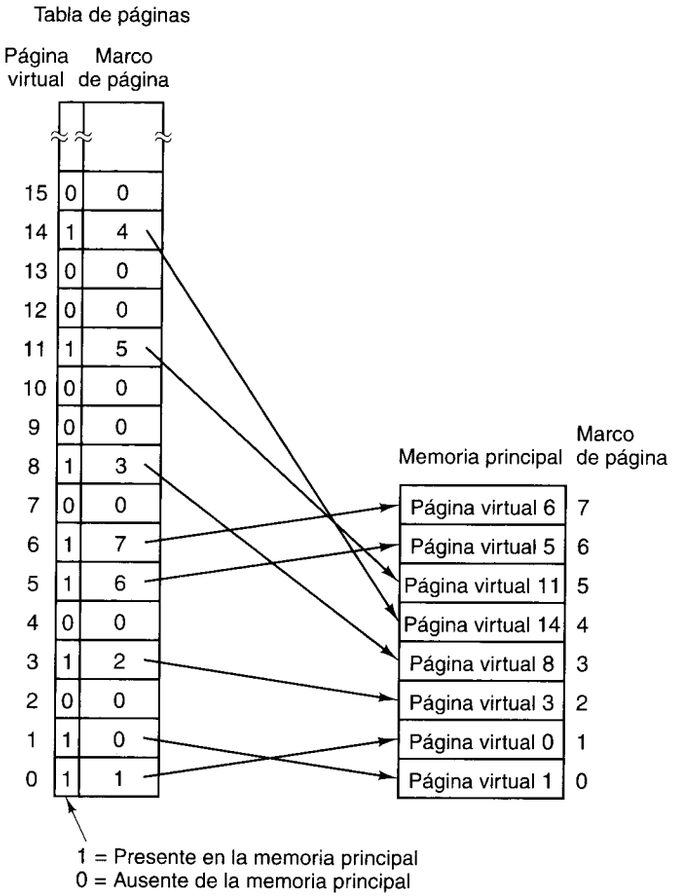


Figura 6-5. Una posible correspondencia entre las primeras 16 páginas virtuales y una memoria principal con ocho marcos de página.

las dos están en diferentes páginas y ambas páginas son diferentes de la página en la que estaba la instrucción, ocurrirán otros dos fallos de página, y se traerán dos páginas más antes de que la instrucción por fin pueda ejecutarse. La siguiente instrucción podría causar más fallos de página, y así sucesivamente.

Este método de operar una memoria virtual se llama **paginación por demanda**, por analogía con el conocido algoritmo de alimentación por demanda para los bebés: si el bebé llora, hay que alimentarlo (en lugar de alimentarlo según un programa estricto). En la paginación por demanda, sólo se traen páginas cuando se solicita realmente una página, no por adelantado.

La pregunta de si debe usarse o no paginación por demanda sólo es pertinente cuando se inicia un programa. Una vez que el programa ha estado trabajando durante cierto tiempo, las páginas requeridas ya se habrán reunido en la memoria principal. Si la computadora es de

tiempo compartido y los procesos se intercambian a disco después de ejecutarse durante 100 ms (digamos), cada programa se reiniciará muchas veces durante su ejecución. Puesto que el mapa de memoria es único para cada programa, y cambia cuando se conmutan los programas, como en un sistema de tiempo compartido, la pregunta se vuelve una cuestión crítica.

La estrategia alternativa se basa en la observación de que la mayor parte de los programas no hace referencia a su espacio de direcciones de manera uniforme, sino que las referencias tienden a concentrarse en un número reducido de páginas. Una referencia a la memoria podría traer una instrucción, podría traer datos o podría guardar datos. En cualquier instante t existe un conjunto formado por todas las páginas utilizadas por las k referencias a memoria más recientes. Denning (1968) lo llamó **conjunto de trabajo**.

Dado que el conjunto de trabajo suele variar lentamente con el tiempo, es posible hacer una conjetura razonable respecto a cuáles páginas se necesitarán cuando se reinicie el programa, con base en el conjunto de trabajo que tenía cuando se suspendió la última vez. Estas páginas podrían cargarse por adelantado antes de iniciar el programa (suponiendo que caben).

6.1.4 Política de reemplazo de páginas

Idealmente, el conjunto de páginas que un programa está usando activa e intensivamente, llamado **conjunto de trabajo**, se puede mantener en la memoria a fin de reducir los fallos de página. Sin embargo, los programadores casi nunca saben cuáles páginas están en el conjunto de trabajo, por lo que el sistema operativo debe descubrir dicho conjunto dinámicamente. Cuando un programa hace referencia a una página que no está en la memoria principal, la página requerida debe traerse del disco. Sin embargo, para que quepa generalmente es necesario enviar alguna otra página de vuelta al disco. Se necesita un algoritmo que decida cuál página debe quitarse.

Probablemente no es una buena idea escoger al azar la página que se quitará. Si la página que contiene la instrucción que causó el fallo es la que se escoge, ocurrirá otro fallo de página tan pronto como se intente traer la siguiente instrucción. Casi todos los sistemas operativos tratan de predecir cuál de las páginas de la memoria es la menos útil en el sentido de que su ausencia tendría el efecto adverso menos sensible sobre el programa en ejecución. Una forma de hacerlo es predecir cuándo ocurrirá la siguiente referencia a cada página y quitar la página cuya siguiente referencia predicha es más lejana en el futuro. En otras palabras, en lugar de expulsar una página que se necesitará en poco tiempo, se trata de seleccionar una que no se necesitará durante un buen tiempo.

Un algoritmo muy utilizado expulsa la página que se usó menos recientemente porque la probabilidad *a priori* de que no esté en el conjunto de trabajo vigente es alta. Éste es el algoritmo **LRU** (**menos recientemente utilizada**, *Least Recently Used*). Aunque este algoritmo normalmente funciona bien, existen situaciones patológicas, como la que se describirá a continuación, en las que falla lamentablemente.

Imagine un programa que ejecuta un ciclo grande que se extiende sobre nueve páginas virtuales en una máquina que tiene espacio sólo para ocho páginas en la memoria física. Una vez que el programa llega a la página 7, el estado de la memoria principal es el que se muestra

en la figura 6-6(a). En algún momento se intenta traer una instrucción de la página virtual 8, lo que causa un fallo de página. Se debe tomar una decisión acerca de cuál página se expulsará. El algoritmo LRU escoge la página virtual 0 porque es la que se usó menos recientemente. Se quita la página virtual 0 y se trae la página virtual 8 para sustituirla, y la situación es la que se muestra en la figura 6-6(b).

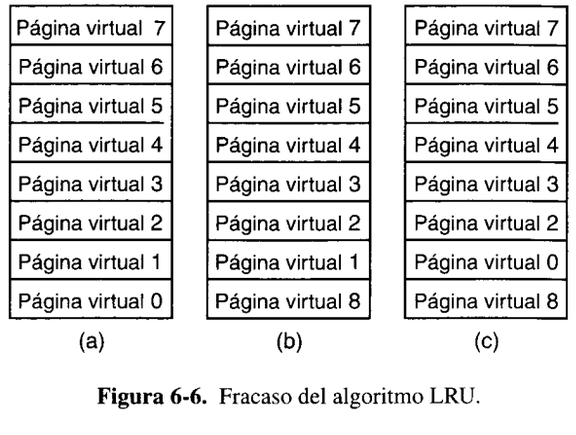


Figura 6-6. Fracaso del algoritmo LRU.

Después de ejecutar las instrucciones de la página virtual 8, el programa salta al principio del ciclo, a la página virtual 0. Este paso causa otro fallo de página. La página virtual 0, que acaba de expulsarse, se tiene que volver a traer. El algoritmo LRU escoge la página 1 para expulsión, lo cual lleva a la situación de la figura 6-6(c). El programa continuará en la página 0 durante un rato, y luego tratará de traer una instrucción de la página virtual 1, causando un fallo de página. Hay que volver a traer la página 1 y para ello se expulsará la página 2.

Ya deberá ser obvio que el algoritmo LRU está tomando la peor decisión en cada ocasión (otros algoritmos también fallan en condiciones similares). Sin embargo, si la memoria principal disponible es mayor que el tamaño del conjunto de trabajo, el algoritmo LRU sí tiende a minimizar el número de fallos de página.

Otro algoritmo de reemplazo de páginas es el de **primero en entrar, primero en salir** (**FIFO**, *First-In First-Out*). FIFO expulsa la página que se cargó menos recientemente, sin importar cuándo fue la última vez que se hizo referencia a ella. Cada marco de página tiene un contador. Inicialmente, todos los contadores contienen 0. Después de manejarse cada fallo de página, se incrementa en 1 el contador de cada página que actualmente está en la memoria, y el contador de la página que se acaba de traer se pone en 0. Cuando se hace necesario escoger una página para expulsarla, se escoge aquella cuyo contador es el más alto. Esto indica que la página correspondiente ha sido testigo del mayor número de fallos de página, y por tanto que se cargó antes que cualquiera de las otras páginas que están en la memoria y tiene (con suerte) la mayor probabilidad *a priori* de no necesitarse más.

Si el conjunto de trabajo es mayor que el número de marcos de página disponibles, ningún algoritmo que no sea un oráculo dará buenos resultados, y los fallos de página serán

frecuentes. Un programa que genera fallos de página de forma continua y frecuente está **hiperpaginando**. Sobra decir que la hiperpaginación es una condición indeseable en cualquier sistema. Si un programa usa una gran cantidad de espacio virtual de direcciones pero tiene un conjunto de trabajo pequeño que cambia lentamente y cabe en la memoria principal disponible, no causará problemas. Esta observación se cumple aunque, a lo largo de su existencia, el programa use cientos de veces más palabras de memoria virtual que las palabras de memoria principal que tiene la máquina.

Si una página que está a punto de ser expulsada no se ha modificado desde que se leyó (lo cual es muy probable si la página contiene programa, no datos), no es necesario volver a escribirla en el disco, pues ya existe ahí una copia exacta. Si la página se modificó después de leerse, la copia del disco ya no es exacta y habrá que reescribir la página.

Si hay una forma de saber si una página no ha cambiado desde que se leyó (página limpia) o si se escribió después en ella (página sucia), podrá evitarse la reescritura de páginas limpias y se ahorrará mucho tiempo. Muchas computadoras tienen un bit por página, en la MMU, que se pone en 0 cuando la página se carga, y que el microprograma o el hardware pone en 1 cada vez que se escribe en ella (es decir, se ensucia). Si el sistema operativo examina este bit, sabrá si la página está limpia o sucia, y por ende si es necesario reescribirla o no.

6.1.5 Tamaño de página y fragmentación

Si por casualidad el programa y los datos del usuario llenan exactamente un número entero de páginas, no se desperdiciará espacio cuando estén en la memoria. Pero si no sucede así, quedará cierto espacio sin utilizar en la última página. Por ejemplo, si el programa y los datos necesitan 26,000 bytes en una máquina que tiene 4096 bytes por página, las primeras seis páginas estarán llenas, para un total de $6 \times 4096 = 24{,}576$ bytes, y la última página contendrá $26{,}000 - 24{,}576 = 1424$ bytes. Puesto que en cada página caben 4096 bytes, se desperdiciarán 2672 bytes. Cada vez que la séptima página esté presente en la memoria, esos bytes ocuparán espacio en la memoria principal pero no servirán de nada. El problema de estos bytes desperdiciados se denomina **fragmentación interna** (porque el espacio desperdiciado está adentro de una página).

Si el tamaño de página es de n bytes, la cantidad media de espacio desperdiciado por fragmentación interna en la última página de un programa será de $n/2$ bytes, situación que sugiere usar páginas pequeñas para minimizar el desperdicio. Por otra parte, un tamaño de página pequeño implica muchas páginas, además de una tabla de páginas grande. Si la tabla de páginas se mantiene en hardware, y es grande, se necesitarán más registros para guardarla, y el costo de la computadora aumentará. Además, se requerirá más tiempo para cargar y guardar estos registros cada vez que se inicie o detenga un programa.

Además, las páginas pequeñas utilizan de forma ineficiente el ancho de banda del disco. Si suponemos que hay que esperar 10 ms para poder iniciar una transferencia (retraso por búsqueda + latencia rotacional), las transferencias grandes serán más eficientes que las pequeñas. Con una tasa de transferencia de 10 MB/s, transferir 8 KB tarda sólo 0.7 ms más que transferir 1 KB (10.1 ms *versus* 10.8 ms).

Sin embargo, las páginas pequeñas tienen la ventaja adicional de que si el conjunto de trabajo consiste en un gran número de regiones pequeñas discretas en el espacio de direcciones virtual podría haber menos hiperpaginación con un tamaño de página pequeño que con uno grande. Por ejemplo, considere una matriz de $10{,}000 \times 10{,}000$, A que se almacena con $A[1, 1]$, $A[2, 1]$, $A[3, 1]$, etc., en palabras de 8 bytes consecutivas. Este almacenamiento ordenado por columna implica que los elementos de la fila 1, $A[1, 1]$, $A[1, 2]$, $A[1, 3]$, etc., comenzarán en posiciones separadas 80,000 bytes una de la siguiente. Un programa que realice cálculos extensos con todos los elementos de esta fila usaría 10,000 regiones, cada una separada de la siguiente por 79,992 bytes. Si el tamaño de página fuera de 8 KB, se requeriría una memoria total de 80 MB para contener todas las páginas en uso.

Por otra parte, si el tamaño de página es de 1 KB, se requerirían sólo 10 MB de RAM para contener todas las páginas. Si la memoria disponible fuera de 32 MB, el programa hiperpaginaría si las páginas fueran de 8 KB, pero no si fueran de 1 KB.

6.1.6 Segmentación

La memoria virtual de la que hemos hablado aquí es unidimensional porque las direcciones virtuales van de la 0 a alguna dirección máxima, una tras otra. Para muchos problemas, tener dos o más espacios de direcciones distintos podría ser mucho mejor que tener sólo uno. Por ejemplo, un compilador podría tener muchas tablas que se van creando a medida que avanza la compilación y que incluyen:

1. La tabla de símbolos, que contiene los nombres y atributos de las variables.
2. El texto fuente que se está guardando para el listado impreso.
3. Una tabla que contiene todas las constantes enteras y de punto flotante empleadas.
4. El árbol de análisis sintáctico del programa.
5. La pila empleada para llamadas a procedimientos dentro del compilador.

Las primeras cuatro tablas crecen continuamente a medida que la compilación avanza. La última crece y se encoge de forma impredecible durante la compilación. En una memoria unidimensional, habría que asignar a estas tablas trozos contiguos del espacio de direcciones virtual, como en la figura 6-7.

Considere lo que sucede si un programa tiene un número excepcionalmente grande de variables. El trozo de espacio de direcciones asignado a la tabla de símbolos podría llenarse, aunque haya mucho espacio libre en las otras tablas. Desde luego, el compilador podría limitarse a emitir un mensaje diciendo que la compilación no puede continuar porque hay demasiadas variables, pero hacerlo no sería justo si queda espacio sin ocupar en las demás tablas.

Otra posibilidad es que el compilador actúe como Robin Hood y tome espacio libre de las tablas que tienen mucho para dárselo a las tablas que tienen poco. Esto es factible, pero sería análogo a gestionar sus propias superposiciones: una lata en el mejor de los casos y un exceso de trabajo tedioso y fútil en el peor de los casos.

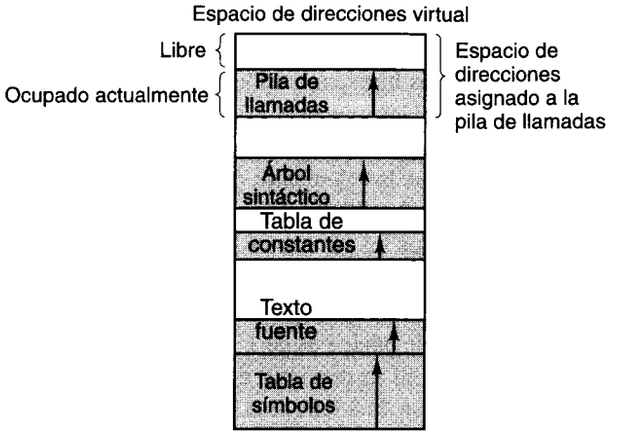


Figura 6-7. En un espacio de direcciones unidimensional con tablas que crecen, una tabla podría chocar con otra.

Lo que realmente se necesita es una forma de evitar al programador el trabajo de administrar las tablas en expansión y contracción, del mismo modo que la memoria virtual elimina la preocupación por tener que organizar el programa en superposiciones.

Una solución sencilla es proporcionar muchos espacios de direcciones totalmente independientes, llamados **segmentos**. Cada segmento consiste en una sucesión lineal de direcciones, de la 0 a alguna dirección máxima. La longitud de cada segmento puede ser cualquiera desde 0 hasta el máximo permitido. Diferentes segmentos pueden tener diferentes longitudes, y generalmente así es. Además, la longitud de los segmentos podría cambiar durante la ejecución. La longitud de un segmento de pila podría incrementarse cada vez que algo se mete en la pila y reducirse cada vez que algo se desempila.

Puesto que cada segmento constituye un espacio de direcciones individual, los diferentes segmentos pueden crecer o encogerse de manera independiente, sin afectar a los otros. Si una pila en cierto segmento necesita más espacio de direcciones para crecer, puede tenerlo, porque no hay nada más en su espacio de direcciones con lo que pueda chocar. Desde luego, un segmento podría llenarse, pero los segmentos suelen ser muy grandes, y esto casi nunca sucede. Para especificar una dirección en este memoria bidimensional segmentada, el programa debe proporcionar una dirección con dos partes: un número de segmento y una dirección dentro de ese segmento. La figura 6-8 ilustra una memoria segmentada que se usa para las tablas de compilador antes mencionadas.

Hacemos hincapié en que un segmento es una entidad lógica, de cuya existencia el programador es consciente, y que usa como una sola entidad lógica. Un segmento podría contener un procedimiento, un arreglo, una pila o una colección de variables escalares, pero por lo regular no contiene una mezcla de diferentes tipos.

Las memorias segmentadas tienen otras ventajas además de simplificar el manejo de estructuras de datos que crecen o se encogen. Si cada procedimiento ocupa un segmento distinto, y la dirección 0 es su dirección de inicio, se simplifica considerablemente la vincu-

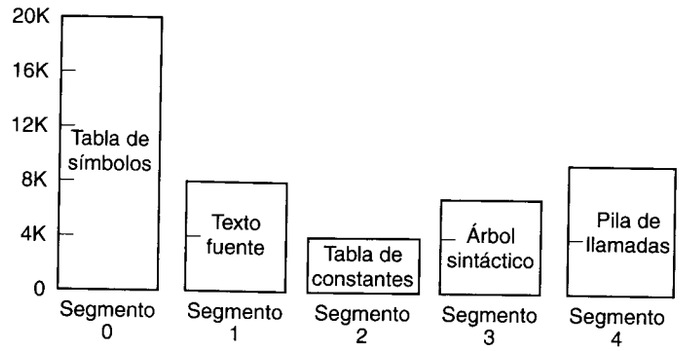


Figura 6-8. Una memoria segmentada permite a cada tabla crecer o encogerse con independencia de las otras tablas.

lación o ligado (*linking*) de procedimientos que se compilan por separado. Una vez que se han compilado y vinculado todos los procedimientos que constituyen un programa, una llamada al procedimiento en el segmento n usará la dirección de dos partes $(n, 0)$ para direccionar la palabra 0 (el punto de ingreso).

Si el procedimiento que está en el segmento n se modifica y recompila posteriormente, no será necesario modificar ningún otro procedimiento (porque no se ha cambiado ninguna dirección de inicio), aunque la nueva versión sea más grande que la antigua. Con una memoria unidimensional, los procedimientos normalmente se empaquetan muy juntos, sin espacio de direcciones entre ellos. Por tanto, la modificación del tamaño de un procedimiento puede afectar la dirección de inicio de otros procedimientos no relacionados. Esto, a su vez, requiere modificar todos los procedimientos que invocan cualquiera de los procedimientos desplazados, a fin de incorporar sus nuevas direcciones de inicio. Si un programa contiene cientos de procedimientos, este proceso puede ser costoso.

La segmentación también facilita compartir procedimientos o datos entre varios programas. Si una computadora tiene varios programas ejecutándose en paralelo (sea procesamiento en paralelo verdadero o simulado), y todos usan ciertos procedimientos de biblioteca, sería un desperdicio de memoria principal proporcionar a cada uno su copia privada. Si se hace que cada procedimiento esté en un segmento distinto, es fácil compartirlos y se elimina la necesidad de tener más de una copia física de cualquier procedimiento compartido en la memoria principal. El resultado es un ahorro de memoria.

Puesto que cada segmento forma una entidad lógica de la que el programador tiene conocimiento, como un procedimiento, un arreglo o una pila, los diferentes segmentos pueden tener diferentes tipos de protección. Un segmento de procedimiento podría especificarse como "sólo para ejecución", lo que impediría que alguien lo lea o escriba en él. Un arreglo de punto flotante podría especificarse como de lectura/escritura pero no para ejecución; con ello se detectaría cualquier intento por saltar a él. Este tipo de protección suele ser útil para detectar errores de programación.

Es importante entender por qué la protección tiene sentido en una memoria segmentada pero no en una memoria paginada unidimensional (es decir, lineal). En una memoria

segmentada el usuario sabe qué hay en cada segmento. Normalmente, un segmento no contiene un procedimiento y una pila, por ejemplo, sino sólo uno de los dos. Puesto que cada segmento contiene un solo tipo de objeto, el segmento puede tener la protección apropiada para ese tipo específico. En la figura 6-9 se comparan la paginación y la segmentación.

Consideración	Paginación	Segmentación
¿El programador necesita saber que existe?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿El espacio de direcciones virtual puede exceder el tamaño de la memoria?	Sí	Sí
¿Es fácil manejar tablas con tamaño variable?	No	Sí
¿Por qué se inventó esta técnica?	Para simular memorias grandes	Para tener muchos espacios de direcciones

Figura 6-9. Comparación de paginación y segmentación.

El contenido de una página es, en cierto sentido, accidental. El programador ni siquiera se da cuenta de que está ocurriendo paginación. Aunque sería posible incluir unos cuantos bits en cada entrada de la tabla de páginas para especificar el acceso permitido, si el programador quiere aprovechar esta capacidad tendría que saber en qué parte del espacio de direcciones están las fronteras de las páginas. Lo malo de tal idea es que la paginación se inventó precisamente para hacer innecesario ese tipo de administración. Puesto que el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos están en la memoria principal todo el tiempo, pueden direccionarse sin preocuparse por la administración de su superposición.

6.1.7 Implementación de la segmentación

Hay dos formas de implementar la segmentación: intercambio y paginación. En el primer esquema, cierto conjunto de segmentos está en la memoria en un momento dado. Si se hace referencia a un segmento que no está en la memoria, se trae ese segmento. Si no hay espacio para él, será preciso escribir primero en el disco uno o más segmentos (a menos que ya exista ahí una copia limpia, en cuyo caso puede abandonarse la copia que está en la memoria). En cierto sentido, el intercambio de segmentos es parecido a la paginación por demanda: los segmentos entran y salen conforme se van necesitando.

Sin embargo, la implementación de la segmentación difiere de la de la paginación en un aspecto fundamental: las páginas tienen un tamaño fijo y los segmentos no. La figura 6-10(a) muestra un ejemplo de memoria física que inicialmente contiene cinco segmentos. Considere ahora qué sucede si el segmento 1 se expulsa y se coloca en su lugar el segmento 7, que es más pequeño. Llegamos a la configuración de memoria de la figura 6-10(b). Entre el segmento 7 y el segmento 2 hay un área desocupada, es decir, un hueco. Luego el segmento 4 es sustituido por el segmento 5, como en la figura 6-10(c), y el segmento 3 es sustituido por el segmento 6, como en la figura 6-10(d). Después de un rato de funcionar el sistema, la memoria estará dividida en varios trozos, algunos con segmentos y otros con huecos. Este fenómeno se llama **fragmentación externa** (porque se desperdicia espacio que está fuera de los segmentos, en los huecos que hay entre ellos). La fragmentación externa también se conoce como **cuadriculado**.

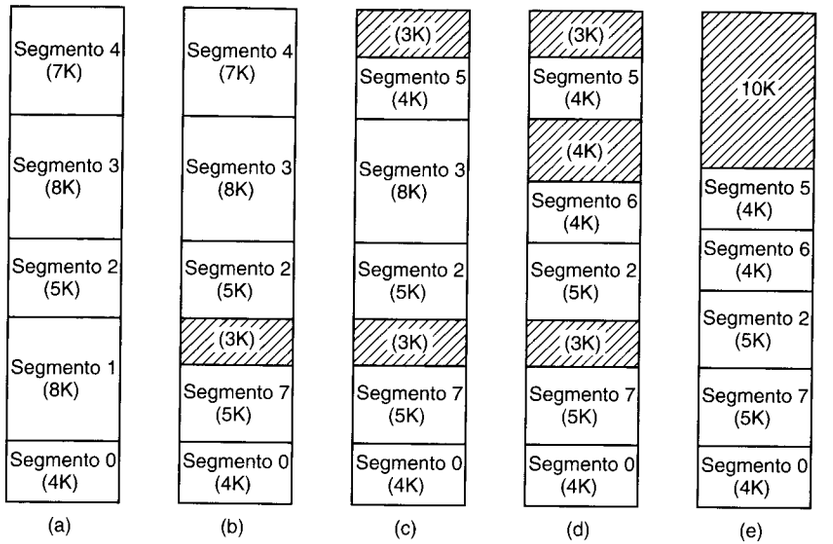


Figura 6-10. (a)-(d) Aparición de fragmentación externa. (e) Eliminación de la fragmentación externa por compactación.

Considere qué sucedería si el programa hiciera referencia al segmento 3 cuando la memoria está experimentando fragmentación externa, como en la figura 6-10(d). El espacio total que ocupan los huecos es de 10K, más que suficiente para el segmento 3, pero como el espacio está distribuido en pequeños fragmentos inútiles no podemos cargar simplemente el segmento 3; será necesario quitar primero algún otro segmento.

Una forma de evitar fragmentación externa es la siguiente: cada vez que aparezca un hueco, los segmentos que están después del hueco se desplazan para acercarlos a la posición de memoria 0 y así eliminar ese hueco y dejar un hueco grande al final. Como alternativa, podríamos esperar hasta que la fragmentación externa sea grave (por ejemplo, más de cierto porcentaje de la memoria total desperdiciado en huecos) antes de realizar la compactación (eliminación de huecos). La figura 6-10(e) muestra cómo quedaría la memoria de la figura 6-10(d) después de una compactación. La intención de compactar la memoria es reunir todos los pequeños huecos inútiles en un solo hueco grande, en el que será posible colocar uno o más segmentos. La compactación tiene la desventaja obvia de que se desperdicia cierto tiempo para llevarla a cabo. En general, no se puede compactar después de que se crea cada hueco, pues se perdería demasiado tiempo.

Si el tiempo requerido para compactar la memoria es demasiado largo, se requiere un algoritmo para determinar cuál hueco deberá usarse para un segmento dado. La gestión de

huecos requiere mantener una lista de las direcciones y tamaños de todos los huecos. Un algoritmo muy popular, llamado **de mejor ajuste**, escoge el hueco más pequeño en el que cabe el segmento requerido. La idea es equiparar los segmentos con los huecos y así evitar gastar un trozo de un agujero grande, que podría necesitarse después para un segmento grande.

Otro algoritmo muy utilizado, llamado **de primer ajuste**, explora circularmente la lista de agujeros y escoge el primer agujero que es lo bastante grande como para albergar el segmento. Esto obviamente toma menos tiempo que examinar toda la lista hasta encontrar el mejor ajuste. Resulta sorprendente que el de primer ajuste es también un mejor algoritmo en términos de desempeño global que el de mejor ajuste, porque este último tiende a generar muchos huecos pequeños totalmente inútiles (Knuth, 1997).

El primero y mejor ajuste tiende a reducir el tamaño medio de los huecos. Cada vez que un segmento se coloca en un hueco mayor que él, lo que sucede casi siempre (los ajustes exactos son raros), el agujero se divide en dos partes. Una parte la ocupa el segmento y la otra es el nuevo hueco. Éste siempre es más pequeño que el hueco viejo. A menos que exista un proceso compensador que vuelva a crear huecos grandes a partir de huecos pequeños, tanto el primer ajuste como el mejor ajuste tarde o temprano llenarán la memoria con huecos pequeños e inútiles.

Uno de esos procesos compensadores es el siguiente. Cada vez que un segmento se saca de la memoria y uno de sus vecinos inmediatos, o ambos, son huecos, no segmentos, los huecos adyacentes se pueden juntar en un hueco grande. Si el segmento 5 se quitara de la figura 6-10(d), los dos huecos que lo flanquean y los 4K utilizados por el segmento se fusionarían en un solo hueco de 11K.

Al principio de esta sección dijimos que hay dos formas de implementar la segmentación: intercambio y paginación. Nuestra explicación hasta ahora se ha centrado en el intercambio. Con este esquema, segmentos enteros se transfieren entre la memoria y el disco por demanda. La otra forma de implementar la segmentación es dividir cada segmento en páginas de tamaño fijo y paginarlas por demanda. En este esquema, algunas de las páginas de un segmento podrían estar en la memoria y otras en disco. Para paginar un segmento se requiere una tabla de páginas individual para cada segmento. Puesto que un segmento no es más que un espacio de direcciones lineal, todas las técnicas que hemos visto hasta ahora para la paginación se aplican a cada segmento. La única novedad aquí es que cada segmento tiene su propia tabla de páginas.

Uno de los primeros sistemas operativos que combinaron la segmentación con la paginación fue **MULTICS (Sistema de Información y Cómputo Multiplexado**, *MULtiplexed Information and Computing Service*), que inicialmente fue un proyecto conjunto de MIT, Bell Labs y General Electric (Corbató y Vyssotsky, 1995; y Organick, 1972). Las direcciones de MULTICS tenían dos partes: un número de segmento y una dirección dentro del segmento. Había un segmento de descriptores para cada proceso, que contenía un descriptor para cada segmento. Cuando se presentaba una dirección virtual al hardware, se usaba el número de segmento como índice del segmento de descriptores para localizar el descriptor del segmento accesado, como se muestra en la figura 6-11. El descriptor apuntaba a la tabla de páginas, la que permitía paginar cada segmento de la forma acostumbrada. Para mejorar el desempeño, las combinaciones segmento/página de uso más reciente se guardaban en una **memoria asociativa** de 16 entradas en hardware que permitía buscarlas rápidamente. Aun-

que hace mucho que MULTICS desapareció, su espíritu sigue vivo porque la memoria virtual de todas las CPU de Intel a partir del 386 ha seguido de cerca su modelo.

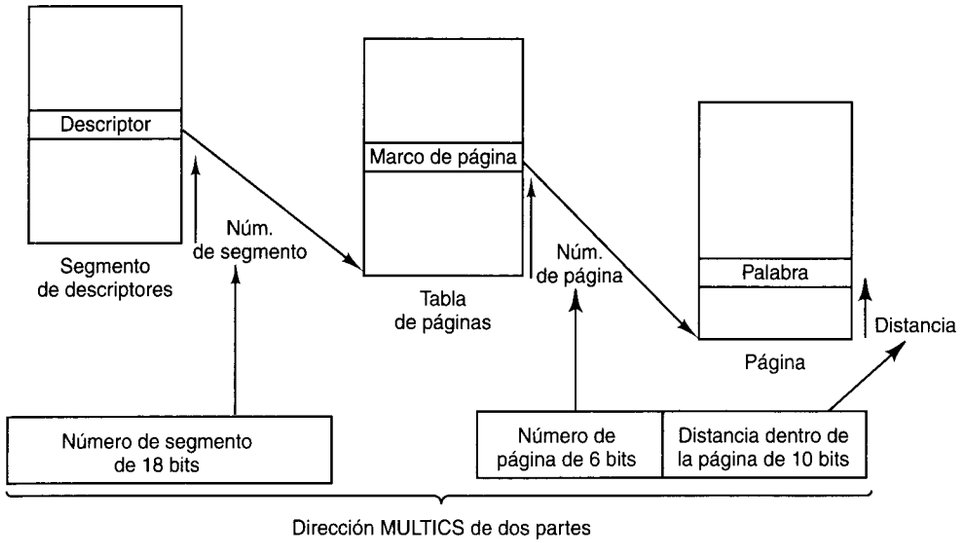


Figura 6-11. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

6.1.8 Memoria virtual en el Pentium II

El Pentium II tiene un sistema de memoria virtual avanzado que maneja paginación por demanda, segmentación pura y segmentación con paginación. El corazón de la memoria virtual del Pentium II consiste de dos tablas: la **tabla de descriptores locales** (**LDT**, *Local Descriptor Table*) y la **tabla de descriptores globales** (**GDT**, *Global Descriptor Table*). Cada programa tiene su propia LDT, pero hay una sola GDT compartida por todos los programas de la computadora. La LDT describe los segmentos locales de cada programa, incluidos los de código, datos, pila, etc., mientras que la GDT describe los segmentos del sistema, incluido el sistema operativo mismo.

Como explicamos en el capítulo 5, para que un programa de Pentium II accese a un segmento, primero debe cargar un selector de ese segmento en uno de sus registros de segmento. Durante la ejecución, CS contiene el selector del segmento de código, DS contiene el selector del segmento de datos, etc. Cada selector es un número de 16 bits, como se muestra en la figura 6-12.

Uno de los bits del selector indica si el segmento es local o global (es decir, si está en la LDT o en la GDT). Otros 13 bits especifican el número de entrada de la LDT o GDT, así que estas tablas sólo pueden contener 8K (2^{13}) descriptores de segmentos. Los otros dos bits

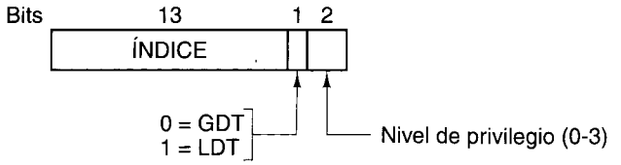


Figura 6-12. Selector de Pentium II.

tienen que ver con la protección y se describirán después. El descriptor 0 no es válido y su uso causa una trampa. Puede cargarse sin peligro en un registro de segmento para indicar que ese registro no está disponible, pero causa una trampa si se usa.

Cuando un selector se carga en un registro de segmento, el descriptor correspondiente se trae de la LDT o GDT y se guarda en registros internos de la MMU, para poder acceder a él rápidamente. Un descriptor consiste en 8 bytes e incluye la dirección base del segmento, su tamaño y otra información, como se muestra en la figura 6-13.

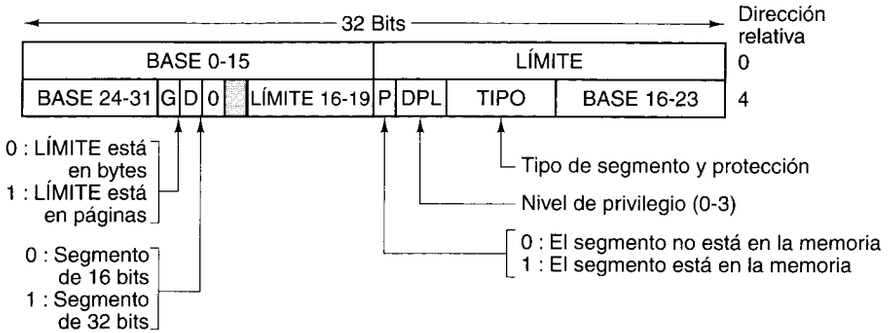


Figura 6-13. Descriptor de segmento de código de Pentium II. Los segmentos de datos son un poco diferentes.

El formato del selector se escogió ingeniosamente a modo de facilitar la localización del descriptor. Primero se selecciona la LDT o bien la GDT, con base en el bit 2 del selector. Luego el selector se copia en un registro de borrador de la MMU y los tres bits de orden bajo se ponen en 0, con lo que el número de selector de 13 bits se multiplica efectivamente por 8. Por último, se le suma la dirección de la tabla LDT o GDT (que se guarda en registros internos de la MMU) para dar un apuntador directo al descriptor. Por ejemplo, el selector 72 se refiere a la entrada 9 de la GDT, que se encuentra en la dirección GDT + 72.

Sigamos los pasos por los que un par (selector, distancia) se convierte en una dirección física. Tan pronto como el hardware sabe cuál registro de segmento se está usando, puede encontrar el descriptor completo que corresponde a ese selector en sus registros internos. Si el segmento no existe (selector 0) o no está en la memoria (P es 0), ocurre una trampa. El primer caso es un error de programación; el segundo obliga al sistema operativo a obtener el segmento.

Luego, el sistema verifica si la distancia rebasa el final del segmento, en cuyo caso ocurre otra trampa. Lógicamente, debería haber un campo de 32 bits en el descriptor que diera el

tamaño del segmento, pero sólo se cuenta con 20 bits, así que se emplea un esquema distinto. Si el campo G (Granularidad) es 0, el campo LÍMITE da el tamaño exacto del segmento, hasta 1 MB; si es 1, LÍMITE da el tamaño del segmento en páginas en lugar de bytes. El tamaño de página del Pentium II nunca es menor que 4 KB, por lo que 20 bits son suficientes para segmentos de hasta 2^{32} bytes.

Suponiendo que el segmento está en la memoria y la distancia no se sale del intervalo, el Pentium II suma el campo de 32 bits BASE del descriptor a la distancia para formar una **dirección lineal**, como se muestra en la figura 6-14. El campo BASE se descompone en tres fragmentos que se distribuyen por todo el descriptor por razones de compatibilidad con el 80286, en el que BASE sólo tiene 24 bits. Así pues, el campo BASE permite que un segmento inicie en cualquier punto dentro del espacio de direcciones lineal de 32 bits.

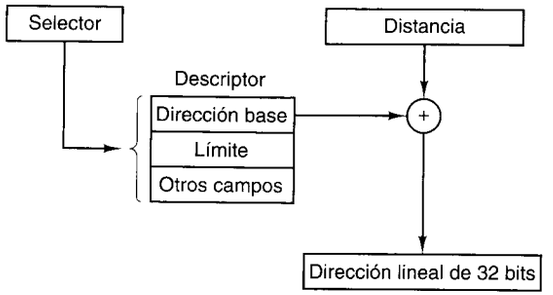


Figura 6-14. Conversión de un par (selector, distancia) en una dirección lineal.

Si se inhabilita la paginación (con un bit en un registro de control global), la dirección lineal se interpreta como la dirección física y se envía a la memoria para efectuar la lectura o escritura. Así, con la paginación desactivada, tenemos un esquema de segmentación pura, y la dirección base de cada segmento está dada en su descriptor. Por cierto, se permite el traslapo de segmentos, tal vez porque sería demasiado problemático y tardado verificar que todos sean disjuntos.

Por otra parte, si la paginación está habilitada la dirección lineal se interpreta como una dirección virtual y se transforma en una dirección física mediante tablas de páginas, prácticamente igual que en nuestros ejemplos. La única complicación es que con una dirección virtual de 32 bits y páginas de 4 K, un segmento podría contener un millón de páginas, por lo que se usa un mapeo de dos niveles para reducir el tamaño de la tabla de páginas cuando los segmentos son pequeños.

Cada programa en ejecución tiene un **directorio de páginas** que consta de 1024 entradas de 32 bits y se encuentra en una dirección a la que un registro global apunta. Cada entrada de este directorio apunta a una tabla de páginas que también contiene 1024 entradas de 32 bits. Las entradas de la tabla de páginas apuntan a marcos de página. El esquema se muestra en la figura 6-15.

En la figura 6-15(a) vemos una dirección lineal desglosada en tres campos: DIRECTORIO, PÁGINA y DISTANCIA. El campo DIRECTORIO se usa primero como índice del directorio de páginas para localizar un apuntador a la tabla de páginas apropiada. Luego se

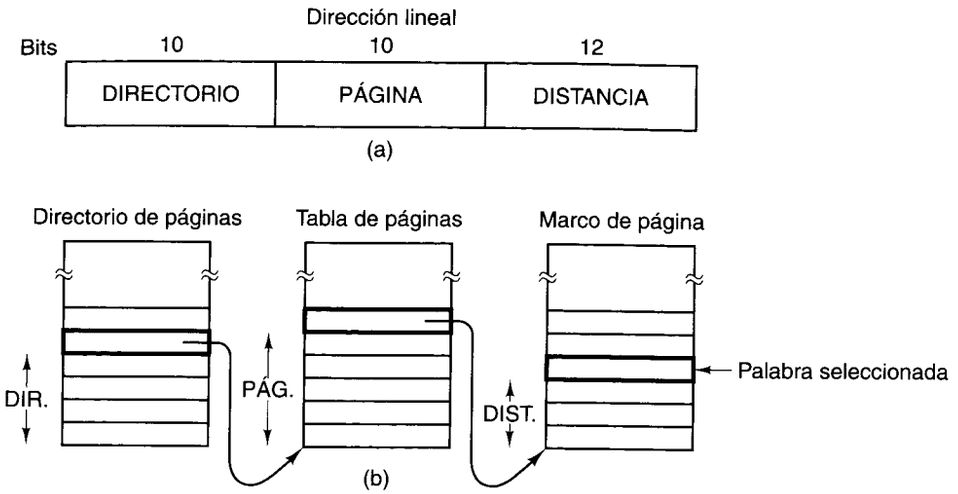


Figura 6-15. Transformación de una dirección lineal en una dirección física.

usa el campo PÁGINA como índice de la tabla de páginas para encontrar la dirección física del marco de página. Por último, DISTANCIA se suma a la dirección del marco de página para obtener la dirección física del byte o palabra direccionado.

Las entradas de la tabla de páginas tienen 32 bits cada una, de los cuales 20 contienen un número de marco de página. Los demás bits controlan el acceso e indican si la página está limpia o no. El hardware ajusta estos bits que son utilizados por el sistema operativo, los bits de protección y otros bits utilitarios.

Cada tabla de páginas tiene entradas para 1024 marcos de página de 4K, por lo que una sola tabla de páginas maneja 4 MB de memoria. Un segmento menor que 4M tendrá un directorio de páginas con una sola entrada, un apuntador a su única tabla de páginas. Así, el gasto extra en el caso de un segmento corto es de sólo dos páginas, en lugar del millón de páginas que se necesitaría en una tabla de páginas de un solo nivel.

A fin de evitar referencias repetidas a la memoria, la MMU del Pentium II cuenta con hardware especial que busca las combinaciones DIRECTORIO-PÁGINA utilizadas más recientemente y las transforma en la dirección física del marco de página correspondiente. Sólo se llevan a cabo los pasos de la figura 6-15 si la combinación actual no se usó recientemente.

Si nos ponemos a pensar un poco, veremos que si se usa paginación en realidad no tiene caso que el campo BASE del descriptor sea distinto de cero. Para lo único que sirve BASE es para desplazarse un poco dentro del directorio de páginas a fin de usar una entrada en ese punto en lugar de al principio. La verdadera razón para incluir BASE es hacer posible la segmentación pura (no paginada), y mantener la compatibilidad con el viejo 80286, que no tenía paginación.

También vale la pena señalar que si una aplicación dada no necesita segmentación y se conforma con un solo espacio de direcciones paginado de 32 bits, es fácil satisfacerla. Se asigna

a todos los registros de segmento el mismo selector, cuyo descriptor tiene BASE = 0 y LÍMITE igual al máximo. Entonces, la distancia de la instrucción será la dirección lineal, empleando un solo espacio de direcciones; esto es, de hecho, paginación tradicional.

Con esto terminamos nuestro tratamiento de la memoria virtual en el Pentium II. Sin embargo, vale la pena hablar un poco de protección, ya que este tema está íntimamente relacionado con la memoria virtual. El Pentium II maneja cuatro niveles de protección, siendo el nivel 0 el que goza de más privilegios, y el nivel 3, el que menos. Dichos niveles se muestran en la figura 6-16. En un instante dado, un programa en ejecución está en cierto nivel, indicado por un campo de dos bits en su **palabra de estado del programa (PSW)**, un registro en hardware que contiene los códigos de condición y varios otros bits de estado. Además, cada segmento del sistema pertenece a un nivel dado.

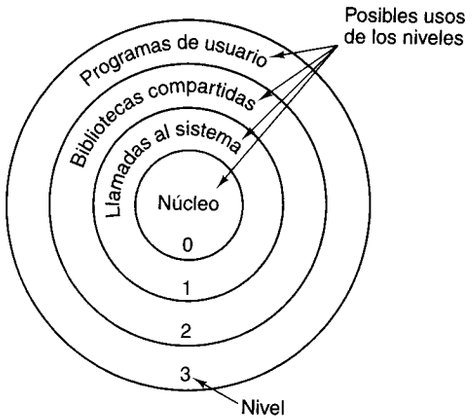


Figura 6-16. Protección en el Pentium II.

En tanto un programa se limite a usar segmentos que estén en su mismo nivel, todo funcionará de maravilla. Se permiten los intentos de accesar a datos que están en un nivel más alto, pero los intentos de accesar a datos en un nivel inferior están prohibidos y causan trampas. Los intentos por invocar procedimientos de otro nivel (más alto o más bajo) están permitidos, pero de forma muy controlada. Para efectuar una llamada internivel, la instrucción CALL debe contener un selector en lugar de una dirección. Este selector designa un descriptor llamado **puerta de llamada**, que da la dirección del procedimiento que se desea invocar. Así, no es posible saltar a la mitad de un segmento de código arbitrario en un nivel distinto; sólo pueden usarse los puntos de ingreso oficiales.

En la figura 6-16 se sugiere un posible uso de este mecanismo. En el nivel 0 está el núcleo del sistema operativo, que se encarga de la E/S, la gestión de memoria y otras tareas cruciales. En el nivel 1 está el manejador de llamadas al sistema. Los programas de usuario pueden invocar procedimientos de este nivel para solicitar que se efectúen llamadas al sistema, pero sólo es posible invocar una lista específica y protegida de procedimientos. El nivel 2 contiene procedimientos de biblioteca, posiblemente compartidos entre muchos programas

en ejecución. Los programas de usuario pueden invocar estos procedimientos, pero no modificarlos. Por último, los programas de usuario se ejecutan en el nivel 3, que es el que menos protección tiene. Al igual que el esquema de gestión de memoria del Pentium II, el sistema de protección se basa en el de MULTICS.

Las trampas e interrupciones usan un mecanismo similar a las puertas de llamada; también hacen referencia a descriptores, no a direcciones absolutas, y dichos descriptores apuntan a procedimientos específicos que deben ejecutarse. El campo TIPO de la figura 6-13 distingue entre segmentos de código, segmentos de datos y los diversos tipos de puertas.

6.1.9 Memoria virtual en el UltraSPARC II

El UltraSPARC II es una máquina de 64 bits y maneja una memoria virtual paginada basada en una dirección virtual de 64 bits. Sin embargo, por razones de ingeniería y costos, los programas no pueden usar todo el espacio de direcciones virtual de 64 bits. Sólo se reconocen 44 bits, por lo que los programas no pueden exceder 1.8×10^{13} bytes. La memoria virtual permitida se divide en dos zonas de 2^{43} bytes cada una, una en la parte superior del espacio de direcciones virtual y una en la parte inferior. En medio hay un hueco que contiene direcciones que no pueden usarse; un intento por usarlas causa un fallo de página.

La memoria física máxima en un UltraSPARC II es de 2^{41} bytes, que equivale a unos 2200 GB, lo bastante grande para casi todas las aplicaciones ordinarias. Se reconocen cuatro tamaños de página: 8 KB, 64 KB, 512 KB y 4 MB. En la figura 6-17 se ilustran las correspondencias implícitas en estos cuatro tamaños de página.

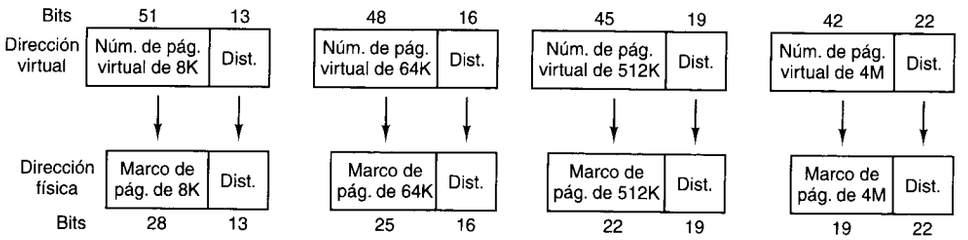


Figura 6-17. Transformaciones virtual-física en el UltraSPARC II.

Por lo extremadamente grande de espacio de direcciones virtual, una tabla de páginas sencilla como la del Pentium II no sería práctica. En vez de ello, la MMU del UltraSPARC II adopta una estrategia muy diferente: contiene una tabla en hardware llamada **buffer de consulta para traducción** (**TLB**, *Translation Lookaside Buffer*) que transforma números de página virtual en números de marco de página físico. Para el tamaño de página de 8K hay 2^{31} números de página virtual, o sea, más de dos millones. Es evidente que no todos pueden estar en el mapa.

En vez de ello, el TLB sólo contiene los números de página virtual más recientemente usados. Se sigue la pista por separado a las páginas de instrucciones y a las de datos, y el TLB guarda los 64 números de página virtual más recientemente usados de cada categoría. Cada entrada del TLB contiene un número de página virtual y el número de marco de página físico correspondiente. Cuando se presentan a la MMU un número de proceso, llamado **contexto**, y

una dirección virtual dentro de ese contexto, la MMU emplea circuitos especiales para comparar simultáneamente el número de página virtual contenido en la dirección virtual con todas las entradas del TLB. Si se encuentra una concordancia, el número de marco de página contenido en esa entrada del TLB se combina con la distancia tomada de la dirección virtual para formar una dirección física de 41 bits y producir algunas banderas, como los bits de protección. El TLB se ilustra en la figura 6-18(a).

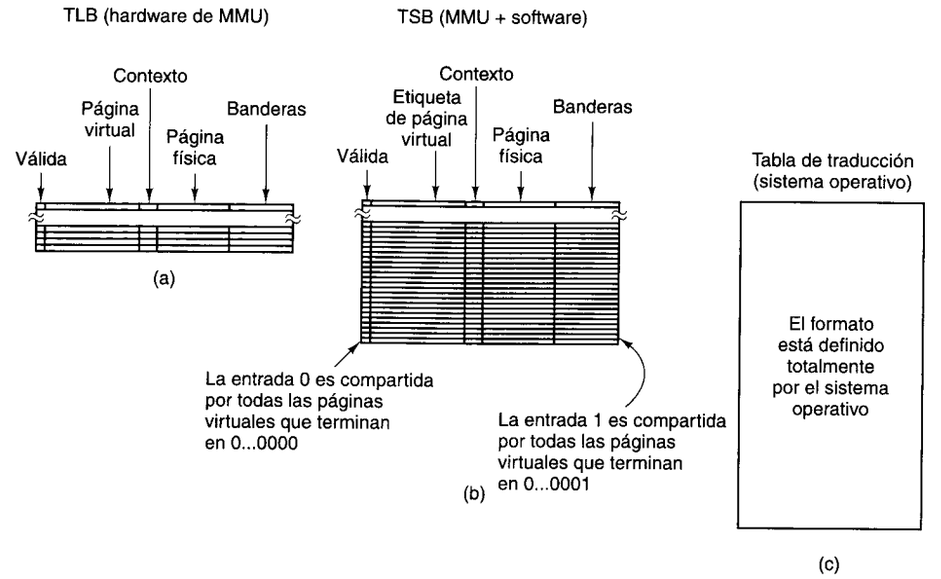


Figura 6-18. Estructuras de datos empleadas para traducir direcciones virtuales en el UltraSPARC. (a) TLB. (b) TSB. (c) Tabla de traducción.

Si ninguna de las entradas de la TLB concuerda, ocurre un **fallo de TLB**, que causa un salto al sistema operativo a través de una trampa. El manejo del fallo corresponde al sistema operativo. Cabe señalar que un fallo de TLB es muy diferente de un fallo de página. Puede ocurrir un fallo de TLB aunque la página a la que se hizo referencia esté en la memoria. En teoría, el sistema operativo puede hacer lo que quiera para cargar una nueva entrada de TLB para la página virtual requerida. Sin embargo, para acelerar esta operación crucial, se puede contar con cierta ayuda del hardware si es que el software coopera.

En particular, se espera que el sistema operativo mantenga una caché en software con las entradas de TLB más utilizadas en una tabla llamada **buffer de almacenamiento para traducción** (**TSB**, *Translation Storage Buffer*). Esta tabla se organiza como caché de páginas virtuales con correspondencia directa. Cada entrada de TSB de 16 bytes se refiere a una página virtual y contiene un bit de validez, el número de contexto, la etiqueta de dirección virtual, el número de página física y algunos bits que sirven como banderas. Si el tamaño del caché es de, digamos, 8192 entradas, todas las páginas virtuales cuyos 13 bits de orden bajo sean 0000000000000 competirán por la entrada 0 del TSB. Así mismo, todas las páginas virtuales cuyos bits de orden bajo sean 0000000000001 competirán por la entrada 1 del TSB,

como se muestra en la figura 6-18(b). El tamaño del TSB lo determina el software y se comunica a la MMU por medio de registros especiales a los que sólo el sistema operativo tiene acceso.

Cuando ocurre un fallo de TLB, el sistema operativo verifica si la entrada correspondiente en el TSB contiene la página virtual requerida. La MMU ayuda aquí calculando la dirección de esta entrada y colocándola en un registro interno MMU accesible para el sistema operativo. Si ocurre un acierto de caché en el TSB, se borrará alguna entrada del TLB y la entrada de TSB requerida se copiará en el TLB. El hardware también ayuda a escoger cuál entrada del TLB se borrará utilizando un algoritmo LRU de un bit.

Si la búsqueda en el TSB no corre con suerte y la página virtual a la que se hizo referencia no está en la caché, el sistema operativo usa otra tabla para localizar la información acerca de la página, que podría o no estar en la memoria principal. La tabla que se usa para esta consulta de último recurso se llama **tabla de traducción**. Puesto que el hardware no ayuda con las consultas a esta tabla, el sistema operativo está en libertad de usar el formato que quiera. Por ejemplo, puede usar *hashing* dividiendo el número de página virtual entre algún número primo *p*, y usar el residuo como índice de una tabla de apuntadores, cada uno de los cuales apunta a una lista enlazada de entradas de página virtual que dan *p* por *hashing*. Cabe señalar que estas entradas no son las páginas, sino entradas de TSB. El resultado de buscar una página en la tabla de traducción podría ser localizar la página en la memoria, en cuyo caso se actualizará la entrada de TSB de la caché en software, pero también podría ser el descubrimiento de que la página no está en la memoria, en cuyo caso se inicia la acción estándar de fallo de página.

Resulta interesante comparar los esquemas de paginación del Pentium II y el UltraSPARC II. El Pentium II maneja segmentación pura, paginación pura y segmentos paginados. El UltraSPARC II sólo tiene paginación. El Pentium II también usa hardware para recorrer la tabla de páginas y volver a cargar el TLB en caso de un fallo de TLB. El UltraSPARC II simplemente transfiere el control al sistema operativo cuando hay un fallo de TLB.

La razón primordial de esta diferencia es que el Pentium II usa segmentos de 32 bits, y tales segmentos tan pequeños (sólo un millón de páginas) pueden manejarse con tablas de páginas convencionales. En teoría, el Pentium II tendría problemas si un programa usara miles de segmentos, pero dado que ninguna versión de Windows ni de UNIX reconoce más de un segmento por proceso, el problema no se presenta. En contraste, el UltraSPARC II es una máquina de 64 bits y puede tener hasta 2000 millones de páginas, por lo que las tablas de páginas convencionales no funcionan. En el futuro, todas las máquinas tendrán espacios de direcciones virtuales de 64 bits, por lo que esquemas como el del UltraSPARC II se convertirán en la norma. En (Jacob y Mudge, 1998b) se hace una comparación del Pentium II, el UltraSPARC II y otros cuatro esquemas de memoria virtual.

6.1.10 Memoria virtual y uso de cachés

Aunque a primera vista no parece haber relación entre la memoria virtual (paginada por demanda) y el uso de cachés, en lo conceptual son muy similares. Con memoria virtual todo el programa se mantiene en disco y se divide en páginas de tamaño fijo. Un subconjunto de

estas páginas está en la memoria principal. Si el programa usa casi todo el tiempo las páginas que están en la memoria, habrá pocos fallos de página y el programa trabajará rápidamente. Con cachés, todo el programa se mantiene en la memoria principal y se divide en bloques de caché de tamaño fijo. Un subconjunto de estos bloques está en el caché. Si el programa usa casi todo el tiempo los bloques que están en el caché, habrá pocos fallos de caché y el programa trabajará rápidamente. En lo conceptual, las dos cosas son idénticas, excepto que operan en diferentes niveles de la jerarquía.

Desde luego, la memoria virtual y el uso de cachés también tienen ciertas diferencias. Una de ellas es que los fallos de caché son manejados por el hardware, mientras que los fallos de página son manejados por el sistema operativo. Además, los bloques de caché suelen ser mucho más pequeños que las páginas (por ejemplo, 64 bytes *versus* 8 KB). Además, la correspondencia entre páginas virtuales y marcos de página es diferente, y las tablas de páginas se organizan usando como índice los bits de orden alto de la dirección virtual, mientras que las cachés usan como índice los bits de orden bajo de la dirección de memoria. Pese a esto, es importante darse cuenta de que éstas son diferencias de implementación. El concepto subyacente es muy similar.

6.2 INSTRUCCIONES DE E/S VIRTUALES

El conjunto de instrucciones en el nivel ISA es totalmente distinto del conjunto de instrucciones de la microarquitectura. Tanto las operaciones que pueden efectuarse como los formatos de las instrucciones son muy diferentes en los dos niveles. La existencia de unas cuantas instrucciones que son iguales en ambos niveles es en esencia accidental.

En contraste, el conjunto de instrucciones del nivel OSM contiene casi todas las instrucciones del nivel ISA, con la adición de unas cuantas instrucciones nuevas pero importantes y la eliminación de unas cuantas instrucciones que podrían ser perjudiciales. La entrada/salida es una de las áreas en la que los dos niveles difieren considerablemente. La razón de esta diferencia es sencilla: un usuario que puede ejecutar las verdaderas instrucciones de E/S del nivel ISA podría leer datos confidenciales almacenados en cualquier lugar del sistema, escribir en las terminales de otros usuarios y, en general, causar muchos estropicios además de ser una amenaza para la seguridad del sistema. Además, los programadores normales que están en sus cabales no quieren efectuar ellos mismos la E/S en el nivel ISA porque ello es en extremo tedioso y complejo. Se efectúa asignando valores a campos y bits en diversos registros de dispositivo, esperando hasta que la operación se lleva a cabo, y verificando después qué sucedió. Como ejemplo de esto último, los discos por lo regular tienen bits de registros de dispositivo que detectan los siguientes errores, entre muchos otros:

1. El brazo del disco no realizó correctamente la búsqueda.
2. Se especificó memoria inexistente como buffer.
3. La E/S de disco inició antes de que terminara la anterior.
4. Error de temporización al leer.
5. Se direccionó un disco inexistente.

6. Se direccionó un cilindro inexistente.
7. Se direccionó un sector inexistente.
8. Error de suma de verificación al leer.
9. Error de verificación de escritura después de operación de escritura.

Cuando ocurre uno de estos errores, se enciende el bit correspondiente de un registro de dispositivo. Pocos usuarios quieren ocuparse de seguir la pista a todos estos bits de error y a una gran cantidad de información de estado adicional.

6.2.1 Archivos

Una forma de organizar la E/S virtual es usar una abstracción llamada **archivo**. En su forma más sencilla, un archivo consiste en una secuencia de bytes que se escriben en un dispositivo de E/S. Si el dispositivo de E/S es un dispositivo de almacenamiento, como un disco, el archivo se podrá leer posteriormente; claro que si no se trata de un dispositivo de almacenamiento (por ejemplo, una impresora), no podrá leerse. Un disco puede contener muchos archivos, cada uno con un tipo de datos específico; por ejemplo, una imagen, una hoja de cálculo o el texto del capítulo de un libro. Los diferentes archivos tienen diferentes longitudes y otras propiedades. La abstracción de archivo permite organizar la E/S virtual de una forma sencilla.

Para el sistema operativo, un archivo normalmente es sólo una secuencia de bytes, como dijimos antes. Cualquier estructura adicional es asunto de los programas de aplicación. La E/S con archivos se efectúa con llamadas al sistema para abrir, leer, escribir y cerrar archivos. Antes de leer un archivo, es preciso abrirlo. El proceso de abrir un archivo permite al sistema operativo localizar el archivo en el disco y traer a la memoria la información necesaria para accesar a él.

Una vez que se ha abierto un archivo, puede leerse. La llamada al sistema read debe tener, como mínimo, los siguientes parámetros:

1. Una indicación de cuál archivo abierto debe leerse.
2. Un apuntador a un buffer en la memoria en el que se colocarán los datos.
3. El número de bytes que se leerán.

La llamada read coloca los datos solicitados en el buffer, y por lo regular devuelve el número de bytes que se leyeron realmente, que podría ser menor que el número solicitado (no es posible leer 2000 bytes de un archivo de 1000 bytes).

Todo archivo abierto tiene asociado un apuntador que indica cuál byte se leerá a continuación. Después de un read, el apuntador se incrementa en el número de bytes leídos, por lo que reads consecutivos leen bloques consecutivos de datos del archivo. Por lo regular hay una forma de asignar a este apuntador un valor específico para que los programas puedan acceder aleatoriamente a cualquier parte del archivo. Cuando un programa termina de leer un archivo, puede cerrarlo para informar al sistema operativo que ya no usará el archivo y que puede liberar el espacio en tablas que se estaban usando para contener la información acerca del archivo.

Los sistemas operativos de *mainframes* tienen una idea más sofisticada de lo que es un archivo. Ahí, un archivo puede ser una sucesión de **registros lógicos**, cada uno con una estructura bien definida. Por ejemplo, un registro lógico podría ser una estructura de datos que consiste en cinco elementos: dos cadenas de caracteres, "Nombre" y "Supervisor"; dos enteros, "Departamento" y "Oficina" y un booleano, "SexoEsFemenino". Algunos sistemas operativos distinguen entre archivos en los que todos los registros tienen la misma estructura y los que contienen una mezcla de diferentes tipos de registros.

La instrucción de entrada virtual básica lee el siguiente registro del archivo especificado y lo coloca en la memoria principal a partir de una dirección especificada, como se ilustra en la figura 6-19. Para realizar esta operación, hay que indicar a la instrucción virtual cuál archivo debe leer y en qué lugar de la memoria debe colocar el registro. Es común que haya opciones para leer un registro en particular, especificado sea por su posición en el archivo o por su clave.

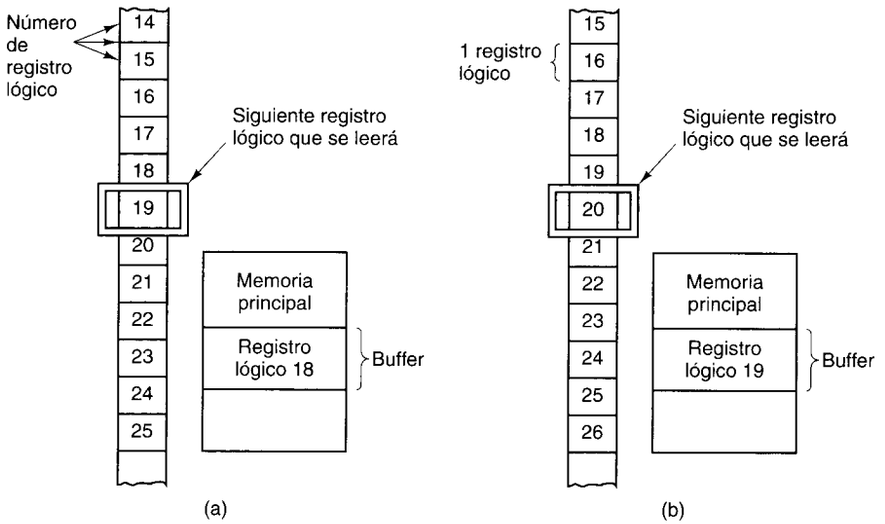


Figura 6-19. Lectura de un archivo que consiste en registros lógicos. (a) Antes de leer el registro 19. (b) Después de leer el registro 19.

La instrucción de salida virtual básica escribe un registro lógico de la memoria en un archivo. Instrucciones write consecutivas producen registros lógicos consecutivos en el archivo.

6.2.2 Implementación de instrucciones de E/S virtuales

Para entender cómo se implementan las instrucciones de E/S virtuales, es necesario examinar la organización y almacenamiento de los archivos. Una cuestión básica que debe resolverse con todos los sistemas de archivos es la asignación de almacenamiento. La unidad de asignación puede ser un solo sector de disco, pero es común que consista en un bloque de sectores consecutivos.

Otra propiedad fundamental de una implementación de sistema de archivos es si un archivo se almacena en unidades de asignación consecutivas o no. La figura 6-20 muestra un disco sencillo con una superficie que consiste en cinco pistas de 12 sectores cada una. La figura 6-20(a) muestra un esquema de asignación en el que el sector es la unidad básica de asignación de espacio y un archivo consiste en sectores consecutivos. Es común usar la asignación consecutiva de bloques de archivo en los CD-ROM. La figura 6-20(b) muestra un esquema de asignación en el que el sector es la unidad básica de asignación pero en el que un archivo no tiene que ocupar sectores consecutivos. Este esquema es la norma en los discos duros.

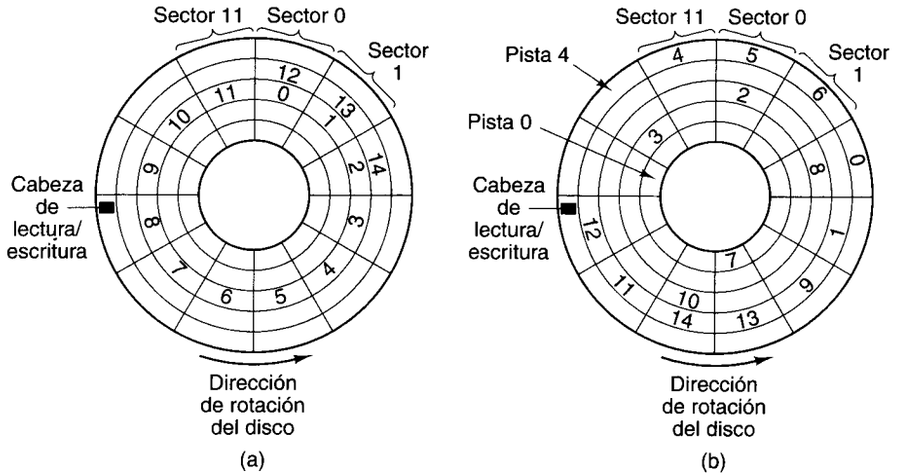


Figura 6-20. Estrategias de asignación de disco. (a) Un archivo en sectores consecutivos. (b) Un archivo en sectores no consecutivos.

Existe una diferencia importante entre la forma como un programador de aplicaciones ve un archivo y la forma en que lo ve el sistema operativo. El programador ve el archivo como una secuencia lineal de bytes o palabras lógicas. El sistema operativo ve el archivo como una colección ordenada, aunque no necesariamente consecutiva, de unidades de asignación en el disco.

Para que el sistema operativo entregue el byte o el registro lógico *n* de algún archivo cuando se le solicite, debe tener algún método para localizar los datos. Si el archivo se asigna consecutivamente, el sistema operativo sólo tiene que conocer la posición del principio del archivo para poder calcular la posición del byte o registro lógico requerido.

Si el archivo no se asigna consecutivamente, no es posible calcular la posición de un byte o registro lógico arbitrario sólo a partir de la posición del principio del archivo. Para localizar cualquier byte o registro lógico se necesita una tabla llamada **índice de archivos** que da las unidades de asignación y sus direcciones de disco reales. El índice de archivos se puede organizar como una lista de direcciones de bloques de disco (como en UNIX) o como lista de registros lógicos, dando la dirección en disco y la distancia para cada uno. A veces cada registro lógico tiene una **clave** y los programas pueden referirse a un registro por su clave, en lugar de por su número de registro lógico. En este caso se requiere la segunda organización, en la que

cada entrada contiene no sólo la ubicación del registro en el disco, sino también su clave. Esta organización es común en las *mainframes*.

Un método alternativo para localizar las unidades de asignación de un archivo es organizar el archivo como lista enlazada. Cada unidad de asignación contiene la dirección de su sucesora. Una forma de implementar este esquema de forma eficiente es guardar en la memoria principal la tabla con las direcciones de todas las sucesoras. Por ejemplo, en el caso de un disco con unidades de asignación de 64K, el sistema operativo podría tener una tabla en la memoria con 64K entradas, cada una de las cuales da el índice de su sucesora. Por ejemplo, si un archivo ocupara las unidades de asignación 4, 52 y 19, la entrada 4 de la tabla contendría un 52, la entrada 52 contendría un 19 y la entrada 19 contendría un código especial (digamos 0 o –1) para indicar el fin del archivo. Los sistemas de archivos empleados por MS-DOS y Windows 95 y Windows 98 funcionan así. Windows NT reconoce este sistema de archivos pero además tiene su propio sistema de archivos nativo que funciona de forma más parecida al de UNIX.

Hasta ahora hemos hablado de archivos asignados consecutiva y no consecutivamente, pero no hemos explicado por qué se usan ambos tipos. La administración de bloques de los archivos asignados consecutivamente es más sencilla, pero cuando no se conoce por adelantado el tamaño máximo del archivo casi nunca es posible usar esta técnica. Si un archivo iniciara en el sector *j* y pudiera crecer hacia sectores consecutivos, podría toparse con otro archivo en el sector *k* y no tener más espacio para expandirse. Si el archivo no se asigna consecutivamente, esta situación no representa un problema, porque los bloques subsecuentes se pueden colocar en cualquier lugar del disco. Si un disco contiene varios archivos que están creciendo, y no se conoce el tamaño final de ninguno de ellos, almacenarlos como archivos consecutivos será casi imposible. A veces es posible cambiar de lugar un archivo existente, pero siempre es costoso.

Por otra parte, si se conoce por adelantado el tamaño máximo de todos los archivos, como sucede cuando se graba un CD-ROM, el programa de grabación puede preasignar una serie de sectores cuya longitud es exactamente igual a la de cada archivo. Así pues, si se van a colocar archivos con longitudes de 1200, 700, 2000 y 900 sectores en un CD-ROM, simplemente se inician en los sectores 0, 1200, 1900 y 3900, respectivamente (si nos olvidamos de la tabla de contenido). Encontrar cualquier parte de cualquier archivo es fácil si se conoce el primer sector del archivo.

Para asignar espacio de disco a un archivo, el sistema operativo debe saber cuáles bloques están disponibles y en cuáles ya están almacenados otros archivos. En el caso de un CD-ROM, el cálculo se efectúa de una vez por todas antes de la grabación, pero en el caso de un disco duro los archivos entran y salen constantemente. Un método consiste en mantener una lista de todos los huecos, donde un hueco es cualquier cantidad de unidades de asignación contiguas. Esta lista se llama **lista libre**. En la figura 6-21(a) se ilustra la lista libre del disco de la figura 6-20(b).

Un método alternativo consiste en mantener un mapa de bits, con un bit por unidad de asignación, como se muestra en la figura 6-21(b). Un bit 1 indica que la unidad de asignación ya está ocupada, y un bit 0 indica que está disponible.

El primer método tiene la ventaja de que facilita la localización de un hueco de cierto tamaño, pero tiene la desventaja de que el tamaño de la lista es variable. A medida que se

Pista	Sector	Número de sectores en el hueco
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

Pista	Sector 0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	C	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	0	1	1	0	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Figura 6-21. Dos formas de llevar la contabilidad de los sectores disponibles. (a) Una lista libre. (b) Un mapa de bits.

crean y destruyen archivos, la longitud de la lista fluctúa, característica que es indeseable. El mapa de bits tiene la ventaja de tener un tamaño constante. Además, para modificar la situación de una unidad de asignación, de disponible a ocupada, sólo requiere modificar un bit. Por otra parte, no es fácil encontrar un bloque de un tamaño dado. Ambos métodos requieren que, cuando se asigna o borra cualquier archivo del disco, la lista o tabla de asignación se actualice.

Antes de abandonar el tema de la implementación de sistemas de archivos, vale la pena comentar algo acerca del tamaño de la unidad de asignación. En este sentido hay varios factores importantes. Primero, el tiempo de búsqueda y el retraso rotacional dominan los accesos a disco. Habiendo invertido 10 ms en llegar al principio de una unidad de asignación, es mucho mejor leer 8 KB (en aproximadamente 1 ms) que 1 KB (en aproximadamente 0.125 ms), ya que leer 8 KB como ocho unidades de 1 KB requeriría ocho búsquedas. La eficiencia de transferencia es un argumento a favor de tener unidades de asignación grandes.

Otro argumento a favor es que tener unidades de asignación pequeñas también implica tener muchas de ellas. Esto, a su vez, implica que los índices de archivos o las tablas de lista enlazada en la memoria serán grandes. De hecho, la razón por la que MS-DOS tuvo que cambiar a unidades de asignación de varios sectores fue el hecho de que las direcciones en disco se guardaban como números de 16 bits. Cuando los discos comenzaron a tener más de 64K sectores, la única forma de representarlos era usar unidades de asignación más grandes, para que el número de unidades de asignación no excediera 64K. La primera versión de Windows 95 tenía el mismo problema, pero una versión subsecuente usó números de 32 bits. Windows 98 reconoce ambos tamaños.

Por otra parte, un argumento en favor de tener unidades de asignación pequeñas es el hecho de que pocos archivos ocupan un número entero de unidades de asignación. Por tanto, se desperdiciará algo de espacio en la última unidad de asignación de casi todos los archivos. Si el archivo es mucho mayor que la unidad de asignación, el espacio desperdiciado medio será la mitad de una unidad de asignación. Cuanto mayor sea la unidad de asignación, mayor será la cantidad de espacio desperdiciado. Si en promedio los archivos son mucho más pequeños

que la unidad de asignación, se desperdiciará la mayor parte del espacio de disco. Por ejemplo, en una partición de disco de 2 GB de MS-DOS o Windows 95 versión 1, las unidades de asignación son de 32 KB, por lo que un archivo de 100 caracteres desperdicia 32,668 bytes de espacio de disco. La eficiencia de almacenamiento es un argumento en favor de usar unidades de asignación pequeñas. En la actualidad, la eficiencia de transferencia suele ser el factor más importante, por lo que hay una tendencia de los tamaños de bloque a aumentar con el tiempo.

6.2.3 Instrucciones para gestión de directorios

En los albores de la computación, la gente guardaba sus programas y datos en tarjetas perforadas en archiveros en sus oficinas. A medida que el tamaño y el número de los programas y datos creció, esta situación se volvió cada vez más indeseable y llevó finalmente a la idea de usar la memoria secundaria de la computadora (por ejemplo, los discos) como lugar de almacenamiento para los programas y datos. La información a la que la computadora puede acceder directamente sin necesidad de intervención humana se dice que está **en línea**, en contraste con la información **fuera de línea**, que requiere intervención humana (por ejemplo, insertar un CD-ROM) para que la computadora pueda accesar a ella.

La información en línea se guarda en archivos, y los programas pueden accesar a ella usando las instrucciones de E/S de archivos que acabamos de explicar. Sin embargo, se necesitan instrucciones adicionales para seguir la pista a toda la información almacenada en línea, reunirla en unidades convenientes y protegerla contra el uso no autorizado.

La forma en que un sistema operativo acostumbra organizar los archivos en línea es agrupándolos en **directorios**. En la figura 6-22 se muestra un ejemplo de organización de directorios. Se incluyen llamadas al sistema para efectuar al menos las siguientes funciones:

1. Crear un archivo e introducirlo en un directorio.
2. Borrar un archivo de un directorio.
3. Cambiar el nombre de un archivo.
4. Modificar la situación de protección de un archivo.

Se usan diversos esquemas de protección. Una posibilidad es que el dueño de cada archivo especifique una contraseña secreta para cada archivo. Al intentar acceder a un archivo, un programa debe proporcionar la contraseña, que el sistema operativo revisa para ver si es correcta antes de permitir el acceso. Otro método de protección es que el dueño de cada archivo proporcione una lista explícita de las personas cuyos programas pueden acceder a ese archivo.

Todos los sistemas operativos modernos permiten a los usuarios mantener más de un directorio de archivos. Por lo regular, cada directorio es en sí un archivo y, como tal, puede listarse en otro directorio, lo que da pie a un árbol de directorios. Tener varios directorios es útil sobre todo para los programadores que trabajan en varios proyectos. Así, pueden agrupar en un directorio todos los archivos relacionados con un proyecto. Mientras trabajan en ese proyecto, no se distraerán por la presencia de archivos que nada tienen que ver. Los directorios también son una forma cómoda de compartir archivos con los miembros de un grupo.

Archivo 0
Archivo 1
Archivo 2
Archivo 3
Archivo 4
Archivo 5
Archivo 6
Archivo 7
Archivo 8
Archivo 9
Archivo 10

Nombre:	Patio	
Longitud:	1840	
Tipo:	Anatidae dataram	
Fecha creado:	Marzo 16 de 1066	
Último acceso:	Septiembre 1 de 1492	
Último cambio:	Julio 4 de 1776	
Total de accesos:	144	
Bloque 0:	Pista 4	Sector 6
Bloque 1:	Pista 19	Sector 9
Bloque 2:	Pista 11	Sector 2
Bloque 3:	Pista 77	Sector 0

Figura 6-22. (a) Directorio de archivos de usuario. (b) Contenido de una entrada típica de un directorio de archivos.

6.3 INSTRUCCIONES VIRTUALES PARA PROCESAMIENTO EN PARALELO

Algunos cálculos se pueden programar de forma más conveniente para dos o más procesos cooperativos que se ejecutan en paralelo (es decir, simultáneamente en diferentes procesadores), que para un solo proceso. Otros cálculos se pueden dividir en fragmentos, que entonces se pueden ejecutar en paralelo a fin de reducir el tiempo requerido para el cálculo total. Para que varios procesos trabajen juntos de forma paralela se requieren ciertas instrucciones virtuales, que analizaremos en las secciones que siguen.

Las leyes de la física proporcionan una justificación más del interés actual en el procesamiento en paralelo. Según la teoría de la relatividad especial de Einstein, es imposible transmitir señales eléctricas a una velocidad mayor que la de la luz, que es de casi 1 pie/ns en el vacío, y menos en alambre de cobre o fibra óptica. Este límite tiene importantes implicaciones para la organización de las computadoras. Por ejemplo, si una CPU necesita datos de una memoria principal que está a 1 pie de distancia, la solicitud tardará al menos 1 ns en llegar a la memoria y la respuesta tardará otro nanosegundo en llegar a la CPU. Por consiguiente, las computadoras que quieran operar con tiempos de menos de un nanosegundo tendrán que ser muy pequeñas. Una estrategia alternativa para acelerar las computadoras es construir máquinas con muchas CPU. Una computadora con mil CPU de 1 ns podría tener la misma potencia de cómputo que una CPU con un tiempo de ciclo de 0.001 ns, pero la construcción de la primera podría ser mucho más fácil y económica.

En una computadora con más de una CPU, cada uno de varios procesos cooperativos puede asignarse a su propia CPU, para que los procesos puedan avanzar simultáneamente. Si sólo se cuenta con un procesador, el efecto del procesamiento en paralelo puede simularse haciendo que el procesador ejecute cada proceso por turno durante un tiempo corto. En otras palabras, varios procesos pueden compartir el procesador.

La figura 6-23 muestra la diferencia entre el verdadero procesamiento en paralelo, con más de un procesador físico, y el procesamiento paralelo simulado, con un solo procesador físico. Aun cuando se simula el procesamiento paralelo, es útil pensar que cada proceso tiene su propio procesador virtual dedicado. En el caso simulado surgen los mismos problemas de comunicación que se presentan cuando hay un verdadero procesamiento paralelo.

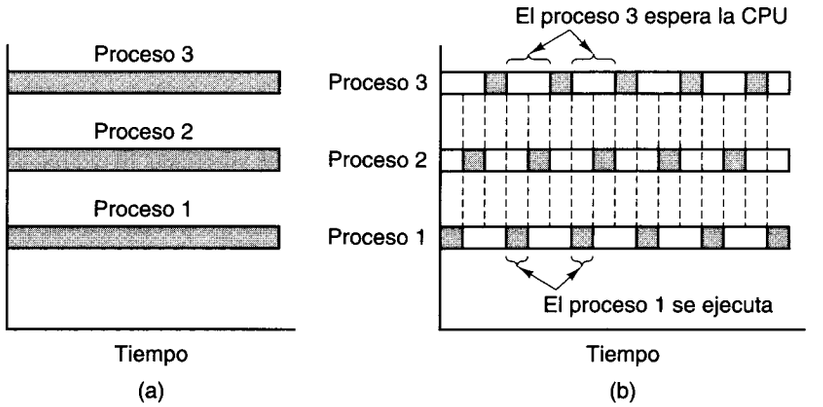


Figura 6-23. (a) Verdadero procesamiento paralelo con múltiples CPU. (b) Procesamiento paralelo simulado por conmutación de una CPU entre tres procesos.

6.3.1 Creación de procesos

Cuando se va a ejecutar un programa, debe operar como parte de algún proceso. Este proceso, como todos los demás, se caracteriza por un estado y un espacio de direcciones a través del cual se puede accesar al programa y los datos. El estado incluye como mínimo el contador de programa, una palabra de estado del programa, un apuntador de pila y los registros generales.

Casi todos los sistemas operativos modernos permiten crear y terminar procesos dinámicamente. Para aprovechar plenamente esta característica y lograr el procesamiento en paralelo, se requiere una llamada al sistema para crear un proceso nuevo. Esta llamada al sistema podría limitarse a crear un clon del proceso creador, o podría permitir al proceso creador especificar el estado inicial del nuevo proceso, incluido su programa, datos y dirección de inicio.

En algunos casos, el proceso creador (padre) mantiene un control parcial o incluso total del proceso creado (hijo). Con este fin, se incluyen instrucciones virtuales para que un padre pare, reinicie, examine y termine sus hijos. En otros casos, el padre tiene menos control sobre sus hijos. Una vez que se ha creado un proceso, el padre no tiene manera de detenerlo, reiniciarlo, examinarlo o terminarlo forzosamente. En tal caso, los procesos se ejecutan de forma independiente uno del otro.

6.3.2 Condiciones de competencia

En muchos casos los procesos paralelos necesitan comunicarse y sincronizarse para poder efectuar su trabajo. En esta sección examinaremos la sincronización de procesos y explicaremos algunas de las dificultades con la ayuda de un ejemplo detallado. En la sección que sigue se presentará una solución a estas dificultades.

Considere una situación que consiste en dos procesos independientes, el 1 y el 2, que se comunican por medio de un buffer compartido en la memoria principal. Para simplificar, llamaremos al proceso 1 el **productor** y al proceso 2 el **consumidor**. El productor calcula números primos y los coloca en el buffer, uno por uno. El consumidor los saca del buffer uno por uno y los imprime.

Estos dos procesos se ejecutan en paralelo con diferente velocidad. Si el productor descubre que el buffer está lleno, se "duerme"; es decir, suspende temporalmente su operación en espera de una señal del consumidor. Más adelante, una vez que el consumidor ha sacado un número del buffer, envía una señal al productor para despertarlo; es decir, reiniciarlo. Así mismo, si el consumidor descubre que el buffer está vacío, se duerme. Una vez que el productor ha colocado un número en el buffer vacío, despierta al consumidor dormido.

En este ejemplo usaremos un buffer circular para la comunicación entre procesos. Los apuntadores *in* y *out* se usarán como sigue: *in* apunta a la siguiente palabra desocupada (donde el productor pondrá el siguiente número primo) y *out* apunta al siguiente número que el consumidor quitará. Si *in* = *out*, el buffer está vacío, como se muestra en la figura 6-24(a). Una vez que el productor ha generado algunos primos, la situación es la que se muestra en la figura 6-24(b). La figura 6-24(c) ilustra el buffer después de que el consumidor ha sacado algunos de estos números primos para imprimirlos. La figura 6-24(d)-(f) muestra el efecto de una actividad de buffer constante. El tope del buffer es lógicamente contiguo a su base; es decir, el buffer da la vuelta. Si hubo una ráfaga repentina de entradas e *in* dio la vuelta hasta estar sólo una palabra detrás de *out* (por ejemplo, *in* = 52 y *out* = 53), el buffer está lleno. La última palabra no se usa; si se usara, no habría forma de saber si *in* = *out* significa que el buffer está lleno o que está vacío.

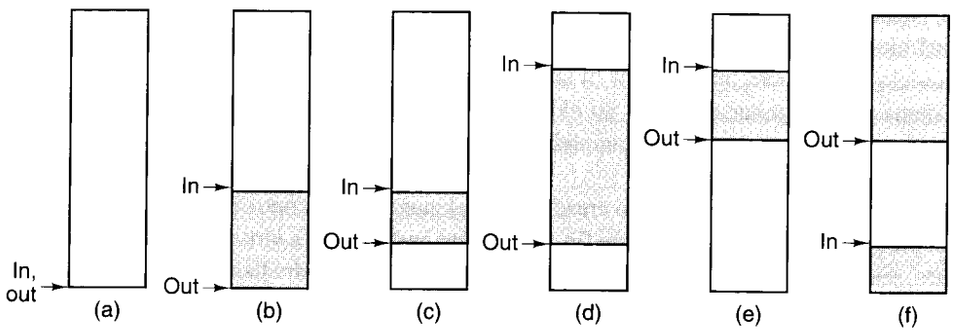


Figura 6-24. Uso de un buffer circular.

La figura 6-25 muestra una forma sencilla de implementar el problema del productor y el consumidor en Java. Esta solución usa tres clases, *m*, *productor* y *consumidor*. La clase *m* ("main"), contiene algunas definiciones de constantes, los apuntadores *in* y *out*, y el buffer mismo, que en este ejemplo contiene 100 números primos: *buffer*[0] a *buffer*[99].

Esta solución usa enlaces de Java para simular procesos paralelos. Con esta solución tenemos una clase *productor* y una clase *consumidor*, que se ejemplifican en las variables *p* y *c*, respectivamente. Ambas clases se derivan de una clase base *Thread* (enlace), que tiene un método *run* (ejecutar). La clase *run* contiene el código del enlace. Cuando se invoca el método *start* de un objeto derivado de *Thread*, se inicia un nuevo enlace.

Cada enlace es como un proceso, excepto que todos los enlaces dentro de un solo programa Java se ejecutan en el mismo espacio de direcciones. Esta característica es conveniente para que todos compartan un buffer común. Si la computadora tiene dos o más CPU, cada enlace se puede planificar en una CPU distinta, y se logra un verdadero paralelismo. Si sólo hay una CPU, los enlaces se ejecutan por tiempo compartido en ella. Seguiremos refiriéndonos al productor y al consumidor como procesos (ya que lo que realmente nos interesa son los procesos paralelos), aunque Java sólo reconozca enlaces paralelos y no procesos verdaderamente paralelos.

La función *siguiente* permite incrementar fácilmente a *in* y *out*, sin tener que escribir código que verifique la condición de continuidad del tope al fondo en cada ocasión. Si el parámetro de *siguiente* es 98 o menos, se devuelve el siguiente entero mayor, pero si el parámetro es 99 quiere decir que llegamos al final del buffer, y se devuelve 0.

Necesitamos un mecanismo que permita a cualquiera de los procesos dormirse cuando no pueda continuar. Los diseñadores de Java entendieron la necesidad de una capacidad así e incluyeron los métodos *suspend* (suspender, para dormirse) y *resume* (reanudar, para despertarse) en la clase *Thread* desde la primera versión de Java. Estos métodos se usan en la figura 6-25.

Ahora llegamos al código para el productor y el consumidor. Primero, el productor genera un primo nuevo en P1. Observe el uso de *m.MAX_PRIMO* aquí. El prefijo *m.* es necesario para indicar que estamos hablando del *MAX_PRIMO* definido en la clase *m*. Por la misma razón necesitamos anteponer este prefijo también a *in*, *out*, *buffer* y *siguiente*.

Luego el productor verifica (en P2) si *in* está una posición atrás de *out*. Si así es (por ejemplo, si *in* = 62 y *out* = 63), quiere decir que el buffer está lleno y el productor se duerme llamando a *suspend* en P2. Si el buffer no está lleno, el nuevo primo se inserta en el buffer (P3) e *in* se incrementa (P4). Si el nuevo valor de *in* está 1 adelante de *out* (P5) (por ejemplo, si *in* = 17 y *out* = 16), quiere decir que *in* y *out* deben haber sido iguales antes de que *in* se incrementara. El productor concluye que el buffer estaba vacío y que el consumidor estaba, y todavía está, durmiendo. Por tanto, el productor invoca *resume* para despertar al consumidor (P5). Por último, el productor comienza a buscar el siguiente número primo.

Estructuralmente, el programa del consumidor es similar. Primero se efectúa una prueba (C1) para ver si el buffer está vacío. Si así es, el consumidor no tiene nada que hacer, por lo que se duerme. Si el buffer no está vacío, el consumidor saca el siguiente número que se imprimirá (C2) e incrementa *out* (C3). Si *out* está dos posiciones adelante de *in* en este momento (C4), debió haber estado una posición adelante de *in* antes de que éste se incrementara.

```
public class m {
    final public static int TAM_BUFFER = 100;                    // buffer va de 0 a 99
    final public static long MAX_PRIMO = 100000000000L;          // parar aquí
    public static int in = 0, out = 0;                           // apuntadores a los datos
    public static long buffer[ ] = new long[TAM_BUFFER];         // los primos se guardan aquí
    public static productor p;                                   // nombre del productor
    public static consumidor c;                                  // nombre del consumidor

    public static void main(String args[ ]) {                    // clase principal
        p = new productor( );                                    // crear el productor
        c = new consumidor( );                                   // crear el consumidor
        p.start( );                                              // iniciar el productor
        c.start( );                                              // iniciar el consumidor
    }

    // Función utilitaria para incrementar circularmente in y out
    public static int siguiente(int k) {if (k < TAM_BUFFER – 1) return (k + 1); else return(0);}
}

clas productor extends Thread {                                  // clase de productor
    public void run( ) {                                         // código de productor
        long primo = 2;                                          // variable de borrador

        while (primo < m.MAX_PRIMO) {
            primo = sig_primo(primo);                            // enunciado P1
            if (m.siguiente(m.in) == m.out) suspend();           // enunciado P2
            m.buffer[m.in] = primo;                              // enunciado P3
            m.in = m.siguiente(m.in);                            // enunciado P4
            if (m.siguiente(m.out) == m.in) m.c.resume();        // enunciado P5
        }
    }

    private long sig_primo(long primo){...}                      // función que calcula
}                                                                // el siguiente primo

class consumidor extends Thread {                                // clase consumidor
    public void run( ) {                                         // código del consumidor
        long omirp = 2;                                          // variable de borrador

        while (omirp < m.MAX_PRIMO){
            if (m.in == m.out) suspend( );                       // enunciado C1
            omirp = m.buffer[m.out];                             // enunciado C2
            m.out = m.siguiente(m.out);                          // enunciado C3
            if (m.out == m.siguiente(m.siguiente(m.in)))
            m.p.resume( );                                       // enunciado C4
            System.out.println(omirp);                           // enunciado C5
        }
    }
}
```

Figura 6-25. Procesamiento en paralelo con condición de competencia fatal.

Puesto que *in* = *out* – 1 es la condición "buffer lleno", el productor debe haber estado durmiendo, así que el consumidor lo despertará con *resume*. Por último, se imprime el número (C5) y el ciclo se repite.

Lamentablemente, este diseño contiene un defecto fatal, como se muestra en la figura 6-26. Recuerde que los dos procesos se ejecutan de forma asincrónica y a diferentes velocidades, que podrían ser variables. Considere el caso en que sólo queda un número en el buffer, en la entrada 21, e *in* = 22 y *out* = 21, como en la figura 6-26(a). El productor está en el enunciado P1 buscando un número primo y el consumidor está ocupado en C5 imprimiendo el número que está en la posición 20. El consumidor termina de imprimir el número, realiza la prueba en C1, y saca el último número del buffer en C2; luego incrementa *out*. En este momento, tanto *in* como *out* tienen el valor 22. El consumidor imprime el número y pasa a C1, donde trae a *in* y *out* de la memoria para compararlos, como se muestra en la figura 6-26(b).

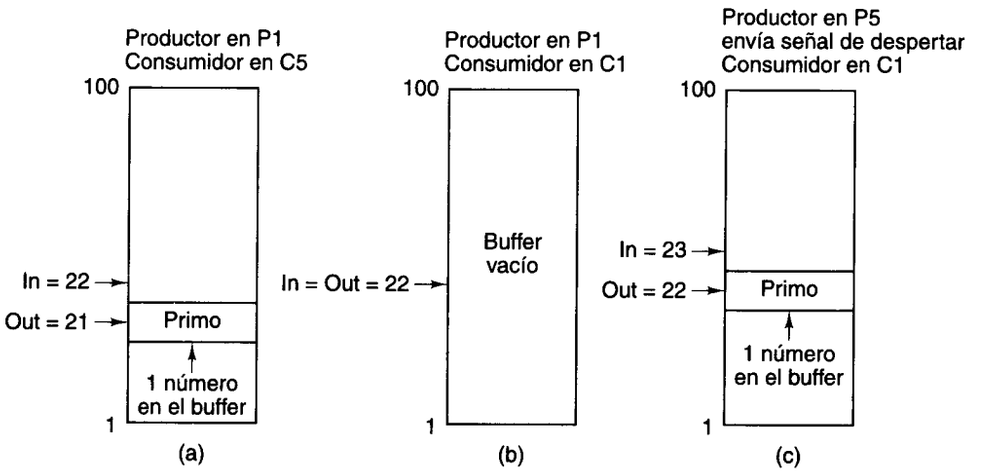


Figura 6-26. Fracaso del mecanismo de comunicación productor-consumidor.

En este momento preciso, después de que el consumidor ha traído a *in* y *out* pero antes de haberlos comparado, el productor encuentra el siguiente primo, lo coloca en el buffer P3 e incrementa *in* en P4. Ahora *in* = 23 y *out* = 22. En P5 el productor descubre que *in* = *siguiente*(*out*). En otras palabras, *in* es 1 más que *out*, lo que indica que hay un elemento en el buffer. Por ello, el productor concluye (incorrectamente) que el consumidor está durmiendo, así que le envía una señal de despertar (es decir, invoca *resume*), como se muestra en la figura 6-26(c). Claro que el consumidor todavía está despierto, por lo que la señal de despertar se pierde. El productor comienza a buscar el siguiente número primo.

En este momento el consumidor continúa. El consumidor ya había traído *in* y *out* de la memoria antes de que el productor colocara el último número en el buffer. Como ahora tanto

in como *out* tienen el valor 22, el consumidor se duerme. Ahora el productor encuentra otro número primo, examina los apuntadores y encuentra que *in* = 24 y *out* = 22, por lo que supone que hay dos números en el buffer (lo cual es cierto) y que el consumidor está despierto (lo cual es falso). El productor sigue repitiendo el ciclo. En algún momento, el buffer se llena y el productor se duerme. Ahora ambos procesos están durmiendo y seguirán haciéndolo indefinidamente.

El problema aquí es que entre el momento en que el consumidor trajo a *in* y *out* y el momento en que se durmió, el productor llegó, descubrió que *in* = *out* + 1, supuso que el consumidor estaba dormido (aunque todavía no lo estaba), y envió una señal de despertar que se perdió porque el consumidor todavía estaba despierto. Este problema se denomina **condición de competencia** porque el éxito del método depende de quién gana la competencia por probar *in* y *out* después de que *out* se incrementa.

El problema de las condiciones de competencia se conoce bien. De hecho, es tan grave que varios años después de la introducción de Java Sun modificó la clase *Thread* y recomendó no usar las llamadas *suspend* y *resume* porque daban pie a condiciones de competencia muy a menudo. La solución que se ofreció fue una basada en el lenguaje, pero como aquí estamos estudiando sistemas operativos analizaremos una solución distinta, que muchos sistemas operativos manejan, entre ellos UNIX y NT.

6.3.3 Sincronización de procesos con semáforos

La condición de competencia se puede resolver de por lo menos dos maneras. Una solución consiste en equipar cada proceso con un "bit de despertar pendiente". Siempre que se envía una señal de despertar a un proceso que todavía está en ejecución, se enciende su bit de despertar pendiente. Si un proceso se duerme y el bit de despertar pendiente está encendido, de inmediato se le reinicia y se apaga el bit de despertar pendiente. De hecho, lo que hace este bit es guardar la señal de despertar superflua para usarla en el futuro.

Aunque este método resuelve la condición de competencia cuando sólo hay dos procesos, falla en el caso general de n procesos en comunicación porque podría ser necesario guardar hasta $n - 1$ señales de despertar. Desde luego, se podría equipar a cada proceso con $n - 1$ bits de despertar pendiente para que pudiera contar hasta $n - 1$ en el sistema unario, pero se trata de una solución un tanto torpe.

Dijkstra (1968b) propuso una solución más general al problema de sincronizar procesos paralelos. En algún lugar de la memoria hay dos variables enteras no negativas llamadas **semáforos**. El sistema operativo proporciona dos llamadas al sistema que operan con semáforos, up y down. Up suma 1 a un semáforo y down resta 1 a un semáforo.

Si se realiza una operación down con un semáforo cuyo valor es mayor que 0, el semáforo se decrementa en 1 y el proceso que llamó a down continúa. En cambio, si el semáforo es 0, down no puede llevarse a cabo; el proceso que lo llamó se duerme y permanece dormido hasta que el otro proceso realiza up con ese semáforo.

La instrucción up verifica si el semáforo es 0. Si así es y el otro proceso está durmiendo por él, el semáforo se incrementa en 1. Entonces, el proceso dormido puede finalizar la operación down que lo suspendió, el semáforo se vuelve a poner en 0 y los dos procesos pueden

continuar. Una instrucción up con un semáforo distinto de cero simplemente lo incrementa en 1. En esencia, un semáforo es un contador para guardar señales de despertar que se usarán posteriormente, a fin de que no se pierdan. Una propiedad indispensable de las instrucciones de semáforo es que, una vez que un proceso ha iniciado una instrucción con un semáforo, ningún otro proceso podrá acceder al semáforo hasta que el primero haya finalizado la instrucción o bien se haya suspendido tratando de ejecutar down con un 0. La figura 6-27 resume las propiedades fundamentales de las llamadas al sistema up y down.

Instrucción	Semáforo = 0	Semáforo > 0
Up	Semáforo = semáforo + 1 si el otro proceso se detuvo tratando de llevar a cabo una instrucción down con este semáforo, ahora puede finalizar la down y continuar su ejecución	Semáforo = semáforo + 1;
Down	El proceso se detiene hasta que el otro proceso hace up con este semáforo	Semáforo = semáforo – 1

Figura 6-27. Efecto de una operación de semáforo.

Como se dijo antes, Java tiene una solución basada en el lenguaje para manejar condiciones de competencia, y ahora estamos hablando de sistemas operativos. Por tanto, necesitamos una forma de expresar el uso de semáforos en Java, aunque no forme parte del lenguaje ni de las clases estándar. Lo haremos suponiendo que se escribieron dos métodos nativos, *up* y *down*, que efectúan las llamadas al sistema up y down, respectivamente. Si invocamos estos métodos con enteros ordinarios como parámetros, podremos expresar el uso de semáforos en programas escritos en Java.

La figura 6-28 muestra cómo puede eliminarse la condición de competencia mediante el uso de semáforos. Se añaden dos semáforos a la clase *m*, *disponible*, que inicialmente es 100 (el tamaño del buffer), y *lleno*, que inicialmente es 0. El productor comienza a ejecutarse en P1 en la figura 6-28, y el consumidor comienza a ejecutarse en C1 igual que antes. La llamada *down* con *lleno* detiene el proceso consumidor de inmediato. Cuando el productor encuentra el primer número primo, invoca a *down* con *disponible* como parámetro, lo que decrementa *disponible* a 99. En P5, el productor invoca a *up* con *lleno* como parámetro, lo que incrementa *lleno* a 1. Esta acción libera al consumidor, que ahora puede finalizar su llamada *down* suspendida. En este momento, *lleno* es 0 y ambos procesos están en ejecución.

Volvamos a examinar ahora la condición de competencia. En cierto momento, *in* = 22, *out* = 21, el productor está en P1 y el consumidor está en C5. El consumidor termina lo que estaba haciendo y pasa a C1 donde invoca *down* con *filled*, que tenía el valor 1 antes de la llamada y 0 después. Luego, el consumidor saca el último número del buffer e invoca *up* con *disponible*, que lo incrementa a 100. El consumidor imprime el número y pasa a C1. Justo antes de que el consumidor pueda invocar a *down*, el productor encuentra el siguiente primo y en una secuencia rápida ejecuta los enunciados P2, P3 y P4.

En este punto, *lleno* es 0. El productor está a punto de aplicarle up y el consumidor está a punto de invocar a *up*. Si el consumidor ejecuta su instrucción primero, se suspenderá hasta que el productor lo libere (llamando a *up*). Por otra parte, si el productor actúa primero, el semáforo se pondrá en 1 y el consumidor no se suspenderá. En ninguno de los casos hay pérdida de una señal de despertar. Claro que esto era lo que buscábamos al introducir los semáforos.

La propiedad fundamental de las operaciones con semáforos es que son indivisibles. Una vez que se ha iniciado una operación con un semáforo, ningún otro proceso puede usar el semáforo hasta que el primer proceso haya finalizado la operación o bien haya quedado suspendido en el intento. Además, con los semáforos no se pierden señales de despertar. En contraste, los enunciados if de la figura 6-25 son indivisibles. Entre la evaluación de la condición y la ejecución del enunciado seleccionado otro proceso podría enviar una señal de despertar.

El problema de la sincronización de procesos se ha eliminado efectivamente declarando que las llamadas al sistema up y down que efectúan *up* y *down* son indivisibles. Para que estas operaciones sean indivisibles, el sistema operativo debe prohibir que dos o más procesos usen el mismo semáforo al mismo tiempo. Como mínimo, una vez efectuada una llamada al sistema up o down, no se ejecutará ningún otro código de usuario hasta que la llamada se haya llevado a cabo. Es común implementar los semáforos inhabilitando las interrupciones durante las operaciones con semáforos.

La sincronización con semáforos es una técnica que funciona con un número arbitrario de procesos. Varios procesos pueden estar dormidos tratando de finalizar una llamada al sistema down con el mismo semáforo. Cuando algún otro proceso por fin ejecuta up con ese semáforo, se permite que uno de los procesos en espera finalice su down y continúe su ejecución. El valor del semáforo seguirá siendo 0 y los demás procesos seguirán esperando.

Quizá podamos aclarar la naturaleza de los semáforos con una analogía. Imagine un día de campo con 20 equipos de volibol divididos en 10 juegos (procesos) cada uno de los cuales juega en su propia cancha, y una canasta grande (el semáforo) para los balones de volibol. Lo malo es que sólo hay siete balones. En un instante dado, hay entre cero y siete balones en la canasta (el semáforo tiene un valor entre 0 y 7). Colocar un balón en la canasta es un up porque incrementa el valor del semáforo. Sacar un balón de la canasta es un down porque decrementa el valor del semáforo.

Al principio del día de campo, cada cancha envía un jugador a la canasta para conseguir un balón. Siete de ellos logran conseguir un balón (llevan a cabo el down); tres se ven obligados a esperar un balón (es decir, no logran finalizar el down). Sus juegos se suspenden temporalmente. Tarde o temprano, uno de los otros juegos termina y coloca su balón en la canasta (ejecuta up). Esta operación permite a uno de los tres jugadores que esperan junto a la canasta obtener un balón (finalizar un down inconcluso), lo que permite a un juego continuar. Los otros dos juegos siguen suspendidos hasta que se coloquen otros dos balones en la canasta. Cuando se devuelven dos balones (se ejecutan otros dos up), los dos últimos juegos pueden continuar.


```
public class m {
    final public static int TAM_BUFFER = 100;                    // buffer va de 0 a 99
    final public static long MAX_PRIMO = 100000000000L;          // parar aquí
    public static int in = 0, out = 0;                           // apuntadores a los datos
    public static long buffer[ ] = new long[TAM_BUFFER];         // los primos se guardan aquí
    public static productor p;                                   // nombre del productor
    public static consumidor c;                                  // nombre del consumidor
    public static int lleno = 0, disponible = 100;               // semáforos

    public static void main(String args[ ]) {                    // clase principal
        p = new productor( );                                    // crear el productor
        c = new consumidor( );                                   // crear el consumidor
        p.start( );                                              // iniciar el productor
        c.start( );                                              // iniciar el consumidor
    }
    // Función utilitaria para incrementar circularmente in y out
    public static int siguiente(int k) {if (k = TAM_BUFFER – 1) return (k + 1); else return(0);}
}

class productor extends Thread {                                 // clase productor
    native void up(int s); native void down(int s);              // métodos para semáforos
    public void run( ) {                                         // código de productor
        long primo = 2;                                          // variable de borrador

        while (primo < m.MAX_PRIMO) {
            primo = sig_primo(primo);                            // enunciado P1
            down(m.disponible);                                  // enunciado P2
            m.buffer[m.in] = primo;                              // enunciado P3
            m.in = m.siguiente(m.in);                            // enunciado P4
            up(m.lleno);                                         // enunciado P5
        }
    }

    private long sig_primo(long primo){...}                      // función que calcula
}                                                                // el siguiente primo

class consumidor extends Thread {                                // clase consumidor
    native void up(int s); native void down(int s);              // métodos para semáforos
    public void run( ) {                                         // código del consumidor
        long omirp = 1;                                          // variable de borrador

        while (omirp < m.MAX_PRIMO){
            down(m.lleno);                                       // enunciado C1
            omirp = m.buffer[m.out];                             // enunciado C2
            m.out = m.siguiente(m.out);                          // enunciado C3
            up(m.disponible);                                    // enunciado C4
            System.out.println(omirp);                           // enunciado C5
        }
    }
}
```

Figura 6-28. Procesamiento en paralelo con semáforos.

6.4 EJEMPLOS DE SISTEMAS OPERATIVOS

En esta sección seguiremos examinando nuestros sistemas de ejemplo, el Pentium II y el UltraSPARC II. En cada caso veremos un sistema operativo empleado en ese procesador. Para el Pentium II usaremos Windows NT (que abreviaremos como NT en lo que sigue); para el UltraSPARC II usaremos UNIX. Puesto que UNIX es más sencillo y en muchos aspectos más elegante, comenzaremos con él. Además, UNIX se diseñó e implementó primero y tuvo una importante influencia sobre NT, por lo que este orden es más lógico que el opuesto.

6.4.1 Introducción

En esta sección presentaremos una breve introducción a nuestros dos sistemas operativos de ejemplo, UNIX y NT, concentrándonos en su historia, estructura y llamadas al sistema.

UNIX

UNIX se creó en Bell Labs a principios de los años setenta. La primera versión fue escrita por Ken Thompson en ensamblador para la minicomputadora PDP-7. Pronto apareció una versión para la PDP-11, escrita en un nuevo lenguaje llamado C ideado e implementado por Dennis Ritchie. En 1974, Ritchie y Thompson publicaron un artículo fundamental sobre UNIX (Ritchie y Thompson, 1974). Por el trabajo descrito en ese artículo ellos recibieron posteriormente el prestigioso Premio Turing de la ACM (Ritchie, 1984; Thompson, 1984). La publicación de este artículo hizo que muchas universidades solicitaran a Bell Labs una copia de UNIX. Puesto que la compañía fundadora de Bell Labs, AT&T, era un monopolio regulado en ese entonces y no tenía permiso para entrar en el negocio de las computadoras, no tuvo reparo en otorgar licencias de UNIX a las universidades por una cuota modesta.

En una de esas coincidencias que a menudo moldean la historia, la PDP-11 era la computadora preferida por casi todos los departamentos de ciencias de la computación universitarios, y una opinión muy difundida tanto entre los profesores como los estudiantes era que los sistemas operativos que venían con la PDP-11 eran horribles. UNIX pronto llenó el vacío, en buena medida porque venía acompañado por el código fuente completo, lo que permitía a los profesores y estudiantes hacerle todas las adiciones y modificaciones que se les ocurrían.

Una de las muchas universidades que adquirieron UNIX desde el principio fue la University of California en Berkeley. Puesto que contaba con el código fuente completo, Berkeley pudo modificar el sistema considerablemente. Los cambios más notables fueron un puerto para la minicomputadora VAX y la adición de memoria virtual paginada, la extensión de los nombres de archivo de 14 a 255 caracteres, y la inclusión del protocolo de redes TCP/IP, que ahora se usa en Internet (en gran medida por el hecho de que estaba en Berkeley UNIX).

Mientras Berkeley estaba efectuando todos estos cambios, AT&T siguió desarrollando UNIX, y sacó el System III en 1982 y el System V en 1984. Para fines de los años ochenta se usaban ampliamente dos versiones diferentes e incompatibles de UNIX: Berkeley UNIX y System V. Esta situación en el mundo UNIX, junto con el hecho de que no había estándares para los

formatos de programas binarios, inhibieron considerablemente el éxito comercial de UNIX porque era imposible para los proveedores de software escribir y empaquetar programas UNIX con la expectativa de que funcionarían en cualquier sistema UNIX (como se hacía todo el tiempo con MS-DOS). Después de muchas peleas y discusiones, la IEEE Standards Board creó un estándar llamado **POSIX** (**Portable Operating System IX**). POSIX también se conoce por su número de norma IEEE, P1003, y posteriormente se convirtió en un estándar internacional.

El estándar POSIX se divide en muchas partes, cada una de las cuales cubre un área diferente de UNIX. La primera parte, P1003.1, define las llamadas al sistema; la segunda, P1003.2, define los programas utilitarios básicos, etc. El estándar P1003.1 define unas 60 llamadas al sistema que todos los sistemas que cumplan con la norma deben apoyar. Éstas son las llamadas básicas para leer y escribir archivos, crear procesos nuevos, y demás. Casi todos los sistemas UNIX actuales apoyan las llamadas al sistema P1003.1, pero muchos reconocen también llamadas adicionales, sobre todo las definidas por System V y/o Berkeley UNIX. En general, esto representa unas 100 llamadas al sistema además del conjunto POSIX. El sistema operativo para el UltraSPARC II se basa en System V y se llama **Solaris**. Éste también reconoce muchas de las llamadas al sistema de Berkeley.

En la figura 6-29 se presenta una clasificación aproximada de las llamadas al sistema de Solaris. Las llamadas para gestión de archivos y directorios son las categorías más grandes e importantes. Casi todas provienen de P1003.1, y una fracción relativamente grande de las otras se deriva de System V.

Categoría	Ejemplos
Gestión de archivos	Abrir, leer, escribir, cerrar y poner candados a archivos
Gestión de directorios	Crear y eliminar directorios; trasladar archivos
Gestión de procesos	Engendrar, terminar, rastrear y señalizar procesos
Gestión de memoria	Compartir memoria entre procesos; proteger páginas
Obtener/fijar parámetros	Obtener ID de usuario, grupo o proceso; fijar prioridad
Fechas y horas	Fijar tiempos de acceso a archivos; usar temporizador de intervalor, ejecutar perfiles
Trabajo con redes	Establecer/aceptar conexión; enviar/recibir mensaje
Diversas	Habilitar contabilización; manipular cuotas de disco; reiniciar el sistema

Figura 6-29. Clasificación a grandes rasgos de las llamadas al sistema de UNIX.

Un área que se debe en su mayor parte a Berkeley UNIX y no a System V es la de trabajo con redes. Berkeley inventó el concepto de **socket**, que es el punto terminal de una conexión de red. Los contactos de pared con cuatro terminales a los que pueden conectarse los teléfonos sirvieron como modelo para este concepto. Un proceso UNIX puede crear un *socket*, vincularse a él y establecer una conexión con un *socket* de una máquina distante. Luego el proceso puede intercambiar datos en ambas direcciones por esta conexión, casi siempre empleando el protocolo TCP/IP. Puesto que la tecnología de trabajo con redes ha estado en UNIX desde hace

décadas y es muy estable y madura, una fracción considerable de los servidores de Internet ejecutan UNIX.

Dado que hay muchas implementaciones de UNIX, es difícil decir mucho acerca de la estructura del sistema operativo, ya que cada una presenta varias diferencias respecto a todas las otras. No obstante, en general, la figura 6-30 se aplica a casi todas ellas. En la parte baja hay una capa de *drivers* de dispositivos que aíslan al sistema de archivos del hardware desnudo. Originalmente, cada *driver* de dispositivo se escribió como entidad independiente, distinta de todas las demás. Este sistema dio pie a duplicación de trabajo, ya que muchos *drivers* deben ocuparse del control de flujo, manejo de errores, prioridades, separación entre datos y control, etc. Esta observación llevó a Dennis Ritchie a crear un esquema llamado **streams** para escribir *drivers* de forma modular. Con un *stream*, es posible establecer una conexión bidireccional entre un proceso usuario y un dispositivo de hardware e insertar uno o más módulos en el camino. El proceso de usuario mete datos en el *stream*, que luego es procesado o transformado por cada módulo hasta llegar al hardware. Los datos entrantes experimentan el procesamiento inverso.

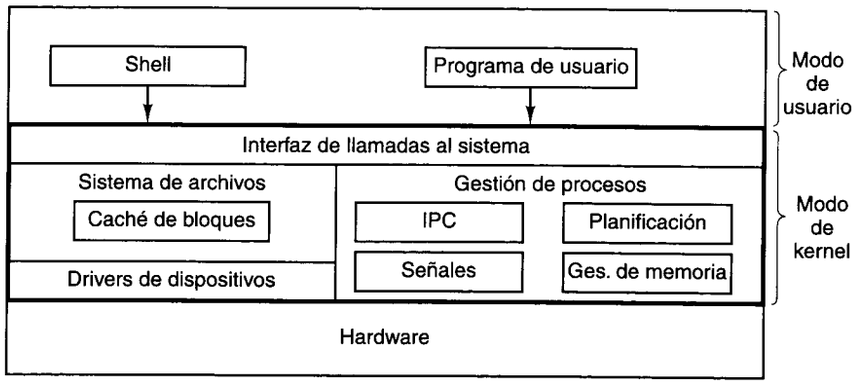


Figura 6-30. Estructura de un sistema UNIX representativo.

Encima de los *drivers* de dispositivos está el sistema de archivos, que administra nombres de archivo, directorios, asignación de bloques en disco, protección y mucho más. Parte del sistema de archivos es una **caché de bloques** en el que se retienen los bloques que se leyeron del disco más recientemente, por si se vuelven a necesitar pronto. Se han utilizado diversos sistemas de archivos con el paso de los años, incluido el sistema de archivos rápido Berkeley (McKusick *et al.*, 1984) y sistemas de archivos con estructura de bitácora (Rosenblum y Ousterhout, 1991; Seltzer *et al.*, 1993).

La otra parte del kernel de UNIX es la porción de gestión de procesos. Entre sus diversas funciones, esta parte maneja la comunicación entre procesos (IPC, *InterProcess Communication*), que permite a los procesos comunicarse entre sí y sincronizarse para evitar condiciones de competencia. Se proporcionan diversos mecanismos. El código de gestión de procesos

también se encarga de la planificación de procesos, que se basa en prioridades. Las señales, que son una forma de interrupción (asincrónica) por software, también se gestionan aquí. Por último, la gestión de memoria también se efectúa en esta porción. Casi todos los sistemas UNIX apoyan una memoria virtual paginada por demanda, a veces con unas cuantas funciones extra, como la posibilidad de que varios procesos compartan regiones de espacio de direcciones comunes.

Desde su nacimiento, UNIX ha tratado de ser un sistema pequeño, a fin de mejorar la confiabilidad y el desempeño. Las primeras versiones de UNIX se basaban exclusivamente en texto, empleando terminales capaces de exhibir 24 o 25 líneas de 80 caracteres ASCII. La interfaz con el usuario era manejada por un programa de nivel de usuario llamado **shell**, que ofrecía una interfaz de línea de comandos. Puesto que el *shell* no formaba parte del kernel, era fácil añadir nuevos *shells*, y con el tiempo se inventaron varios, cada uno más sofisticado que el anterior.

Más adelante, cuando aparecieron las terminales gráficas, se creó en MIT, un sistema de ventanas para UNIX llamado **X Windows**. Tiempo después, se colocó encima de Windows X una **GUI (interfaz gráfica con el usuario**, *Graphical User Interface*) más robusta, llamada **Motif**. En congruencia con la filosofía UNIX de tener un kernel pequeño, casi todo el código de X Windows y Motif se ejecuta en modo de usuario, fuera del kernel.

Windows NT

Cuando la IBM PC original salió al mercado en 1981, estaba equipada con un sistema operativo de 16 bits, en modo real, monousuario, orientado a línea de comandos, llamado MS-DOS 1.0. Este sistema operativo consistía en 8 KB de código residente en la memoria. Dos años después apareció un sistema mucho más potente de 24 KB, MS-DOS 2.0, que contenía un procesador de línea de comandos (*shell*) con varias funciones copiadas de UNIX. Cuando IBM sacó la PC/AT basada en el 286 en 1984, venía equipada con MS-DOS 3.0, que para entonces ocupaba 36 KB. Con los años, MS-DOS siguió adquiriendo funciones nuevas, pero seguía siendo un sistema orientado a línea de comandos.

Inspirado por el éxito de la Apple Macintosh, Microsoft decidió añadir a MS-DOS una interfaz gráfica con el usuario a la que llamó **Windows**. Las primeras tres versiones de Windows, que culminaron con Windows 3.x, no eran verdaderos sistemas operativos, sino interfaces gráficas con el usuario montadas encima de MS-DOS, que seguía teniendo el control de la máquina. Todos los programas se ejecutaban en el mismo espacio de direcciones, y un error en cualquiera de ellos podía parar en seco a toda la máquina.

Con el lanzamiento de Windows 95 en 1995 no llegó a eliminarse a MS-DOS, aunque introdujo una nueva versión, 7.0. Juntos, Windows 95 y MS-DOS 7.0 contenían casi todas las características de un sistema operativo más robusto, incluidas memoria virtual, gestión de procesos y multiprogramación. Sin embargo, Windows 95 no era un programa cabal de 32 bits; contenía grandes fragmentos de código viejo de 16 bits (además de un poco de código de 32 bits) y seguía usando el sistema de archivos de MS-DOS, con casi todas sus limitaciones. El único cambio importante al sistema de archivos fue la adición de nombres de archivo largos en lugar de los nombres de 8 + 3 caracteres permitidos en MS-DOS.

Incluso con la llegada de Windows 98 en 1998, MS-DOS seguía ahí (ahora en su versión 7.1) y seguía ejecutando código de 16 bits. Aunque un poco más de funcionalidad migró de la parte MS-DOS a la parte Windows, y se adoptó como norma una organización de disco apropiada para discos mayores, en su corazón Windows 98 no era muy diferente de Windows 95. La principal diferencia fue la interfaz con el usuario, que integró el escritorio, Internet y la televisión de forma más estrecha. Fue precisamente esta integración la que atrajo la atención del Departamento de Justicia de Estados Unidos, que entonces demandó a Microsoft alegando que era un monopolio ilegal.

Mientras ocurrían todas estas cosas, Microsoft también estaba enfrascado en la creación de un sistema operativo de 32 bits totalmente nuevo que se estaba escribiendo desde cero. Este nuevo sistema se llamó **Windows New Technology**, o **Windows NT**. En un principio, NT se anunció como el reemplazo de todos los demás sistemas operativos para PC basadas en Intel, pero su adopción fue un tanto lenta y posteriormente se reorientó hacia el extremo superior del mercado, donde encontró un nicho. Ahora también se está popularizando en el extremo inferior.

NT se vende en dos versiones: servidor y estación de trabajo. Estas dos versiones son casi idénticas y se generan a partir del mismo código fuente. La versión para servidor se diseñó para máquinas que operan como servidores de archivos e impresoras basadas en LAN y tiene funciones de administración más completas que la versión para estación de trabajo, que está dirigida a la computación de escritorio monousuario. La versión para servidor tiene una variante (empresa) pensada para sitios grandes. Las diversas versiones se afinan de forma distinta, y cada una se optima para su entorno esperado. Fuera de estas diferencias menores, todas las versiones son prácticamente iguales. De hecho, casi todos los archivos ejecutables son idénticos para todas las versiones. NT por sí mismo descubre qué versión está examinando una variable de una estructura de datos interna (el Registro). La licencia prohíbe a los usuarios modificar esta variable y así convertir la versión para estación de trabajo (de bajo costo) en las versiones para servidor o empresa (mucho más costosas). Aquí no haremos más distinciones entre estas versiones.

MS-DOS y todas las versiones previas de Windows eran sistemas monousuario. NT, en cambio, apoya la multiprogramación, así que varios usuarios pueden trabajar en la misma máquina al mismo tiempo. Por ejemplo, un servidor de red podría tener varios usuarios conectados simultáneamente por una red, cada uno de los cuales accede a sus propios archivos de forma protegida.

NT es un verdadero sistema operativo de 32 bits con multiprogramación. NT Apoya múltiples procesos de usuario, cada uno de los cuales tiene un espacio de direcciones virtual completo de 32 bits paginado por demanda. Además, el sistema mismo se escribió como código de 32 bits exclusivamente.

Una de las mejoras originales de NT respecto a Windows 95 fue su estructura modular que consistía en un núcleo más o menos pequeño que operaba en modo de kernel más varios procesos servidores que operaban en modo de usuario. Los procesos de usuario interactuaban con los procesos servidores empleando el modelo cliente-servidor: un cliente envía un mensaje de solicitud a un servidor; el servidor efectúa el trabajo y devuelve el resultado al cliente en un segundo mensaje. Tal estructura modular facilitó transportar el sistema a varias computadoras además de la línea Intel, incluidas la Alpha de DEC, PowerPC de IBM y MIPS

de SGI. Sin embargo, por razones de desempeño, a partir de NT 4.0 prácticamente todo el sistema se volvió a poner en el kernel.

Podríamos hablar por mucho tiempo acerca de la estructura interna de NT y el aspecto de su interfaz de llamadas al sistema. Puesto que lo que nos interesa primordialmente aquí es la máquina virtual que presentan los diversos sistemas operativos (es decir, las llamadas al sistema), daremos un breve resumen de la estructura del sistema y luego nos dedicaremos a la interfaz de llamadas al sistema.

La estructura de NT se ilustra en la figura 6-31. Consiste en varios módulos estructurados en capas que colaboran para implementar el sistema operativo. Cada módulo tiene una función en particular y una interfaz bien definida con los otros módulos. Casi todos los módulos están escritos en C, aunque una parte de la interfaz con dispositivos gráficos está escrita en C++ y una porción muy pequeña de las capas más bajas está escrita en lenguaje ensamblador.

En la parte inferior está una capa delgada llamada **capa de abstracción del hardware**. La tarea de esta capa es presentar al resto del sistema operativo dispositivos de hardware abstractos, carentes de las peculiaridades e idiosincrasias tan abundantes en el hardware real. Entre los dispositivos que se modelan están las cachés externas al chip, temporizadores, buses de E/S, controladores de interrupciones y controladores de DMA. Al exponer éstos al resto del sistema en una forma idealizada, se facilita el traslado de NT a otras plataformas de hardware, ya que casi todas las modificaciones requeridas están concentradas en un solo lugar.

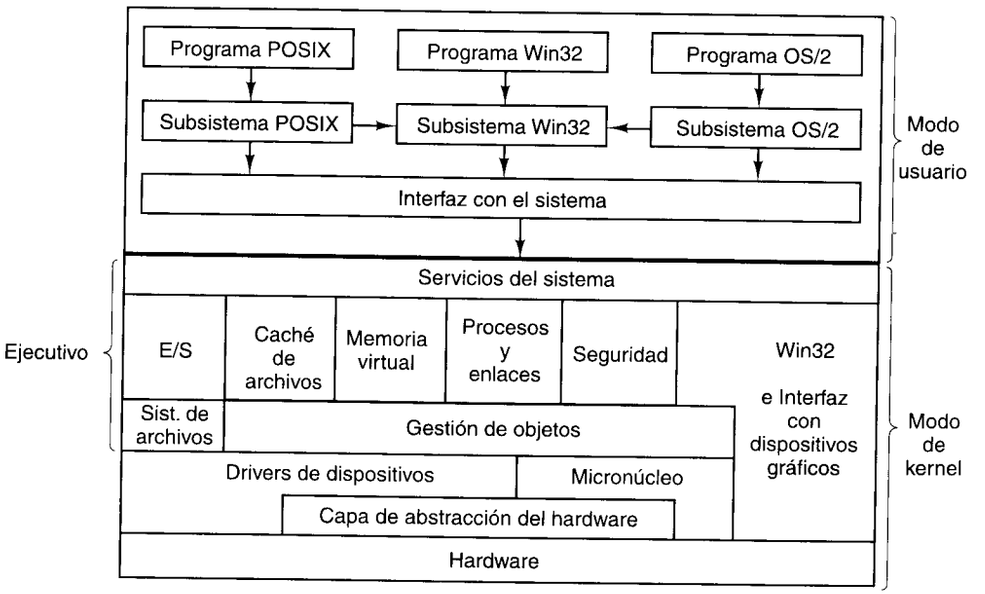


Figura 6-31. Estructura de Windows NT.

Arriba de la capa de abstracción del hardware está una capa que contiene el micronúcleo y los *drivers* de dispositivos. El micronúcleo y todos los *drivers* tienen acceso directo al hardware cuando lo necesitan, pues contienen código dependiente del hardware.

El **micronúcleo** apoya los objetos primitivos del núcleo, el manejo de interrupciones, trampas y excepciones, la planificación y sincronización de procesos, la sincronización de multiprocesadores y la gestión del tiempo. El propósito de esta capa es lograr que el resto del sistema operativo sea totalmente independiente del hardware, y por tanto muy portátil. El micronúcleo reside todo el tiempo en la memoria principal y no puede ser desalojado, aunque puede ceder temporalmente el control para atender las interrupciones de E/S.

Cada ***driver* de dispositivo** puede controlar uno o más dispositivos de E/S, pero un *driver* también puede hacer cosas no relacionadas con un dispositivo específico, como cifrar un flujo de datos o simplemente proporcionar acceso a estructuras de datos del núcleo. Puesto que los usuarios pueden instalar *drivers* de dispositivos nuevos, pueden afectar al núcleo y corromper el sistema. Por esta razón, los *drivers* se deben escribir con mucho cuidado.

Arriba del micronúcleo y los *drivers* de dispositivos está la porción superior del sistema operativo, llamada **ejecutivo**. El ejecutivo es independiente de la arquitectura y se puede trasladar a otras máquinas con relativamente poco esfuerzo. Esta parte abarca tres capas.

La capa más baja contiene los sistemas operativos y el gestor de objetos. Los **sistemas de archivos** apoyan el uso de archivos y directorios. El **gestor de objetos** maneja objetos que el núcleo conoce. Éstos incluyen procesos, enlaces (procesos ligeros dentro de un espacio de direcciones), archivos, directorios, semáforos, dispositivos de E/S, temporizadores y muchos más. El gestor de objetos también administra un espacio de nombres en el que se pueden colocar los objetos recién creados para poder hacer referencia a ellos después.

La siguiente capa consta de seis partes principales, como se muestra en la figura 6-31. El **gestor de E/S** proporciona un ámbito para administrar dispositivos de E/S y ofrece servicios de E/S genéricos. Este componente utiliza los servicios del sistema de archivos, que a su vez usa los *drivers* de dispositivos, además de los servicios del gestor de objetos.

El **gestor de cachés** mantiene los bloques de disco más recientemente usados en la memoria para agilizar el acceso a ellos en (el probable) caso de que se necesiten otra vez. Su tarea es determinar cuáles bloques es probable que se vayan a volver a necesitar y cuáles no. Es posible configurar NT con varios sistemas de archivos, en cuyo caso el gestor de caché dará servicio a todos ellos, con lo que se evita que cada uno tenga que realizar su propia administración de caché. Cuando se necesita un bloque, se le pide al gestor de caché que lo proporcione. Si el caché no tiene el bloque, el gestor se dirige al sistema de archivos apropiado para conseguirlo. Puesto que se puede establecer una correspondencia entre archivos y los espacios de direcciones de los procesos, el caché debe interactuar con el administrador de memoria virtual para establecer la consistencia requerida.

El **administrador de memoria virtual** implementa la arquitectura de memoria virtual paginada por demanda de NT. Este componente controla la correspondencia entre las páginas virtuales y los marcos de página físicos, y al hacerlo obliga al cumplimiento de las reglas de protección que restringen el acceso de cada proceso exclusivamente a las páginas que pertenecen a su espacio de direcciones y no a los de otros procesos (excepto en circunstancias especiales). El administrador de memoria también maneja ciertas llamadas al sistema relacionadas con la memoria virtual.

El **gestor de procesos y enlaces** maneja los procesos y los enlaces, lo que incluye su creación y destrucción; se ocupa de los mecanismos que permiten controlarlos, más que de las políticas de uso de procesos y enlaces.

El **gestor de seguridad** hace cumplir el complejo mecanismo de seguridad de NT, que cumple con los requisitos C2 del Libro Naranja del Departamento de la Defensa de Estados Unidos. El Libro Naranja especifica un gran número de reglas a las que debe ajustarse un sistema certificado, desde la verificación de autenticidad de los ingresos al sistema (*logins*) hasta el manejo del control de acceso y el hecho de que las páginas virtuales deben llenarse de ceros antes de reutilizarse.

La **interfaz con dispositivos gráficos** se encarga de la gestión de imágenes para el monitor y las impresoras, y cuenta con llamadas al sistema que permiten a los programas de usuario escribir en el monitor o las impresoras de forma independiente del dispositivo. Esta interfaz también contiene el administrador de ventanas y *drivers* de dispositivos de hardware. En versiones de NT previas a la 4.0, este componente estaba en el espacio de usuario pero su desempeño era muy bajo, por lo que Microsoft lo pasó al núcleo para hacerlo más rápido. El modelo Win32 también maneja la mayor parte de las llamadas del sistema. También estaba originalmente en el esapcio de usuario pero fue movido al núcleo para mejorar su rendimiento

Encima del ejecutivo hay una capa delgada llamada **servicios del sistema**. Su función es proporcionar una interfaz con el ejecutivo, la cual acepta las verdaderas llamadas al sistema de NT e invoca otras partes del ejecutivo para que las lleve a cabo.

Afuera del núcleo están los programas de usuario y los subsistemas del entorno. Se incluyen **subsistemas del entorno** porque no es recomendable que los programas de usuario efectúen llamadas al sistema directamente (aunque técnicamente sí pueden hacerlo). En vez de ello, cada subsistema de entorno exporta un conjunto (distinto) de llamadas a funciones que los programas de usuario pueden utilizar. En la figura 6-31 se muestran tres subsistemas de entorno: Win32 (para programas NT o Windows 95/98), POSIX (para programas UNIX que han sido trasladados) y OS/2 (para programas OS/2 que han sido trasladados).

Las aplicaciones Windows usan las funciones Win32 y se comunican con el subsistema Win32 para efectuar llamadas al sistema. El **subsistema Win32** acepta las llamadas a las funciones Win32 y usa el módulo de biblioteca de **interfaz con el sistema** (que en realidad es un archivo DLL; véase el capítulo 7) para efectuar las verdaderas llamadas al sistema NT requeridas para ejecutarlas.

También hay un **subsistema POSIX** que proporciona el apoyo mínimo para aplicaciones UNIX. Este subsistema reconoce sólo la funcionalidad P1003.1 y casi nada más. Se trata de un subsistema cerrado, lo que implica que sus aplicaciones no pueden usar los recursos del subsistema Win32, y esto restringe considerablemente su campo de acción. En la práctica, trasladar cualquier programa UNIX real a NT utilizando este subsistema es casi imposible. Se incluyó sólo porque algunas dependencias del gobierno estadounidense exigen que los sistemas operativos para computadoras del gobierno cumplan con P1003.1. Este subsistema no es autosuficiente y utiliza el subsistema Win32 para efectuar algunas de sus tareas, pero sin exportar la interfaz Win32 completa a sus programas de usuario.

El subsistema OS/2 tiene limitaciones similares de su funcionalidad y es probable que se omita en alguna versión futura. También usa el subsistema Win32. Además, hay un subsistema de entorno MS-DOS que no se muestra en la figura.

Habiendo examinado brevemente la estructura de NT, pasaremos a lo que principalmente nos interesa, que son los servicios que ofrece. Esta interfaz es la principal conexión

del programador con el sistema. Lamentablemente, Microsoft nunca ha publicado la lista completa de llamadas al sistema de NT, y además las modifica de una versión a otra. En tales condiciones, escribir programas que efectúen llamadas al sistema directamente es casi imposible.

Lo que Microsoft hizo fue definir un conjunto de llamadas que recibe el nombre de **interfaz para programación de aplicaciones** (**API**, *Application Programming Interface*) **Win32**, y que se conoce públicamente. Estas llamadas son procedimientos de biblioteca que efectúan llamadas al sistema para realizar el trabajo o bien, en algunos casos, realizan el trabajo en el mismo procedimiento de biblioteca del espacio de usuario o en el subsistema Win32. Las llamadas de la API Win32 no cambian al cambiar las versiones, a fin de promover la estabilidad. Sin embargo, también hay llamadas de la API de NT que sí pueden cambiar de una versión a otra. Aunque no todas las llamadas de la API Win32 son llamadas al sistema de NT, es mejor concentrarnos en ellas aquí más que en las verdaderas llamadas al sistema NT porque las llamadas de la API Win32 están bien documentadas y son más estables.

La filosofía de la API Win32 es totalmente distinta de la filosofía de UNIX. En esta última, todas las llamadas al sistema se conocen públicamente y forman una interfaz mínima: eliminar siquiera una de ellas reduciría la funcionalidad del sistema operativo. La filosofía Win32 es ofrecer una interfaz muy completa, a veces con tres o cuatro formas de hacer la misma cosa, e incluir muchas funciones que a todas luces no deberían ser llamadas al sistema (y que no lo son), como una llamada API para copiar todo un archivo.

Muchas llamadas de la API Win32 crean objetos del núcleo de un tipo u otro, incluidos archivos, procesos, enlaces, filas de procesamiento, etc. Cada llamada que crea un objeto del núcleo devuelve al invocador un resultado llamado **asa o manija** (*handle*). Esta manija se usa después para realizar operaciones con el objeto. Las manijas son específicas para el proceso que creó el objeto al que se hace referencia con la manija; no se pueden pasar directamente a otro proceso y usarse en él (así como los descriptores UNIX no pueden pasarse a otros procesos y usarse ahí). Sin embargo, en ciertas circunstancias es posible duplicar una manija y pasarla a otros procesos de forma protegida, permitiéndole accesar de forma controlada a objetos que pertenecen a otros procesos. Además, cada objeto tiene asociado un descriptor de seguridad que indica con detalle quién puede y quién no puede realizar qué tipos de operaciones con el objeto.

A veces se dice que NT está orientado a objetos porque la única forma de manipular objetos del núcleo es invocar métodos (funciones API) con sus manijas. Por otra parte, NT carece de algunas de las propiedades más básicas de los sistemas orientados a objetos, como herencia y polimorfismo.

La API Win32 también está disponible en Windows 95/98 (así como en el sistema operativo para aparatos electrónicos de consumo, Windows CE), con unas cuantas excepciones. Por ejemplo, Windows 95/98 no tiene seguridad, por lo que las llamadas API relacionadas con la seguridad simplemente devuelven códigos de error a Windows 95/98. Además, los nombres de archivo de NT usan el conjunto de caracteres Unicode, que no está disponible en Windows 95/98. También hay diferencias en los parámetros a algunas llamadas de funciones API. En NT, por ejemplo, todas las coordenadas de pantalla dadas en las funciones de gráficos son verdaderos números de 32 bits; en Windows 95/98 sólo se usan los 16 bits de orden bajo (por compatibilidad hacia atrás con Windows 3.1). La existencia de la API Win32 en varios

sistemas operativos distintos facilita el traslado de programas entre ellos pero también hace ver más claramente que está un tanto desacoplada de las llamadas al sistema reales. En la figura 6-32 se resumen algunas de las diferencias entre Windows 95/98 y NT.

Característica	Windows 95/98	NT 5.0
¿API Win32?	Sí	Sí
¿Sistema cabal de 32 bits?	No	Sí
¿Seguridad?	No	Sí
¿Correspondencias de archivos protegidas?	No	Sí
¿Espacio de direcciones privado para cada programa MS-DOS?	No	Sí
¿Opera de inmediato, sin tener que configurarse?	Sí	Sí
¿Reconoce UNICODE?	No	Sí
Se ejecuta en	Intel 80x86	80x86, Alpha
¿Apoyo a multiprocesadores?	No	Sí
¿Código reentrante dentro del sistema operativo?	No	Sí
¿El usuario puede escribir algunos datos críticos para el sistema operativo?	Sí	No

Figura 6-32. Algunas diferencias entre versiones de Windows.

6.4.2 Ejemplos de memoria virtual

En esta sección examinaremos la memoria virtual tanto en UNIX como en NT. En general, desde el punto de vista del programador hay muchas similitudes.

Memoria virtual en UNIX

El modelo de memoria de UNIX es sencillo. Cada proceso tiene tres segmentos: código, datos y pila, como se ilustra en la figura 6-33. En una máquina con un solo espacio de direcciones lineal, el código generalmente se coloca cerca de la base de la memoria, seguida de los datos. La pila se coloca en la parte alta de la memoria. El tamaño del código es fijo, pero los datos y la pila pueden crecer: la pila hacia abajo y los datos hacia arriba. Este modelo es fácil de implementar en casi cualquier máquina y es el modelo empleado por Solaris.

Además, si la máquina tiene paginación, se puede paginar todo el espacio de direcciones sin que los programas de usuario siquiera tengan conocimiento de ello; lo único que observan es que está permitido tener programas más grandes que la memoria física de la máquina. Los sistemas UNIX que no tienen paginación en general intercambian procesos enteros entre la memoria y el disco para poder tener un número arbitrariamente grande de procesos en tiempo compartido.

En el caso del Berkeley UNIX, la descripción anterior (memoria virtual paginada por demanda) cubre prácticamente todos los aspectos. En cambio, System V (y Solaris) incluye varias funciones que permiten a los usuarios administrar su memoria virtual de maneras más

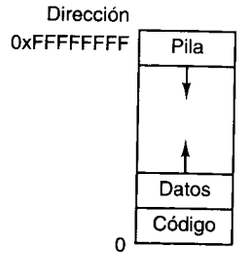


Figura 6-33. Espacio de direcciones de un solo proceso UNIX.

sofisticadas. La más importante de éstas es la capacidad de un proceso para hacer corresponder (una porción de) un archivo con una parte de su espacio de direcciones. Por ejemplo, si un archivo de 12 KB se hace corresponder con la dirección virtual 144K, una lectura de la palabra que está en la dirección 144K leerá la primera palabra del archivo. Esto permite efectuar E/S con archivos sin efectuar llamadas al sistema. Puesto que algunos archivos podrían exceder el tamaño del espacio de direcciones virtual, también es posible mapear sólo una porción de un archivo en lugar de todo. La correspondencia se establece abriendo primero el archivo y obteniendo un descriptor de archivo, *da*, que sirve para identificar el archivo por mapear. Luego el proceso efectúa una llamada

```
dir_fisica = mmap(dir_virtual, longitud, prot, banderas, da, dist_arch)
```

que hace corresponder *longitud* bytes que inician a *dist_arch* dentro del archivo, con el espacio de direcciones virtual a partir de *dir_virtual*. Como alternativa, se puede ajustar el parámetro *banderas* para pedir al sistema que escoja una dirección virtual, que entonces devolverá como *dir_ fisica*. La región mapeada debe ser un número entero de páginas y estar alineada con una frontera de página. El parámetro *prot* puede especificar cualquier combinación de permisos de lectura, escritura o ejecución. La correspondencia puede eliminarse después con unmap.

Varios procesos pueden establecer una correspondencia con el mismo archivo al mismo tiempo. Se ofrecen dos opciones para compartir. En la primera se comparten todas las páginas, de modo que las escrituras realizadas por un proceso sean visibles para todos los demás. Esta opción ofrece un camino de comunicación entre procesos con gran ancho de banda. La otra opción comparte las páginas en tanto ningún proceso las modifique. Apenas un proceso intente escribir en una página, obtendrá un fallo de protección, lo que hará que el sistema operativo le entregue una copia privada de la página para que escriba en ella. Este esquema, llamado **copiar al escribir**, se usa cuando cada proceso necesita creer que es el único que tiene una correspondencia con el archivo.

Memoria virtual en Windows NT

En NT cada proceso de usuario tiene su propio espacio de direcciones virtual. Las direcciones virtuales son de 32 bits, así que cada proceso tiene 4 GB de espacio de direcciones virtual. Los 2 GB inferiores están disponibles para el código y los datos del proceso. Los 2 GB superiores ofrecen acceso (limitado) a la memoria del núcleo, excepto en las versiones para

empresa de NT en las que la división puede ser 3 GB para el usuario y 1 GB para el núcleo o kernel. El espacio de direcciones virtual se pagina por demanda, con tamaño de página fijo (4 KB en el Pentium II).

Cada página virtual puede estar en uno de tres estados: libre, reservada o confirmada. Una **página libre** no se está usando actualmente y una referencia a ella causa un fallo de página. Cuando un proceso se inicia, todas sus páginas están en el estado libre hasta que el programa y los datos iniciales se hacen corresponder con su espacio de direcciones. Una vez que se ha hecho corresponder código o datos con una página, se dice que la página está **confirmada**. Una referencia a una página confirmada se mapea usando el hardware de memoria virtual y tiene éxito si la página está en la memoria principal. Si no es así, ocurre un fallo de página y el sistema operativo busca la página en el disco y la trae a la memoria. Una página virtual también puede estar en el estado **reservado**, lo que significa que no está disponible para mapearse hasta que la reserva se elimine explícitamente. Además de los atributos de libre, reservada y comprometida, las páginas tienen otros atributos, como poderse leer, escribir y ejecutar. Los 64 KB superiores y los 64 KB inferiores de la memoria siempre están libres, a fin de atrapar errores de apuntador (los apuntadores no inicializados suelen valer 0 o –1).

Cada página confirmada tiene una página sombra en el disco, donde se guarda cuando no está en la memoria principal. Las páginas libres y reservadas no tienen páginas sombra, por lo que las referencias a ellas causan fallos de página (el sistema no puede traer una página del disco si no existe en el disco). Las páginas sombra en el disco se organizan en uno o más archivos de paginación. El sistema operativo se mantiene al tanto de cuál página virtual corresponde a qué parte de cuál archivo de paginación. En el caso de texto de programa (sólo para ejecución), el archivo binario ejecutable contiene las páginas sombra; en el caso de páginas de datos, se usan archivos de paginación especiales.

NT, al igual que System V, permite hacer corresponder archivos directamente con regiones de los espacios de direcciones virtuales (es decir, trabaja con páginas consecutivas). Una vez que se ha hecho corresponder un archivo con un espacio de direcciones, se puede leer o escribir usando referencias a la memoria ordinarias.

Los archivos con correspondencia en la memoria se implementan de la misma manera que otras páginas confirmadas, sólo que las páginas sombra pueden estar en el archivo de disco en lugar del archivo de paginación. Por ello, cuando un archivo se mapea en la memoria la versión que está en la memoria podría no ser idéntica a la versión en disco (por escrituras recientes en el espacio virtual de direcciones). Sin embargo, cuando el archivo no está mapeado, o cuando se borra explícitamente, la versión en disco se actualiza.

NT permite explícitamente a dos o más procesos mapear el mismo archivo al mismo tiempo, posiblemente en diferentes direcciones virtuales. Mediante la lectura y escritura de palabras en la memoria, los procesos pueden comunicarse entre sí y enviarse datos con gran ancho de banda, ya que no es necesario copiar. Diferentes procesos podrían tener diferentes permisos de acceso. Puesto que todos los procesos que usan un archivo mapeado comparten las mismas páginas, los cambios hechos por uno de ellos son visibles de inmediato para todos los demás, aunque todavía no se haya actualizado el archivo de disco.

La API Win32 contiene varias funciones que permiten a un proceso administrar explícitamente su memoria virtual. Las más importantes de estas funciones se enumeran en la figura

6-34. Todas ellas operan sobre una región que consiste en una sola página o bien en una sucesión de dos o más páginas que son consecutivas en el espacio de direcciones virtual.

Función API	Significado
VirtualAlloc	Reservar o confirmar una región
VirtualFree	Liberar o desconfirmar una región
VirtualProtect	Cambiar la protección leer/escribir/ejecutar de una región
VirtualQuery	Consultar la situación de una región
VirtualLock	Hacer a una región residente en la memoria (es decir, inhabilitar su paginación)
VirtualUnlock	Hacer una región paginable de la forma normal
CreateFileMapping	Crear un objeto de mapeo de archivo y (opcionalmente) asignarle un nombre
MapViewOfFile	Mapear (parte de) un archivo en el espacio de direcciones
UnmapViewOfFile	Quitar un archivo mapeado del espacio de direcciones
OpenFileMapping	Abrir un objeto de mapeo de archivo creado previamente

Figura 6-34. Principales funciones de la API para administrar la memoria virtual en Windows NT.

Las primeras cuatro funciones de la API no requieren explicación. Las dos que siguen confieren a un proceso la capacidad de fijar hasta 30 páginas en memoria de modo que no se intercambien a disco, y revocar esta propiedad. Un programa en tiempo real podría necesitar esta propiedad, por ejemplo. El sistema operativo establece un límite para evitar que los procesos se vuelvan acaparadores de los recursos . Aunque no se muestran en la figura 6-34, NT también tiene funciones que permiten a un proceso acceder a la memoria virtual de un proceso distinto sobre el cual se le ha concedido control (es decir, para el cual tiene un mango).

Las últimas cuatro funciones de la lista son para administrar archivos con correspondencia en la memoria. Para mapear un archivo, primero hay que crear un objeto de mapeo de archivo con CreateFileMapping. Esta función devuelve un mango para el objeto de mapeo de archivo y opcionalmente asienta en el sistema de archivos un nombre para ese objeto a fin de que otro proceso pueda usarlo. Las dos funciones que siguen establecen y anulan la correspondencia de archivos, respectivamente. Un proceso puede usar la última para mapear un archivo que actualmente también está mapeado por un proceso distinto. De este modo, dos o más procesos pueden compartir regiones de sus espacios de direcciones.

Estas funciones de la API son las fundamentales sobre las que se construye el resto del sistema de administración de memoria. Por ejemplo, hay funciones API que asignan espacio a estructuras de datos en uno o más montículos, y que liberan ese espacio. Se usan montículos para almacenar estructuras de datos que se crean y destruyen dinámicamente. No se aplica recolección de basura a los montículos, así que es responsabilidad del software usuario liberar los bloques de memoria virtual que ya no se usan. (La recolección de basura es la eliminación automática de estructuras de datos no utilizadas, por parte del sistema.) El uso de montículos en NT es similar al uso de la función *malloc* en sistemas UNIX, excepto que ahí puede haber múltiples montículos administrados de forma independiente.

6.4.3 Ejemplos de E/S virtual

La tarea fundamental de cualquier sistema operativo es proveer servicios a los programas de usuario, principalmente servicios de E/S como leer y escribir archivos. Tanto UNIX como NT ofrecen una amplia variedad de servicios de E/S a los programas de usuario. Para casi todas las llamadas al sistema de UNIX, NT tiene una llamada equivalente, pero al contrario no se cumple, pues NT tiene muchas más llamadas y cada una es mucho más complicada que su contraparte en UNIX.

E/S virtual en UNIX

Gran parte de la popularidad del sistema UNIX se debe directamente a su sencillez, que a su vez es un resultado directo de la organización del sistema de archivos. Un archivo ordinario es una sucesión de bytes de 8 bits que comienzan en 0 y llegan hasta un máximo de $2^{32}-1$ bytes. El sistema operativo en sí no impone una estructura de registros a los archivos, aunque muchos programas de usuario consideran a los archivos de texto ASCII como secuencias de líneas, cada una de las cuales termina con un salto de línea.

Cada archivo abierto tiene asociado un apuntador al siguiente byte que se leerá o escribirá. Las llamadas al sistema read y write leen y escriben datos a partir de la posición en el archivo indicada por dicho apuntador. Ambas llamadas adelantan el apuntador después de la operación en una cantidad igual al número de bytes transferidos. Sin embargo, se puede accesar aleatoriamente a los archivos asignando explícitamente un valor específico al apuntador del archivo.

Además de los archivos ordinarios, el sistema UNIX también reconoce archivos especiales, que sirven para accesar a dispositivos de E/S. A cada dispositivo de E/S se le asigna por lo regular uno o más archivos especiales. Al leer del archivo especial asociado, y escribir en él, un programa puede leer del dispositivo de E/S o escribir en él. Los discos, impresoras, terminales y muchos otros dispositivos se manejan de esta manera.

Las principales llamadas al sistema de UNIX para archivos se enumeran en la figura 6-35. La llamada creat (sin la *e* final) se puede usar para crear un archivo nuevo. Hoy día esta llamada no es estrictamente necesaria, porque open también puede crear un archivo nuevo. Unlink elimina un archivo, siempre que el archivo sólo esté en un directorio.

Open sirve para abrir archivos existentes (y crear archivos nuevos). La bandera modo indica cómo se abrirá (para lectura, para escritura, etc.). La llamada devuelve un entero pequeño llamado **descriptor de archivo** que identifica al archivo en llamadas subsecuentes. La E/S real con archivos se efectúa con read y write, cada una de las cuales tiene un descriptor de archivo que indica cuál archivo debe usarse, un buffer para colocar los datos leídos o por escribir, y una cuenta de bytes que indica cuántos datos deben transmitirse. Lseek sirve para ubicar el apuntador de archivo, lo cual hace posible el acceso aleatorio a los archivos.

Stat devuelve información acerca de un archivo, incluido su tamaño, la hora del último acceso, el propietario y más. Chmod cambia el modo de protección de un archivo, por ejemplo, permitiendo o prohibiendo a usuarios distintos del propietario que lo lean. Por último, fcntl realiza varias operaciones diversas con un archivo, como ponerle o quitarle un candado.

La figura 6-36 ilustra el funcionamiento de las principales llamadas para E/S con archivos. Este código es mínimo y no incluye la verificación de errores necesaria. Antes de ingre-

Llamada al sistema	Significado
creat(name, mode)	Crear un archivo, *modo* especifica el modo de protección
unlink(name)	Eliminar un archivo (suponiendo que sólo hay un vínculo a él)
open(name, mode)	Abrir o crear un archivo y devolver un descriptor de archivo
close(fd)	Cerrar un archivo
read(fd, buffer, count)	Leer *cuenta* bytes y colocarlos en *buffer*
write(fd, buffer, count)	Escribir *cuenta* bytes de *buffer*
lseek(fd, offset, w)	Mover el apuntador de archivo según indican *distancia* y *w*
stat(name, buffer)	Devolver información acerca de un archivo
chmod(name, mode)	Cambiar el modo de protección de un archivo
fcntl(fd, cmd, ...)	Realizar diversas operaciones de control como poner un candado a (parte de) un archivo

Figura 6-35. Principales llamadas al sistema UNIX para archivos.

sar en el ciclo, el programa abre un archivo existente, *datos*, y crea un archivo nuevo, *anuevo*. Cada llamada devuelve un descriptor de archivo, *dain* y *daout*, respectivamente. Los segundos parámetros de las dos llamadas son bits de protección que especifican que los archivos son para leerse y escribirse, respectivamente. Ambas llamadas devuelven un descriptor de archivo. Si falla open o create, se devuelve un descriptor de archivo negativo para indicar que la llamada falló.

```
// Open the file descriptors
dain = open("datos", 0);
daout = creat("anuevo", ProtectionBits);

// Ciclo de copiado
do {
      cuenta = read(dain, buffer, bytes);
      if (cuenta > 0) write(daout, buffer, cuenta);
} while (cuenta > 0);

// Cerrar los archivos
close(dain);
close(daout);
```

Figura 6-36. Fragmento de programa para copiar un archivo usando las llamadas al sistema UNIX. Este fragmento está en C porque Java oculta las llamadas al sistema de bajo nivel y estamos tratando de exponerlas.

La llamada a read tiene tres parámetros: un descriptor de archivo, un buffer y una cuenta de bytes. La llamada trata de leer el número deseado de bytes del archivo indicado y colocarlos en el buffer. El número de bytes que realmente se leyeron se devuelve en *cuenta*, que será menor que *bytes* si el archivo era demasiado corto. La llamada write deposita los bytes recién leídos en el archivo de salida. El ciclo continúa hasta que el archivo de entrada se termina de leer, y entonces el ciclo termina y ambos archivos se cierran.

Los descriptores de archivo en UNIX son enteros pequeños (en general menores que 20). Los descriptores 0, 1 y 2 son especiales y corresponden a la **entrada estándar**, la **salida estándar** y el **error estándar**, respectivamente. Normalmente, éstas se refieren al teclado, el monitor y la pantalla, respectivamente, pero el usuario puede redirigirlas a archivos. Muchos programas UNIX reciben sus entradas de la entrada estándar y escriben los resultados procesados en la salida estándar. Tales programas se denominan **filtros**.

Algo estrechamente relacionado con el sistema de archivos es el sistema de directorios. Cada usuario puede tener varios directorios, y cada directorio puede contener tanto archivos como subdirectorios. Los sistemas UNIX normalmente se configuran con un directorio principal, llamado **directorio raíz**, que contiene los subdirectorios *bin* (para programas que se ejecutan con frecuencia), *dev* (para los archivos especiales de dispositivos de E/S), *lib* (para bibliotecas) y *usr* (para los directorios de los usuarios), como se muestra en la figura 6-37. En este ejemplo, el subdirectorio *usr* contiene los subdirectorios *ast* y *jim*. El directorio *ast* contiene dos archivos, *data* y *foo.p*, y un subdirectorio, *bin*, que contiene cuatro juegos.

Los archivos pueden nombrarse dando su **ruta** desde el directorio raíz. Un camino contiene una lista de todos los directorios por los que se pasa desde la raíz hasta el archivo, separando los nombres de directorio con diagonales. Por ejemplo, el nombre de camino absoluto de *game2* es */usr/ast/bin/game2*. Un camino que comienza en la raíz se llama **ruta absoluta**.

En todo momento, cada programa en ejecución tiene un **directorio de trabajo**. Los nombres de ruta también pueden ser relativos al directorio de trabajo, cuyo nombre no comienza con una diagonal, para distinguirlo de los nombres de ruta absolutos. Tales rutas se llaman **rutas relativas**. Si */usr/ast* es el directorio de trabajo, se puede acceder a *game3* usando el camino *bin/game3*. Un usuario puede crear un **vínculo** con un archivo de otra persona usando la llamada al sistema link. En el ejemplo anterior, */usr/ast/bin/game3* y */usr/jim/jotto* accesan el mismo archivo. A fin de evitar ciclos en el sistema de directorios, no se permiten vínculos con directorios. Las llamadas open y creat reciben nombres de ruta absolutos o bien relativos como argumentos.

Las principales llamadas al sistema UNIX para gestión de directorios se enumeran en la figura 6-38. Mkdir crea un directorio nuevo y rmdir elimina un directorio existente (vacío). Las siguientes tres llamadas sirven para leer entradas de directorio. La primera abre el directorio, la siguiente lee entradas de él y la última cierra el directorio. Chdir cambia el directorio de trabajo.

Link crea una entrada de directorio nueva que apunta a un archivo existente. Por ejemplo, la entrada */usr/jim/jotto* podría haberse creado con la llamada

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

o una llamada equivalente empleando nombres de camino relativos, dependiendo del directorio de trabajo del programa que efectúa la llamada. Unlink elimina una entrada de directorio. Si el archivo sólo tiene un vínculo, el archivo se elimina; si tiene dos o más vínculos, se conserva. No importa si el vínculo eliminado es el original o una copia creada posteriormente. Una vez creado un vínculo, es un ciudadano de primera clase, indistinguible del original. La llamada

```
unlink("/usr/ast/bin/game3")
```

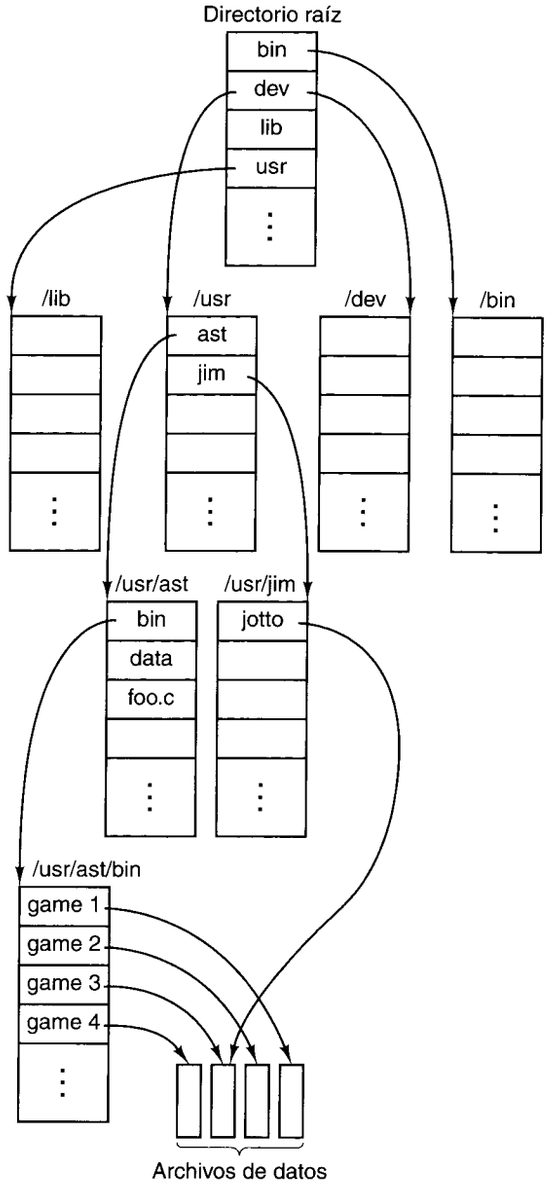


Figura 6-37. Parte de un sistema de directorios UNIX representativo.

hace que en adelante sólo se pueda accesar a *game3* por la ruta */usr/jim/jotto*. Podemos usar link y unlink de esta forma para "trasladar" archivos de un directorio a otro.

Cada archivo (incluidos los directorios, que también son archivos) está asociado a un mapa de bits que indica quién puede accesar al archivo. El mapa contiene tres campos RWX:

Llamada al sistema	Significado
mkdir(name, mode)	Create un directorio nuevo
rmdir(name)	Eliminar un directorio vacío
opendir(name)	Abrir un directorio para leerlo
readdir(dirpointer)	Leer la siguiente entrada de un directorio
closedir(dirpointer)	Cerrar un directorio
chdir(dirname)	Cambiar el directorio de trabajo a *nombredir*
link(name1, name2)	Crear una entrada de directorio *nombre2* que apunta a *nombre1*
unlink(name)	Eliminar *nombre* de su directorio

Figura 6-38. Principales llamadas para gestión de directorios en UNIX.

el primero controla los permisos de lectura (*Read*), escritura (*Write*) y ejecución (*eXecute*) del dueño; el segundo hace lo propio para otros del grupo del dueño; y el tercero, para el resto del mundo. Así, RWX R-X--X implica que el dueño puede leer, escribir y ejecutar el archivo (obviamente, se trata de un programa ejecutable, pues si no el permiso de ejecutar estaría desactivado), mientras que otros de su grupo pueden leerlo y ejecutarlo, y la gente extraña sólo puede ejecutarlo. Con estos permisos, los extraños pueden usar el programa pero no robárselo (copiarlo) porque no tienen permiso de lectura. La asignación de usuarios a grupos corre por cuenta del administrador del sistema, generalmente llamado **superusuario**. El superusuario también tiene el poder de supeditar el mecanismo de protección y leer, escribir o ejecutar cualquier archivo.

Veamos brevemente cómo se implementan los archivos y directorios en UNIX. Si quiere un tratamiento más completo, vea (Vahalia, 1996). Cada archivo (y cada directorio, porque un directorio también es un archivo) tiene asociado un bloque de información de 64 bytes llamado **nodo i** o **inodo**. El nodo i indica quién es el propietario del archivo, cuáles son los permisos, dónde están los datos y cosas así. Los nodos i de los archivos de cada disco se encuentran en secuencia numérica al principio del disco o bien, si el disco está dividido en grupos de cilindros, al principio de cada grupo de cilindros. Así, al recibir un número de nodo i, el sistema UNIX puede localizar el nodo i con sólo calcular su dirección en disco.

Una entrada de directorio consta de dos partes: un nombre de archivo y un número de nodo i. Cuando un programa ejecuta

```
open("foo.c", 0)
```

el sistema busca el nombre de archivo "foo.c" en el directorio de trabajo para encontrar el número de nodo i para ese archivo. Una vez que encuentra el número de nodo i, el sistema puede leer el nodo i, que le proporciona toda la información acerca del archivo.

Cuando se especifica un nombre de camino más largo, los pasos básicos que acabamos de bosquejar se repiten varias veces hasta que se analiza el camino completo. Por ejemplo, para encontrar el número de nodo i para */usr/ast/data*, el sistema primero busca en el directorio raíz una entrada *usr*. Una vez que encuentra el nodo i para *usr*, el sistema puede leer ese archivo (un directorio es un archivo en UNIX). En este archivo, el sistema busca una entrada *ast*, y así encuentra el número de nodo i del archivo */usr/ast*. Si lee */usr/ast*, el sistema puede

entonces encontrar la entrada para *data*, y por tanto el número de nodo i para */usr/ast/data*. Una vez que el sistema tiene el número de nodo i del archivo, puede averiguar todo acerca del archivo leyendo el nodo i.

El formato, contenido y organización de un nodo i varían un poco de un sistema a otro (sobre todo cuando se trabaja con redes), pero casi siempre se encuentran los siguientes elementos en cada nodo i.

1. El tipo de archivo, los nueve bits de protección RWX y unos cuantos bits más.
2. El número de vínculos con el archivo (número de entradas de directorio que apuntan a él).
3. La identidad del propietario.
4. El grupo del propietario.
5. La longitud del archivo en bytes.
6. Trece direcciones en disco.
7. La hora en que se leyó por última vez el archivo.
8. La hora en que se escribió por última vez el archivo.
9. La hora en que se modificó por última vez el nodo i.

El tipo de archivo distingue los archivos ordinarios, directorios y dos tipos de archivos especiales, para dispositivo de E/S estructurados en bloques y no estructurados, respectivamente. El número de vínculos y la identificación del propietario ya se explicaron. La longitud del archivo es un entero de 32 bits que da el byte más alto que tiene un valor. Es perfectamente válido crear un archivo, efectuar un lseek hasta la posición 1,000,000, y escribir un byte, lo que da una longitud de archivo de 1,000,001. Sin embargo, el archivo *no* solicitaría almacenamiento para todos los bytes "faltantes".

Las primeras 10 direcciones en disco apuntan a bloques de datos. Con un tamaño de bloque de 1024 bytes, se pueden manejar de este modo archivos de hasta 10,240 bytes. La dirección 11 apunta a un bloque de disco, llamado **bloque indirecto**, que contiene 256 direcciones en disco. Los archivos de hasta $10{,}240 + 256 \times 1024 = 272{,}384$ bytes se manejan de esta manera. En el caso de archivos aún más grandes, la dirección 12 apunta a un bloque que contiene las direcciones de 256 bloques indirectos, con lo que se pueden manejar archivos de hasta $272{,}384 + 256 \times 256 \times 1024 = 67{,}381{,}248$ bytes. Si este esquema de **bloque indirecto doble** aún es demasiado pequeño, se usa la dirección en disco 13 que apunta a un **bloque indirecto triple** que contiene las direcciones de 256 bloques indirectos dobles. Si se usan las direcciones indirectas, directas, sencillas y dobles, es posible direccionar hasta 16,843,018 bloques que dan un tamaño de archivo máximo teórico de 17,247,250,432 bytes. Puesto que los apuntadores de archivo están limitados a 32 bits, el límite superior práctico es en realidad de 4,294,967,295 bytes. Los bloques de disco libres se mantienen en una lista enlazada. Cuando se necesita un bloque nuevo, se toma el siguiente bloque de la lista. El resultado es que los bloques de cada archivo están dispersos aleatoriamente por todo el disco.

A fin de hacer más eficiente la E/S de disco, cuando un archivo se abre su nodo i se copia en una tabla de la memoria principal y se guarda ahí para agilizar las referencias en tanto el archivo siga abierto. Además, se mantiene en la memoria una reserva de bloques de disco a los que recientemente se hizo referencia. Dado que la mayor parte de los archivos se lee en forma secuencial, es común que una referencia a un archivo requiera el mismo bloque de disco que la referencia anterior. Para reforzar este efecto, el sistema también trata de leer el *siguiente* bloque de un archivo, antes de que se haga referencia a él, a fin de agilizar el procesamiento. El usuario no percibe toda esta optimación; cuando un usuario emite una llamada read, el programa se suspende hasta que los datos solicitados están en el buffer.

Con estos antecedentes, podemos ver cómo funciona la E/S de archivos. Open hace que el sistema busque en los directorios el camino especificado. Si la búsqueda tiene éxito, se lee el nodo i y se coloca en una tabla interna. Las llamadas read y write hacen que el sistema calcule el número de bloque a partir de la posición actual en el archivo. Las direcciones en disco de los 10 primeros bloques siempre están en la memoria principal (en el nodo i); los bloques con números más grandes requieren la lectura de uno o más bloques indirectos primero. Lseek simplemente modifica el apuntador a la posición actual, sin efectuar E/S.

Ahora también es fácil entender link y unlink. Link busca su primer argumento para encontrar el número de nodo i; luego crea una entrada de directorio para el segundo argumento y coloca el número de nodo i del primer archivo en esa entrada; por último, incrementa en uno la cuenta de vínculos en el nodo i. Unlink elimina una entrada de directorio y decrementa la cuenta de vínculos en el nodo i. Si la cuenta es cero, el archivo se borra y todos los bloques se colocan en la lista libre.

E/S virtual en Windows NT

NT maneja varios sistemas de archivos, de los cuales los más importantes son **NTFS** (**NT File System**) y el sistema de archivos **FAT** (**tabla de asignación de archivos**, *File Allocation Table*). El primero es un sistema de archivos nuevo creado específicamente para NT; el segundo es el antiguo sistema de archivos de MS-DOS, que también se usa en Windows 95/98 (aunque con apoyo para nombres de archivo más largos). Puesto que el sistema de archivos FAT es básicamente obsoleto (aunque varios cientos de millones de personas siguen usándolo), estudiaremos NTFS a continuación. A partir de NT 5.0, también se reconoce el sistema de archivos FAT32 empleado en versiones posteriores de Windows 95 y en Windows 98.

Los nombres de archivo en NTFS pueden tener hasta 255 caracteres, y están en Unicode, lo que permite a personas de países que no usan el alfabeto latino (por ejemplo, Japón, India e Israel) escribir nombres de archivo en su lenguaje nativo. (De hecho, NT usa Unicode internamente; las versiones a partir de NT 5.0 tienen un solo binario que se puede usar en cualquier país y emplear el idioma local porque todos los menús, mensajes de error, etc., se guardan en archivos de configuración que dependen del país.) NTFS apoya plenamente nombres sensibles a la caja (de modo que *foo* es diferente de *FOO*). Lo malo es que la API Win32 no apoya plenamente la sensibilidad a la caja para los nombres de archivo, y en absoluto para los nombres de directorio, por lo que los programas que usan Win32 no tienen esta ventaja.

Al igual que en UNIX, un archivo no es más que una secuencia lineal de bytes, aunque hasta un máximo de $2^{64} - 1$. También existen apuntadores de archivo, como en UNIX, pero tienen capacidad de 64 bits, en lugar de 32, a fin de manejar la longitud de archivo máxima. Las llamadas a funciones de la API Win32 para manipular archivos y directorios son similares a grandes rasgos a sus contrapartes en UNIX, excepto que casi todas tienen más parámetros y el modelo de seguridad es distinto. La apertura de un archivo devuelve una manija, que entonces se usa para leer y escribir el archivo. Sin embargo, a diferencia de UNIX, las manijas no son enteros pequeños, y la entrada estándar, la salida estándar y el error estándar tienen que adquirirse explícitamente en lugar de estar predefinidos como 0, 1 y 2 (excepto en modo de consola, en el que se preabren). Las principales funciones de la API Win32 para gestión de archivos se enumeran en la figura 6-39.

Función API	UNIX	Significado
CreateFile	open	Crear un archivo o abrir un archivo existente; devolver una manija
DeleFile	unlink	Destruir un archivo existente
CloseHandle	close	Cerrar un archivo
ReadFile	read	Leer datos de un archivo
WriteFile	write	Escribir datos en un archivo
SetFilePointer	lseek	Hacer que el apuntador de archivo apunte a un lugar específico del archivo
GetFileAtributes	stat	Devolver las propiedades del archivo
LockFile	fcntl	Poner candado a una región del archivo para tener exclusión mutua
UnlockFile	fcntl	Quitar el candado puesto antes a una región del archivo

Figura 6-39. Principales funciones de la API Win32 para E/S con archivos. La segunda columna da el equivalente más cercano en UNIX.

Examinemos estas llamadas brevemente. CreateFile puede servir para crear un archivo nuevo y devolver una manija para él. Esta función API también sirve para abrir archivos existentes porque no existe una función API open. No enumeramos los parámetros para las funciones API de NT porque son muy voluminosos. Por ejemplo, CreateFile tiene siete parámetros, a saber:

1. Un apuntador al nombre del archivo que se creará o abrirá.
2. Banderas que indican si el archivo puede leerse, escribirse o ambas cosas.
3. Banderas que indican si varios procesos pueden abrir el archivo a la vez.
4. Un apuntador al descriptor de seguridad, que indica quién puede accesar el archivo.
5. Banderas que indican qué hacer si el archivo existe/no existe.
6. Banderas que se ocupan de atributos como archivado, compresión, etc.
7. La manija de un archivo cuyos atributos deberán clonarse para el nuevo archivo.

Las siguientes seis funciones API de la figura 6-39 son muy parecidas a las llamadas al sistema UNIX correspondientes. Las últimas dos permiten poner y quitar un candado a una región de un archivo para que un proceso pueda obtener exclusión mutua garantizada en su acceso a ella.

Con estas funciones API es posible escribir un procedimiento para copiar un archivo, análogo a la versión UNIX de la figura 6-36. En la figura 6-40 se muestra un procedimiento semejante (sin verificación de errores), diseñado para imitar la estructura de la figura 6-36. En la práctica no sería necesario escribir un programa para copiar archivos porque existe la función API CopyFile (que ejecuta algo parecido a este programa como procedimiento de biblioteca).

```
// Abrir archivos para entrada y salida.
mangoin = CreateFile("datos", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
mangoout = CreateFile("anuevo", GENERIC_WRITE, 0, NULL, CREATE, ALWAYS,
      FILE_ATTRIBUTE_NORMAL, NULL);

// Copiar el archivo.
do {
      s = ReadFile(mangoin, buffer, BUF_SIZE, &cuenta, NULL);
      if (s > 0 && cuenta > 0) WriteFile(mangoout, buffer, cuenta, &ocnt, NULL);
while (s > 0 && cuenta > 0);

// Cerrar los archivos.
CloseHandle(mangoin);
CloseHandle (mangoout);
```

Figura 6-40. Fragmento de programa para copiar un archivo usando las funciones API de Windows NT. Este fragmento está en C porque Java oculta las llamadas al sistema de bajo nivel y estamos tratando de exponerlas.

NT maneja un sistema de archivos jerárquico similar al de UNIX. Sin embargo, el separador de nombres componentes es \ en lugar de /, un fósil heredado de MS-DOS. Existe el concepto de directorio de trabajo vigente y los nombres de ruta pueden ser relativos o absolutos. Una diferencia importante respecto a UNIX es que UNIX permite montar sistemas de archivos que están en diferentes discos y máquinas juntos en un solo árbol de nombres, lo que oculta la estructura de discos de todo el software. NT 4.0 no tiene esta propiedad, por lo que los nombres de archivo absolutos deben comenzar con una letra de unidad que indica de cuál disco lógico se trata, como en *C:\windows\system\foo.dll*. A partir de NT 5.0, se añadió el montaje de sistemas de archivos al estilo UNIX.

Las principales funciones API para gestión de directorios se dan en la figura 6-41, otra vez con sus equivalentes UNIX más cercanos. Creemos que las funciones no requieren explicación.

Cabe señalar que NT 4.0 no manejaba vínculos de archivos. En el nivel del escritorio gráfico se manejaban atajos, pero estas construcciones eran internas del escritorio y no tenían contraparte en el sistema de archivos mismo. Un programa de aplicación no podía incluir un archivo en un segundo directorio sin copiar todo el archivo. A partir de NT 5.0 se añadió la vinculación de archivos al sistema de archivos propiamente dicho.

Función API	UNIX	Significado
CreateDirectory	mkdir	Crear un directorio nuevo
RemoveDirectory	rmdir	Eliminar un directorio vacío
FindFirstFile	opendir	Inicializar para comenzar a leer las entradas de un directorio
FindNextFile	readdir	Leer la siguiente entrada del directorio
MoveFile		Pasar un archivo de un directorio a otro
SetCurrentDirectory	chdir	Cambiar el directorio de trabajo vigente

Figura 6-41. Principales funciones de la API Win32 para gestión de directorios. La segunda columna da el equivalente UNIX más cercano, si existe.

NT tiene un mecanismo de seguridad mucho más completo que el de UNIX. Aunque hay cientos de funciones API relacionadas con la seguridad, la breve descripción que sigue da la idea general. Cuando un usuario ingresa en el sistema, su proceso inicial recibe una **ficha de acceso** del sistema operativo. La ficha de acceso contiene el **identificador de seguridad** (**SID**, *Security ID*) del usuario, una lista de los grupos de seguridad a los que el usuario pertenece, privilegios especiales que estén disponibles, y unas cuantas cosas más. El motivo para tener una ficha de acceso es concentrar toda la información sobre seguridad en un solo lugar fácil de encontrar. Todos los procesos creados por este proceso heredan la misma ficha de acceso.

Uno de los parámetros que pueden proporcionarse cuando se crea cualquier objeto es su **descriptor de seguridad**, el cual contiene una lista de entradas llamada **lista de control de acceso** (**ACL**, *Access Control List*). Cada entrada permite o prohíbe que algún SID o grupo lleve a cabo algún conjunto de las operaciones con el objeto. Por ejemplo, un archivo podría tener un descriptor de seguridad que especifique que Leonor no tiene ningún acceso al archivo, Carlos puede leer el archivo, Linda puede leer o escribir el archivo, y que todos los miembros del grupo XYZ pueden leer la longitud del archivo pero nada más.

Cuando un proceso trata de realizar una operación con un objeto usando la manija que obtuvo cuando creó el objeto, el gestor de seguridad obtiene la ficha de acceso del proceso y revisa la lista de entradas de la ACL en orden. Tan pronto como se encuentra una entrada que coincide con el SID del invocador o uno de los grupos del invocador, el acceso que se encuentra ahí se toma como definitivo. Por esta razón, se acostumbra colocar en la ACL las entradas que niegan acceso adelante de las que otorgan acceso, para que un usuario al que se niega específicamente el acceso no pueda introducirse por una "puerta trasera" al ser miembro de un grupo al que se le permite el acceso. El descriptor de seguridad también contiene información que sirve para auditar los accesos al objeto.

Veamos ahora con brevedad cómo se implementan los archivos y directorios en NT. Cada disco se divide estáticamente en volúmenes autosuficientes, que equivalen a las particiones de disco en UNIX. Cada volumen contiene archivos, directorios, mapas de bits y otras estructuras de datos para manejar su información. El volumen se organiza como una secuencia lineal de **cúmulos** cuyo tamaño es fijo para cada volumen y varía entre 512 bytes y 64 KB, dependiendo del tamaño de volumen. Se hace referencia a los cúmulos por su distancia al principio del volumen empleando números de 64 bits.

La principal estructura de datos de cada volumen es la **tabla maestra de archivos** (**MFT**, *Master File Table*), que tiene una entrada para cada archivo y directorio del volumen. Estas entradas son análogas a los nodos i en UNIX. La MFT en sí es un archivo, y como tal puede colocarse en cualquier lugar del volumen, lo que elimina el problema que UNIX tiene con bloques de disco defectuosos en medio de los nodos i.

En la figura 6-42 se muestra la MFT, que comienza con una cabecera que contiene información acerca del volumen, como (apuntadores a) el directorio raíz, el archivo de arranque, el archivo de bloques defectuosos, la administración de lista libre, etc. Después viene una entrada por cada archivo o directorio, de 1 KB excepto cuando el tamaño de cúmulo es de 2 KB o más. Cada entrada contiene todos los metadatos (información administrativa) acerca del archivo o directorio. Se permiten varios formatos, y en la figura 6-42 se muestra uno de ellos.

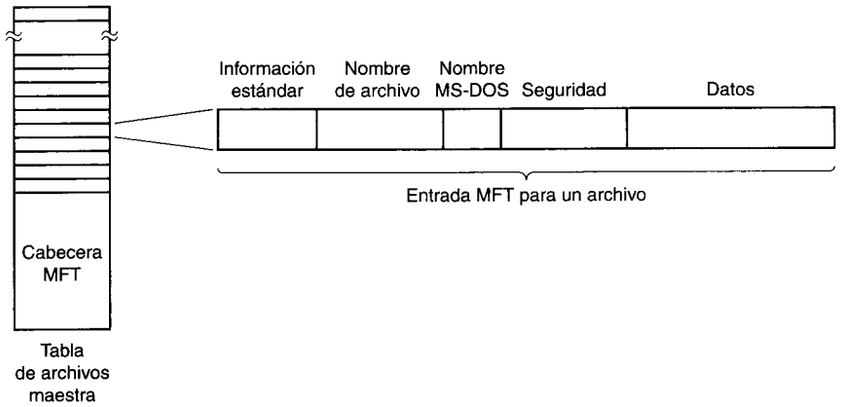


Figura 6-42. La tabla maestra de archivos de Windows NT.

El campo de información estándar contiene información como las marcas de tiempo que necesita POSIX, el número de vínculos a disco duro, los bits de sólo lectura y almacenamiento, etc. Se trata de un campo de longitud fija y siempre está presente. El nombre de archivo tiene una longitud variable, hasta 255 caracteres Unicode. Para que tales archivos sean accesibles a programas viejos de 16 bits, los archivos pueden tener también un nombre MS-DOS, que consiste en ocho caracteres alfanuméricos seguidos opcionalmente de un punto y una extensión de hasta tres caracteres alfanuméricos. Si el archivo ya se ajusta a la regla de nombres 8 + 3 de MS-DOS, no se usa un nombre MS-DOS secundario.

Luego viene la información de seguridad. En versiones hasta NT 4.0 inclusive, el campo de seguridad contenía el descriptor de seguridad real. A partir de NT 5.0 toda la información de seguridad se centralizó en un solo archivo, y el campo de seguridad simplemente apunta a la parte pertinente de ese archivo.

En el caso de archivos pequeños, los datos del archivo mismos están contenidos en la entrada de MFT, lo que ahorra un acceso a disco para obtenerla. Esta idea se llama **archivo inmediato** (Mullender y Tanenbaum, 1987). En el caso de archivos un poco más grandes, este campo contiene apuntadores a los cúmulos que contienen los datos o, lo que es más

común, a series de cúmulos consecutivos, por lo que un solo número de cúmulo y una longitud pueden representar una cantidad arbitraria de datos del archivo. Si una sola entrada de MFT no basta para contener toda la información que se supone debe contener, se pueden encadenar a ella una o más entradas adicionales.

El tamaño de archivo máximo es de 2^{64} bytes. Para tener una idea de cuán grande es un archivo de 2^{64} bytes, imagine que el archivo se escribe en binario y cada 0 o 1 ocupa 1 mm de espacio. El listado de 2^{67} mm tendría 15 años luz de longitud, y podría salir del Sistema Solar, llegar a Alpha Centauri, y regresar.

El sistema de archivos NTFS tiene muchas otras propiedades interesantes que incluyen compresión de datos y tolerancia de fallos empleando transacciones atómicas. Se puede encontrar información adicional al respecto en (Solomon, 1998).

6.4.4 Ejemplos de gestión de procesos

Tanto UNIX como NT permiten dividir un trabajo en varios procesos que se pueden ejecutar en (pseudo)paralelo y comunicarse entre sí, como en el ejemplo del productor y el consumidor que vimos antes. En esta sección explicaremos cómo se controlan los procesos en ambos sistemas. Además, los dos sistemas apoyan el paralelismo dentro de un solo procesador empleando enlaces, así que también hablaremos de esto.

Gestión de procesos en UNIX

En cualquier momento, un proceso UNIX puede crear un subproceso que es una réplica exacta de sí mismo, ejecutando la llamada de sistema denominada fork. El proceso original se llama **padre** y el nuevo se llama **hijo**. Inmediatamente después de la fork, los dos procesos son idénticos e incluso comparten los mismos descriptores de archivo. En adelante, cada uno sigue su propio camino y se comporta con independencia del otro.

En muchos casos, el proceso hijo manipula los descriptores de archivo de ciertas maneras y luego ejecuta la llamada de sistema exec, que sustituye su programa y datos por los programas y datos que se encuentran en un archivo ejecutable que se especifica como parámetro de la llamada exec. Por ejemplo, cuando un usuario teclea un comando *xyz* en una terminal, el intérprete de comandos (*shell*) ejecuta fork para crear un proceso hijo. Luego, este proceso hijo emite exec para ejecutar el programa *xyz*.

Los dos procesos se ejecutan en paralelo (con o sin exec), a menos que el padre desee esperar a que el hijo termine antes de continuar. En tal caso, el padre emite una de dos llamadas de sistema, wait o waitpid, que hacen que el padre se suspenda hasta que el hijo termine emitiendo exit. Una vez que el hijo termina, el padre continúa.

Los procesos pueden ejecutar fork cuantas veces quieran, y esto da pie a un árbol de procesos. En la figura 6-43, por ejemplo, el proceso *A* emitió fork dos veces y creó dos hijos, *B* y *C*. Luego *B* emitió fork dos veces, y *C* lo hizo una vez, para dar el árbol final de seis procesos.

Los procesos en UNIX pueden comunicarse entre ellos mediante una estructura llamada **fila** (*pipe*). Una fila es una especie de buffer en el que un proceso puede escribir un flujo de

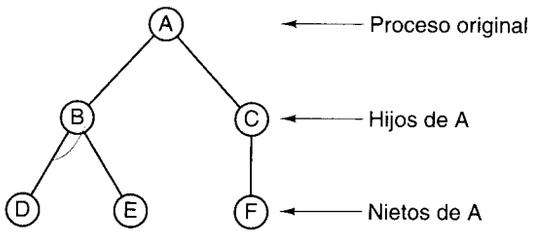


Figura 6-43. Árbol de procesos en UNIX.

datos y otro puede sacar el flujo de él. Los bytes siempre se recuperan de una fila en el orden en que se escribieron; no puede haber acceso aleatorio. Las filas no preservan las fronteras de los mensajes, de modo que si un proceso efectúa cuatro escrituras de 128 bytes y el otro efectúa una lectura de 512 bytes, el lector obtendrá todos los datos juntos, sin indicio de que se escribieron en varias operaciones.

En System V y Solaris, otro mecanismo de comunicación entre procesos es el uso de **colas de mensajes**. Un proceso puede crear una cola de mensajes o abrir una cola ya existente usando msgget. Mediante una cola de mensajes, un proceso puede enviar mensajes con msgsnd y recibirlos con msgrecv. Los mensajes así enviados difieren de los datos dentro de una fila en varios sentidos. Primero, las fronteras de los mensajes se preservan, mientras que una fila no es más que un flujo de datos. Segundo, los mensajes tienen prioridades, de modo que los urgentes pueden adelantarse a otros menos importantes. Tercero, los mensajes tienen tipos, y un msgrecv puede especificar un tipo dado si lo desea.

Otro mecanismo de comunicación es la capacidad de dos o más procesos para compartir una región de sus espacios de direcciones respectivos. UNIX maneja esta memoria compartida haciendo corresponder las mismas páginas con el espacio de direcciones virtual de todos los procesos que las comparten. El resultado es que una escritura efectuada por un proceso en la región compartida es visible de inmediato para los demás procesos. Este mecanismo ofrece un camino de comunicación con gran ancho de banda entre los procesos. Las llamadas de sistema que se usan para manejar memoria compartida tienen nombres como shmat y shmop.

Otra característica de System V y Solaris es la disponibilidad de semáforos. Éstos funcionan básicamente como se describió en el ejemplo del productor-consumidor que se dio en el texto.

Otro recurso que proporcionan todos los sistemas UNIX que cumplen con POSIX es la capacidad para tener varios enlaces de control dentro de un solo proceso. Estos enlaces de control, generalmente llamados sólo **enlaces** (*threads*) son como procesos ligeros que comparten un mismo espacio de direcciones y todo lo asociado con dicho espacio, como descriptores de archivo, variables de entorno y temporizadores pendientes. Sin embargo, cada enlace tiene su propio contador de programa, sus propios registros y su propia pila. Cuando un enlace se bloquea (es decir, se detiene temporalmente hasta que termina una operación de E/S u ocurre algún otro suceso), otros enlaces del mismo proceso pueden seguir ejecutándose. Dos enlaces del mismo proceso que operan como productor y consumidor son similares, pero no idénticos, a dos procesos de un solo enlace cada uno que comparten un segmento de

memoria que contiene un buffer. Las diferencias tienen que ver con el hecho de que en el segundo caso cada proceso tiene sus propios descriptores de archivo, etc., mientras que en el primero todas esas cosas son compartidas. Ya vimos el uso de enlaces Java en nuestro ejemplo del productor-consumidor. Es común que el sistema de tiempo de ejecución de Java use un enlace del sistema operativo para cada uno de sus enlaces, pero esto no es obligatorio.

Como ejemplo de caso en el que los enlaces podrían ser útiles, considere un servidor de la World Wide Web. Un servidor así podría mantener una caché de páginas Web de uso común en la memoria principal. Si una solicitud hace referencia a una página que está en la caché, la página Web se devuelve de inmediato; si no, se trae del disco. Lo malo es que la espera mientras llega la página del disco es larga (digamos 20 ms), y durante este tiempo el proceso está bloqueado y no puede atender ninguna solicitud entrante, ni siquiera las que piden páginas Web que están en la caché.

La solución es tener varios enlaces dentro del proceso servidor, todos los cuales comparten la caché común de páginas Web. Cuando un enlace se bloquea, otros enlaces pueden atender solicitudes nuevas. Para evitar el bloqueo sin enlaces, podríamos tener varios procesos servidores, pero es probable que ello implicaría duplicar la caché, lo que desperdiciaría memoria valiosa.

El estándar UNIX para enlaces se llama **pthreads** y está definido por POSIX (P1003.1C). El estándar contiene llamadas para gestionar y sincronizar enlaces. No está definido si los enlaces están bajo el control del núcleo o totalmente en el espacio de usuario. Las llamadas para enlaces de uso más común se enumeran en la figura 6-44.

Llamada para enlaces	Significado
pthread_create	Crear un enlace nuevo en el espacio de direcciones del invocador
pthread_exit	Terminar el enlace invocado
pthread_join	Esperar que un enlace termine
pthread_mutex_init	Crear un mutex nuevo
pthread_mutex_destroy	Destruir un mutex
pthread_mutex_lock	Cerrar un mutex
pthread_mutex_unlock	Abrir un mutex
pthread_cond_init	Crear una variable de condición
pthread_cond_destroy	Destruir una variable de condición
pthread_cond_wait	Esperar por una variable de condición
pthread_cond_signal	Liberar un enlace que espera por una variable de condición

Figura 6-44. Principales llamadas de POSIX para enlaces.

Examinemos brevemente las llamadas para enlaces que se muestran en la figura 6-44. La primera llamada, pthread_create, crea un enlace nuevo. Si la llamada se lleva a cabo con éxito, se estará ejecutando un enlace más en el espacio de direcciones que los que se estaban ejecutando antes de la llamada. Un enlace que terminó su trabajo y quiere terminar invoca pthread_exit. Un enlace puede esperar que otro enlace termine invocando pthread_join. Si el enlace que se desea esperar ya terminó, pthread_join termina de inmediato; de lo contrario, se bloquea.

Los enlaces pueden sincronizarse empleando candados llamados **mutexes**. Por lo regular, un mutex protege a algún recurso, como un buffer compartido por dos enlaces. Para asegurarse de que sólo un enlace a la vez accese al recurso compartido, se espera que los enlaces cierren el mutex antes de tocar el recurso y lo abran cuando hayan terminado. En tanto todos los enlaces obedezcan este protocolo, podrán evitarse condiciones de competencia. Los mutexes son como semáforos binarios, es decir, semáforos que sólo pueden adoptar los valores 0 y 1. El nombre "mutex" proviene de su uso para asegurar la mutua exclusión en el uso de algún recurso.

Los mutexes se pueden crear y destruir con las llamadas pthread_mutex_init y pthread_mutex_destroy, respectivamente. Un mutex puede estar en uno de dos estados: cerrado o abierto. Cuando un enlace trata de cerrar un mutex abierto (usando pthread_mutex_lock), el candado se cierra y el enlace continúa. En cambio, si un enlace trata de cerrar un mutex que ya está cerrado, se bloquea. Cuando el enlace que cerró el mutex termina de usar el recurso compartido, se espera que abra el mutex correspondiente invocando pthread_mutex_unlock.

Los mutexes se diseñaron para bloqueo a corto plazo, como la protección de una variable compartida, no para una sincronización a largo plazo, como esperar que una unidad de cinta se desocupe. Para este fin se cuenta con **variables de condición**, las cuales se crean y se destruyen con llamadas a pthread_cond_init y pthread_cond_destroy, respectivamente.

Usamos una variable de condición haciendo que un enlace la espere y que otro la señalice. Por ejemplo, si descubre que la unidad de cinta que necesita está ocupada, un enlace invocaría pthread_cond_wait con una variable de condición que todos los enlaces han acordado asociar a la unidad de cinta. Cuando el enlace que está usando la unidad de cinta ya no la necesita (tal vez horas después), usa pthread_cond_signal para liberar exactamente un enlace de los que están esperando esa variable de condición (si los hay). Si ningún enlace está esperando, la variable se pierde. Las variables de condición no cuentan como los semáforos. También se definen otras operaciones con enlaces, mutexes y variables de condición.

Gestión de procesos en Windows NT

NT maneja múltiples procesos, que pueden comunicarse y sincronizarse. Cada proceso contiene por lo menos un enlace, que a su vez contiene al menos una fibra (enlace ligero). Juntos, los procesos, enlaces y fibras proveen un conjunto muy general de herramientas para gestionar paralelismo, tanto en uniprocesadores (máquinas con una sola CPU) como en multiprocesadores (máquinas con varias CPU).

Se crean procesos nuevos empleando la función API CreateProcess. Esta función tiene 10 parámetros, cada uno de los cuales tiene muchas opciones. Este diseño obviamente es mucho más complicado que el de UNIX, en el que fork no tiene parámetros, y exec sólo tiene tres: apuntadores al nombre del archivo que se ejecutará, el arreglo de parámetros de la línea de comandos (analizados sintácticamente) y las cadenas de entorno. A grandes rasgos, los 10 parámetros de CreateProcess son los siguientes:

1. Un apuntador al nombre del archivo ejecutable.
2. La línea de comandos (sin analizar sintácticamente).
3. Un apuntador a un descriptor de seguridad para el proceso.
4. Un apuntador a un descriptor de seguridad para el enlace inicial.
5. Un bit que indica si el nuevo proceso hereda o no las manijas de su creador.
6. Diversas banderas (por ejemplo, modo de error, prioridad, depuración, consolas).
7. Un apuntador a las cadenas de entorno.
8. Un apuntador al nombre del directorio de trabajo vigente del nuevo proceso.
9. Una apuntador a una estructura que describe la ventana inicial en la pantalla.
10. Un apuntador a una estructura que devuelve 18 valores al invocador.

NT no obliga a establecer una jerarquía padre-hijo ni de ningún otro tipo. Todos los procesos se crean iguales. Sin embargo, dado que uno de los 18 parámetros que se devuelven al proceso creador es un mango para el proceso nuevo (que permite tener un control considerable sobre este último), existe una jerarquía implícita en términos de quién tiene una manija para quién. Aunque estas manijas no pueden pasarse directamente a otros procesos, existe una forma en que un proceso puede hacer una manija apropiada para otro proceso y luego dársela, así que la jerarquía de procesos implícita podría no durar mucho.

Cada proceso NT se crea con un solo enlace, pero un proceso puede crear más enlaces posteriormente. La creación de enlaces es más sencilla que la de procesos: CreateThread sólo tiene seis parámetros en lugar de 10: el descriptor de seguridad, el tamaño de pila, la dirección inicial, un parámetro definido por el usuario, el estado inicial del enlace (listo o bloqueado) y el identificador del enlace. El núcleo se encarga de crear los enlaces, por los que tiene un conocimiento claro de ellos (es decir, no se implementan únicamente en el espacio de usuario, como en algunos otros sistemas).

Cuando el núcleo efectúa la planificación, no sólo escoge el proceso que se ejecutará a continuación, sino también el enlace dentro de ese proceso. Esto implica que el núcleo siempre sabe cuáles enlaces están listos y cuáles están bloqueados. Puesto que los enlaces son objetos del núcleo, tienen descriptores de seguridad y manijas. Dado que una manija para un enlace se puede pasar a otro proceso, es posible hacer que un proceso controle los enlaces de un proceso distinto. Esta característica es útil para los depuradores, por ejemplo.

Los enlaces NT son relativamente costosos porque la conmutación de un enlace requiere ingresar en el núcleo y después salir de él. A fin de ofrecer un seudoparalelismo muy ligero, NT cuenta con **fibras**, que son parecidas a los enlaces sólo que son planificados en el espacio de usuario por el programa que las crea (o su sistema de tiempo de ejecución). Cada enlace puede tener varias fibras, así como un proceso puede tener varios enlaces, excepto que cuando una fibra se bloquea lógicamente se coloca a sí misma en la cola de fibras bloqueadas y selecciona otra fibra que se ejecutará en el contexto de su enlace. El núcleo no tiene conocimiento de esta transición porque el enlace se sigue ejecutando, aunque primero podría estar

ejecutando una fibra y después otra. El núcleo sólo administra procesos e enlaces, no fibras. Las fibras son útiles, por ejemplo, cuando programas que administran sus propios enlaces se trasladan a NT.

Los procesos pueden comunicarse de muy diversas maneras, incluidas filas, filas con nombre, buzones, *sockets*, llamadas a procedimientos remotos y archivos compartidos. Las filas tienen dos modos, de bytes y de mensajes, que se seleccionan en el momento en que se crean. Las filas en modo de bytes funcionan igual que en UNIX. Las filas en modo de mensajes son parecidos pero preservan las fronteras de los mensajes, de modo que cuatro escrituras de 128 bytes se leerán como cuatro mensajes de 128 bytes, no un mensaje de 512 bytes, como ocurriría con filas en modo de bytes. También existen filas con nombre y tienen los mismos dos modos que las filas normales. Las filas con nombres también pueden usarse en una red, cosa que no es posible con los normales.

Los **buzones** (*mailslots*) son una característica de NT que no está presente en UNIX. Los buzones son parecidos a las filas en ciertos aspectos, pero no en todos. En primer lugar, son unidireccionales, mientras que las filas son bidireccionales. Los buzones también pueden usarse en una red pero no garantizan la entrega. Por último, los buzones permiten al proceso transmisor difundir un mensaje a varios receptores, no sólo a uno.

Los *sockets* son como filas, excepto que normalmente conectan procesos de diferentes máquinas; sin embargo, también pueden servir para conectar procesos en la misma máquina. En general, casi nunca representa una ventaja usar una conexión de *socket* en lugar de una fila o fila con nombre para la comunicación dentro de la máquina.

Las llamadas a procedimientos remotos permiten al proceso *A* pedir al proceso *B* que invoque un procedimiento en el espacio de direcciones de *B* a nombre de *A*, y que devuelva el resultado a *A*. Existen diversas restricciones de los parámetros. Por ejemplo, no tiene sentido pasar un apuntador a un proceso distinto.

Por último, los procesos pueden compartir memoria estableciendo una correspondencia con el mismo archivo al mismo tiempo. Así, todas las escrituras efectuadas por un proceso aparecen en los espacios de direcciones de los otros procesos. Con este mecanismo es fácil implementar el buffer compartido que usamos en nuestro ejemplo de productor-consumidor.

Así como NT ofrece varios mecanismos de comunicación entre procesos, también provee numerosos mecanismos de sincronización, incluidos semáforos, mutexes, secciones críticas y eventos. Todos estos mecanismos funcionan con enlaces, no con procesos, de modo que cuando un enlace se bloquea en un semáforo los demás enlaces de ese proceso (si los hay) no resultan afectados y pueden continuar su ejecución.

Se crea un semáforo con la función API CreateSemaphore, que puede asignarle un valor inicial dado y definir también un valor máximo. Los semáforos son objetos del núcleo y por ende tienen descriptores de seguridad y manijas. La manija de un semáforo puede duplicarse con DuplicateHandle y pasarse a otro proceso para que varios procesos puedan sincronizarse con el mismo semáforo. Hay llamadas para up y down, aunque tienen los nombres un tanto extraños de ReleaseSemaphore (up) y WaitForSingleObject (down). También es posible asignar a WaitForSingleObject un plazo de tiempo que permita liberar el enlace invocador tarde o temprano, aunque el semáforo siga siendo 0 (aunque los temporizadores vuelven a introducir competencias).

Los mutexes también son objetos del núcleo que se usan para sincronización, pero son más sencillos que los semáforos porque no tienen contadores. Los mutexes son esencialmente candados, con funciones API para cerrarse (WaitForSingleObject) y abrirse ReleaseMutex. Al igual que las manijas de semáforos, las manijas de mutexes pueden duplicarse y pasarse de un proceso a otro para que enlaces de diferentes procesos puedan accesar al mismo mutex.

El tercer mecanismo de sincronización se basa en **secciones críticas**, que son similares a los mutexes sólo que son locales respecto al espacio de direcciones del enlace que las crea. Puesto que las secciones críticas no son objetos del núcleo, no tienen manijas ni descriptores de seguridad y no pueden transferirse entre procesos. El cierre y la apertura se efectúan con EnterCriticalSection y LeaveCriticalSection, respectivamente. Puesto que estas funciones API se efectúan totalmente en el espacio de usuario, son mucho más rápidas que los mutexes.

El último mecanismo de sincronización emplea objetos del núcleo llamados **eventos**. Un enlace puede esperar que ocurra un evento con WaitForSingleObject. Un enlace puede liberar un enlace individual que esté esperando un evento con SetEvent o bien liberar todos los enlaces que esperan un evento, con PulseEvent. Hay varios tipos de eventos y además tienen diversas opciones.

Los eventos, mutexes y semáforos pueden recibir nombres y guardarse en el sistema de archivos como los tubos con nombre. Dos o más procesos pueden sincronizarse abriendo el mismo evento, mutex o semáforo, en lugar de hacer que uno de ellos cree el objeto y luego produzca copias del mango para los demás, aunque ésta no deja de ser una opción.

6.5 RESUMEN

El sistema operativo puede verse como un intérprete de ciertas características arquitectónicas que no se encuentran en el nivel ISA. Las principales de ellas son la memoria virtual, las instrucciones de E/S virtuales y los recursos para procesamiento en paralelo.

La memoria virtual es una característica arquitectónica cuyo propósito es permitir a los programas usar un espacio de direcciones mayor que la memoria física de la máquina, o proporcionar un mecanismo coherente y flexible para proteger y compartir la memoria. La memoria virtual se puede implementar como paginación pura, segmentación pura y una combinación de las dos. En la paginación pura el espacio de direcciones se divide en páginas virtuales de tamaño uniforme. Algunas de éstas se hacen corresponder con marcos de página físicos, pero otras no tienen correspondencia. La MMU traduce una referencia a una página con correspondencia en la dirección física correcta. Una referencia a una página sin correspondencia causa un fallo de página. Tanto el Pentium II como el UltraSPARC II tienen unas MMU avanzadas que manejan tanto memoria virtual como paginación.

La abstracción de E/S más importante en este nivel es el archivo. Un archivo consiste en una secuencia de bytes o registros lógicos que se pueden leer y escribir sin saber cómo funcionan los discos, cintas y otros dispositivos de E/S. El acceso a los archivos puede ser secuencial, aleatorio por número de registro o aleatorio por clave. Los directorios sirven para agrupar archivos. Los archivos se pueden almacenar en sectores consecutivos o dispersarse por todo el disco. En el segundo caso, normal en los discos duros, se requieren estructuras de

datos para localizar todos los bloques de un archivo. El espacio de disco libre puede contabilizarse con una lista o un mapa de bits.

Es común que se apoye el procesamiento en paralelo y se implemente simulando múltiples procesadores operando una sola CPU en tiempo compartido. La interacción no controlada entre procesos puede dar pie a condiciones de competencia. Para resolver este problema se introducen primitivas de sincronización, de las cuales los semáforos son un ejemplo sencillo. Con la ayuda de semáforos, los problemas de productor-consumidor se pueden resolver de forma sencilla y elegante.

Dos ejemplos de sistemas operativos avanzados son UNIX y NT. Ambos apoyan la paginación y archivos con correspondencia en la memoria, así como sistemas de archivos jerárquicos, con archivos que consisten en secuencias de bytes. Por último, ambos manejan procesos y enlaces y proporcionan formas de sincronizarlos.

PROBLEMAS

1. ¿Por qué un sistema operativo interpreta sólo algunas de las instrucciones de nivel 3, mientras que un microprograma interpreta todas las instrucciones de nivel ISA?
2. Una máquina tiene un espacio de direcciones virtual de 32 bits direccionable por bytes. El tamaño de página es de 8 KB. ¿Cuántas páginas de espacio de direcciones virtual existen?
3. Una memoria virtual tiene un tamaño de página de 1024 palabras, ocho páginas virtuales y cuatro marcos de página físicos. La tabla de páginas tiene este aspecto:

Página virtual	Marco de página
0	3
1	1
2	no en la memoria principal
3	no en la memoria principal
4	4
5	no en la memoria principal
6	0
7	no en la memoria principal

   a. Haga una lista de todas las direcciones virtuales que causarán fallos de página.
   b. ¿Qué dirección física tienen 0, 3728, 1023, 1024, 1025, 7800 y 4096?

4. Una computadora tiene 16 páginas de espacio de direcciones virtual pero sólo cuatro marcos de página. Inicialmente, la memoria está vacía. Un programa hace referencia a las páginas virtuales en el orden

   0, 7, 2, 7, 5, 8, 9, 2, 4

   a. ¿Cuáles referencias causan un fallo de página con LRU?
   b. ¿Cuáles referencias causan un fallo de página con FIFO?

5. En la sección 6.1.3. se presentó un algoritmo para implementar una estrategia de reemplazo de páginas FIFO. Idee uno más eficiente. *Sugerencia:* Es posible actualizar el contador en la página recién cargada y no modificar los otros.

6. En los sistemas paginados que describimos en el texto el manejador de fallos de página formaba parte del nivel ISA y por tanto no estaba presente en el espacio de direcciones de ningún programa del nivel OSM. En realidad, el manejador de fallos de página también ocupa páginas, y en ciertas circunstancias (por ejemplo, política de reemplazo de páginas FIFO) podría ser sacado de la memoria. ¿Qué sucedería si el manejador de fallos de página no estuviera presente al ocurrir un fallo de página? ¿Cómo podría corregirse esto?

7. No todas las computadoras tienen un bit en hardware que se enciende automáticamente cuando se escribe en una página. No obstante, resulta útil de seguir la pista a las páginas modificadas, para no tener que suponer el peor de los casos y escribir todas las páginas de vuelta al disco después de usarlas. Suponiendo que cada página tiene bits en hardware que de forma independiente habilitan el acceso para lectura, escritura y ejecución, ¿cómo puede el sistema operativo saber cuáles páginas están limpias y cuáles están sucias?

8. Una memoria segmentada tiene segmentos paginados. Cada dirección virtual tiene un número de segmento de 2 bits, un número de página de 2 bits y una distancia dentro de la página de 11 bits. La memoria principal contiene 32 KB, divididos en páginas de 2 KB. Cada segmento es sólo de lectura, de lectura/ejecución, de lectura/escritura, o de lectura/escritura/ejecución. Las tablas de páginas y de protección tienen este aspecto:

Segmento 0		Segmento 1		Segmento 2	Segmento 3	
Sólo lectura		Leer/ejecutar		Leer/escribir/ejecutar	Leer/escribir	
Página virtual	Marco de página	Página virtual	Marco de página		Página virtual	Marco de página
0	9	0	En disco	Tabla de páginas	0	14
1	3	1	0	no en la	1	1
2	En disco	2	15	memoria	2	6
3	12	3	8	principal	3	En disco

Para cada uno de los siguientes accesos a la memoria virtual, indique qué dirección física se calcula. Si ocurre un fallo, indique de qué tipo es.

Acceso	*Segmento*	*Página*	*Distancia dentro de la página*
1. buscar datos	0	1	1
2. buscar datos	1	1	10
3. buscar datos	3	3	2047
4. guardar datos	0	1	4
5. guardar datos	3	1	2
6. guardar datos	3	0	14
7. saltar a ellos	1	3	100
8. buscar datos	0	2	50
9. buscar datos	2	0	5
10. saltar a ellos	3	0	60

9. Algunas computadoras permiten E/S directamente al espacio de usuario. Por ejemplo, un programa podría iniciar una transferencia de disco a un buffer dentro de un proceso de usuario. ¿Causa esto problemas si se usa compactación para implementar la memoria virtual? Comente.

10. Los sistemas operativos que permiten archivos con correspondencia en la memoria siempre requieren que los archivos se mapeen en fronteras de página. Por ejemplo, con páginas de 4K, un archivo puede mapearse de modo que comience en la dirección virtual 4096, pero no en la dirección virtual 5000. ¿Por qué?

11. Cuando un registro de segmento se carga con un valor en el Pentium II, se trae el descriptor correspondiente y se carga en una parte invisible del registro de segmento. ¿Por qué cree que los diseñadores de Intel hayan decidido hacer esto?

12. Un programa en el Pentium II hace referencia al segmento local 10 con distancia 8000. El campo BASE del segmento LDT 10 contiene 10000. ¿Cuál entrada de directorio usa el Pentium II? ¿Cuál es el número de página? ¿Cuál es la distancia?

13. Comente algunos posibles algoritmos para eliminar segmentos en una memoria segmentada no paginada.

14. Compare la fragmentación interna con la externa. ¿Qué puede hacerse para aliviar cada una?

15. Los supermercados enfrentan constantemente un problema similar al reemplazo de páginas en los sistemas con memoria virtual. Hay una cantidad fija de espacio de anaquel para exhibir un número cada vez mayor de productos. Si llega un producto nuevo importante, digamos alimento para perros con eficiencia del 100%, algún producto existente tendrá que sacarse del inventario a fin de hacer un hueco para aquél. Los algoritmos de reemplazo obvios son LRU y FIFO. ¿Cuál de éstos preferiría usted?

16. ¿Por qué los bloques de caché siempre son mucho más pequeños que las páginas de memoria virtual, a menudo cien veces más pequeños?

17. ¿Por qué muchos sistemas de archivos exigen que un archivo se abra explícitamente con una llamada al sistema open antes de leerse?

18. Compare los métodos de mapa de bits y lista de huecos para contabilizar el espacio libre de un disco que tiene 800 cilindros, cada uno de los cuales tiene 5 pistas de 32 sectores. ¿Cuántos huecos tendría que haber para que la lista de huecos fuera más grande que el mapa de bits? Suponga que la unidad de asignación es el sector y que un hueco requiere una entrada de tabla de 32 bits.

19. Si se quiere predecir el desempeño de un disco, resulta útil tener un modelo de asignación del espacio. Suponga que el disco se ve como un espacio de direcciones lineal de $N >> 1$ sectores, que consiste en una serie de bloques de datos, luego un hueco, luego otra serie de bloques de datos, y así. Si mediciones empíricas muestran que las distribuciones de probabilidad para las longitudes de las series de datos y los huecos son iguales, y la probabilidad de que cualquiera de ellos sea de i sectores es de 2^{-i}, ¿cuál será el número esperado de huecos en el disco?

20. En cierta computadora un programa puede crear tantos archivos como necesite, y todos los archivos pueden crecer dinámicamente durante la ejecución sin proporcionar al sistema operativo información por adelantado acerca de su tamaño final. ¿Cree que los archivos se almacenarán en sectores consecutivos? Explique.

21. Considere el siguiente método con el que un sistema operativo podría implementar instrucciones de semáforos. Cada vez que la CPU está a punto de ejecutar up o down con un semáforo (una variable entera en la memoria), primero ajusta los bits de prioridad o máscara de la CPU de modo que se inhabiliten todas las interrupciones. Luego la CPU trae el semáforo, lo modifica y toma la rama correcta. Por último, la CPU vuelve a habilitar las interrupciones. ¿Este método funciona si

a. hay una sola CPU que conmuta entre procesos cada 100 ms?
b. dos CPU comparten una memoria común en la que se encuentra el semáforo?

22. La Compañía de Sistemas Operativos Quenuncasecaen ha estado recibiendo quejas de algunos de sus clientes relacionadas con su más reciente versión, que incluye operaciones con semáforos. Los clientes creen que es inmoral que los procesos se bloqueen (lo llaman "dormirse en el trabajo"). Dado que es política de la compañía dar a los clientes lo que piden, se ha propuesto añadir una tercera operación, peek, para complementar up y down. Lo que hace peek es examinar el semáforo sin modificarlo ni bloquear el proceso. De este modo, los programas que sientan que es inmoral bloquearse pueden examinar primero el semáforo para ver si pueden ejecutar down sin peligro. ¿Funcionará esta idea si tres o más procesos usan el semáforo? ¿Y si dos procesos usan el semáforo?

23. Prepare una tabla que muestre cuáles de los procesos P1, P2 Y P3 se están ejecutando y cuáles están bloqueados, en función del tiempo desde 0 hasta 1000 ms. Los tres procesos ejecutan instrucciones up y down con el mismo semáforo. Cuando dos procesos están bloqueados y se ejecuta una up, se reinicia el proceso que tiene el número más bajo, es decir, P1 tiene preferencia sobre P2 y P3, etc. Inicialmente, los tres procesos están en ejecución y el semáforo es 1.

En t = 100 P1 ejecuta down
En t = 200 P1 ejecuta down
En t = 300 P2 ejecuta up
En t = 400 P3 ejecuta down
En t = 500 P1 ejecuta down
En t = 600 P2 ejecuta up
En t = 700 P2 ejecuta down
En t = 800 P1 ejecuta up
En t = 900 P1 ejecuta up

24. En un sistema de reservaciones de líneas aéreas es necesario asegurar que mientras un proceso está ocupado usando un archivo, ningún otro proceso puede usarlo también. De lo contrario, dos procesos distintos, trabajando para dos agentes de viajes distintos, podrían sin darse cuenta vender el último asiento de algún vuelo. Idee un método de sincronización empleando semáforos que asegure que sólo un proceso a la vez accederá a cada archivo (suponiendo que los procesos obedecen las reglas).

25. Para poder implementar semáforos en una computadora con varias CPU que comparten una memoria común, los arquitectos de computadoras a menudo incluyen una instrucción de Probar y Poner Candado (*Test and Set Lock*) TSL X prueba la posición X. Si el contenido es 0, se pone en 1 en un solo ciclo de memoria indivisible, y se pasa por alto la siguiente instrucción. Si el contenido no es 0, la TSL actúa como nop (ninguna operación). Con TSL es posible escribir procedimientos *lock* y *unlock* con las siguientes propiedades. *lock* (x) verifica si x tienen candado; si no, pone un candado a x y devuelve el control; si x ya tiene un candado, el procedimiento espera hasta que se abre el candado y entonces pone un candado a x y devuelve el control. *unlock* abre un candado existente. Si todos los procesos ponen un candado a la tabla de semáforos antes de usarla, sólo un proceso a la vez podrá meterse con las variables y apuntadores, y se evitarán las competencias. Escriba *lock* y *unlock* en lenguaje ensamblador. (Haga cualquier supuesto razonable que necesite.)

26. Indique los valores de *in* y *out* para un buffer circular de 65 palabras de largo después de cada una de las operaciones siguientes. Ambas variables valen inicialmente 0.

a. Se meten 22 palabras
b. Se sacan 9 palabras
c. Se meten 40 palabras
d. Se sacan 17 palabras
e. Se meten 12 palabras
f. Se sacan 45 palabras
g. Se meten 8 palabras
h. Se sacan 11 palabras

27. Suponga que una versión de UNIX usa bloques de disco de 2K y guarda 512 direcciones de disco en cada bloque indirecto (sencillo, doble y triple). ¿Qué tamaño máximo podría tener un archivo? (Suponga que los apuntadores de archivo son de 64 bits.)

28. Suponga que la llamada al sistema UNIX

```
unlink("/usr/ast/bin/game3")
```

se ejecuta en el contexto de la figura 6-37. Describa cuidadosamente los cambios que se efectúan en el sistema de directorios.

29. Imagine que tiene que implementar el sistema UNIX en una microcomputadora con muy escasa memoria principal. Por más que trata de meter a la fuerza el sistema en el espacio disponible, no logra hacer que quepa, así que escoge una llamada al sistema al azar, la cual sacrificará por el bien general. Se escogió pipe, que crea los tubos empleados para enviar flujos de bytes de un proceso a otro. ¿Sigue siendo posible implementar la redirección de E/S de alguna manera? ¿Y las filas de procesamiento? Comente los problemas y posibles soluciones.

30. El Comité para el Trato Justo de los Descriptores de Archivo está organizando una protesta contra el sistema UNIX porque cada vez que éste devuelve un descriptor de archivo, siempre devuelve el número más bajo que no se está usando actualmente. Por consiguiente, los descriptores de archivo con números altos pocas veces se usan. Su plan es devolver el número más bajo que el programa todavía no ha usado, en lugar del número más bajo que no se está usando actualmente. Se asegura que la implementación es trivial, que no afectará a los programas existentes, y que es más justo. ¿Qué piensa usted?

31. En NT es posible configurar una lista de control de acceso de tal manera que Roberta no tenga ningún acceso a un archivo, pero todos los demás tengan pleno acceso a él. ¿Cómo cree que se implemente esto?

32. Describa dos formas de programar problemas de productor-consumidor empleando buffers compartidos y semáforos en NT. Piense en cómo implementar el buffer compartido en cada caso.

33. Es común probar los algoritmos de reemplazo de páginas por simulación. Para este ejercicio, escriba un simulador para una memoria virtual basada en páginas en una máquina con 64 páginas de 1 KB. El simulador debe mantener una sola tabla de 64 entradas, una por página, que contengan el número de página física que corresponde a esa página virtual. El simulador deberá leer un archivo que contiene direcciones virtuales en decimal, una dirección por línea. Si la página correspondiente está en la memoria, se registra un acierto de página; si no, se invoca un procedimiento de

reemplazo de páginas que escoja una página para expulsarla (es decir, una entrada de la tabla para sobreescribirla) y se registra un fallo de página. No ocurre realmente transporte de páginas. Genere un archivo que consista en direcciones aleatorias y pruebe el desempeño con LRU y con FIFO. Ahora genere un archivo de direcciones en el que x% de las direcciones sean cuatro bytes más altas que la dirección anterior (a fin de simular localidad). Realice pruebas con diversos valores de x e informe sus resultados.

34. Escriba un programa para UNIX o NT que acepte como entrada el nombre de un directorio. El programa deberá imprimir una lista de los archivos del directorio, una línea por archivo, y después del nombre del archivo imprimir el tamaño del archivo. Imprima los nombres de archivo en el orden en que aparezcan en el directorio. Las ranuras no utilizadas del directorio deberán indicarse con esta expresión: (no utilizado).

7

EL NIVEL DE LENGUAJE ENSAMBLADOR

En los capítulos 4, 5 y 6 examinamos tres niveles que están presentes en casi todas las computadoras contemporáneas. Este capítulo se ocupa primordialmente de otro nivel que también está presente en prácticamente todas las computadoras modernas: el nivel de lenguaje ensamblador. El nivel de lenguaje ensamblador difiere en un aspecto importante de los niveles de microarquitectura, ISA y sistema operativo: se implementa por traducción en lugar de interpretación.

Los programas que convierten un programa de usuario escrito en algún lenguaje, a otro lenguaje, se llaman **traductores**. El lenguaje en el que está escrito el programa original se llama **lenguaje fuente**, y el lenguaje al que se convierte se llama **lenguaje objetivo**. Ambos lenguajes definen niveles. Si se cuenta con un procesador capaz de ejecutar directamente programas escritos en el lenguaje fuente, no hay necesidad de traducir el programa fuente al lenguaje objetivo.

Se usa traducción cuando se cuenta con un procesador (sea hardware o un intérprete) para el lenguaje objetivo pero no para el lenguaje fuente. Si la traducción se efectuó correctamente, la ejecución del programa traducido dará exactamente los mismos resultados que habría dado la ejecución del programa fuente si se hubiera contado con un procesador para él. Por tanto, es posible implementar un nuevo nivel para el que no hay un procesador traduciendo primero los programas escritos para ese nivel a un nivel objetivo, y ejecutando después los programas resultantes en el nivel objetivo.

Es importante señalar la diferencia entre traducción e interpretación. En la traducción no se ejecuta directamente el programa original en el lenguaje fuente; en vez de ello, se convier-

te en un programa equivalente llamado **programa objeto** o **programa binario ejecutable** que se ejecuta sólo después de que se ha llevado a cabo la traducción. En la traducción hay dos pasos bien definidos:

1. Generación de un programa equivalente en el lenguaje objetivo.
2. Ejecución del programa recién generado.

Estos dos pasos no ocurren simultáneamente. El segundo paso no se inicia hasta que el primero ha finalizado. En la interpretación sólo hay un paso: ejecutar el programa fuente original. No es necesario generar primero un programa equivalente, aunque a veces el programa fuente se convierte en una forma intermedia (por ejemplo, código de bytes Java) para facilitar la interpretación.

Mientras el programa objeto se está ejecutando, sólo son evidentes tres niveles: el nivel de microarquitectura, el nivel ISA y el nivel de máquina del sistema operativo. Por tanto, en el momento de la ejecución pueden encontrarse tres programas —el programa objeto del usuario, el sistema operativo y el microprograma (si lo hay)— en la memoria de la computadora. Todos los rastros del programa fuente original han desaparecido. Así pues, el número de niveles presentes en el momento de la ejecución podría diferir del número de niveles presentes antes de la traducción. No obstante, cabe señalar que, si bien definimos un nivel por las instrucciones y construcciones lingüísticas con que cuentan sus programadores (y no por la técnica de implementación), otros autores a veces hacen una distinción mayor entre los niveles implementados con intérpretes en el momento de la ejecución y los niveles implementados por traducción.

7.1 INTRODUCCIÓN AL LENGUAJE ENSAMBLADOR

Los traductores pueden dividirse a grandes rasgos en dos grupos, dependiendo de la relación entre el lenguaje fuente y el lenguaje objetivo. Si el lenguaje fuente es en lo esencial una representación simbólica de un lenguaje de máquina numérico, el traductor se llama **ensamblador** y el lenguaje fuente se llama **lenguaje ensamblador**. Si el lenguaje fuente es un lenguaje de alto nivel como Java o C y el lenguaje objetivo es un lenguaje de máquina numérico o una representación simbólica de tal lenguaje, el traductor se llama **compilador**.

7.1.1 ¿Qué es un lenguaje ensamblador?

Un lenguaje ensamblador puro es un lenguaje en el que cada enunciado produce exactamente una instrucción de máquina. En otras palabras, existe una correspondencia uno a uno entre las instrucciones de máquina y los enunciados del programa en ensamblador. Si cada línea del programa en lenguaje ensamblador contiene exactamente un enunciado y cada palabra de máquina contiene exactamente una instrucción de máquina, entonces un programa en ensamblador de n líneas producirá un programa en lenguaje de máquina de n palabras.

La razón por la que la gente usa lenguaje ensamblador en lugar de programar en lenguaje de máquina (en hexadecimal) es que es mucho más fácil programar en lenguaje ensamblador. El uso de nombres simbólicos y direcciones simbólicas en lugar de direcciones binarias u octales representa una gran diferencia. Casi toda la gente puede recordar que las abreviaturas para sumar, restar, multiplicar y dividir son ADD, SUB, MUL y DIV, pero pocos pueden recordar los valores numéricos correspondientes que la máquina usa. El programador en lenguaje ensamblador sólo tiene que recordar los nombres simbólicos porque el ensamblador los traduce a instrucciones de máquina.

Los mismos comentarios aplican a las direcciones. El programador en lenguaje ensamblador puede asignar nombres simbólicos a posiciones de memoria y dejar que el ensamblador se preocupe por insertar los valores numéricos correctos. El programador en lenguaje de máquina siempre debe trabajar con los valores numéricos de las direcciones. Es por esto que ya nadie programa en lenguaje de máquina, aunque se hacía hace algunas décadas, antes de inventarse los ensambladores.

Los lenguajes ensambladores tienen otra propiedad, además de la correspondencia uno a uno entre enunciados en lenguaje ensamblador e instrucciones de máquina, que los distingue de los lenguajes de alto nivel. El programador en ensamblador tiene acceso a todas las características e instrucciones disponibles en la máquina objetivo. El programador en lenguaje de alto nivel no. Por ejemplo, si la máquina objetivo tiene un bit de desbordamiento, un programa en lenguaje ensamblador puede probarlo, pero un programa en Java no puede hacerlo directamente. Un programa en lenguaje ensamblador puede ejecutar todas las instrucciones del conjunto de instrucciones de la máquina objetivo, pero el programa en lenguaje de alto nivel no puede hacerlo. En síntesis, todo lo que puede hacerse en lenguaje de máquina puede hacerse en lenguaje ensamblador, pero muchas instrucciones, registros, y características similares no están disponibles para el programador en lenguaje de alto nivel. Los lenguajes para programación de sistemas, como C, suelen ser un híbrido entre estos dos tipos, con la sintaxis del lenguaje de alto nivel pero con una buena parte del acceso a la máquina que tiene un lenguaje ensamblador.

Una diferencia final que vale la pena dejar en claro es que un programa en lenguaje ensamblador sólo puede ejecutarse en una familia de máquinas, mientras que un programa escrito en un lenguaje de alto nivel tiene el potencial para ejecutarse en muchas máquinas. En muchas aplicaciones, esta posibilidad de trasladar software de una máquina a otra tiene gran importancia práctica.

7.1.2 ¿Por qué usar lenguaje ensamblador?

No nos engañemos: la programación en lenguaje ensamblador es difícil; no es para los débiles ni los cobardes. Además, escribir un programa en lenguaje ensamblador toma mucho más tiempo que escribir el mismo programa en un lenguaje de alto nivel; y también tarda mucho más en depurarse y es mucho más difícil de mantener.

En tales condiciones, ¿por qué alguien habría de programar en lenguaje ensamblador? Hay dos razones: desempeño y acceso a la máquina. Antes que nada, un programador experto en lenguaje ensamblador a menudo puede producir código que es mucho más pequeño y

rápido que el que puede producir un programador en lenguaje de alto nivel. En algunas aplicaciones la rapidez y el tamaño son críticas. Muchas aplicaciones incorporadas, como el código de una tarjeta inteligente, el código de un teléfono celular, los controladores de dispositivos, las rutinas de BIOS y los ciclos interiores de aplicaciones para las que el desempeño es crítico pertenecen a esta categoría.

En segundo lugar, algunos procedimientos requieren acceso total al hardware, cosa que suele ser imposible en lenguajes de alto nivel. Por ejemplo, los manejadores de interrupciones y trampas de bajo nivel de un sistema operativo, y los controladores de dispositivos de muchos sistemas incorporados de tiempo real pertenecen a esta categoría.

La primera razón para programar en lenguaje ensamblador (obtener un buen desempeño) suele ser la más importante, así que la examinaremos con mayor detenimiento. En casi todos los programas, un porcentaje pequeño del código ocupa un porcentaje elevado del tiempo de ejecución. Es común que el 1% del programa ocupe el 50% del tiempo de ejecución y que el 10% del programa ocupe el 90% del tiempo de ejecución.

Supongamos, por ejemplo, que se requieren 10 años-programador para escribir cierto programa en un lenguaje de alto nivel y que el programa resultante requiere 100 s para ejecutar cierto programa de prueba estándar. (Un **programa de prueba estándar** se usa para comparar computadoras, compiladores, etc.). Escribir todo el programa en lenguaje ensamblador podría requerir 50 años-programador, debido a la menor productividad de los programadores en lenguaje ensamblador. El programa final podría ejecutar el programa de prueba estándar en unos 33 s, ya que un programador ingenioso puede superar a un compilador ingenioso en un factor de 3 (aunque nadie se pone de acuerdo en estas proporciones). La situación se ilustra en la figura 7-1.

Con base en la observación de que sólo una diminuta fracción del código ocupa casi todo el tiempo de ejecución, puede adoptarse otra estrategia. El programa se escribe primero en un lenguaje de alto nivel. Luego se efectúa una serie de mediciones para determinar cuáles partes del programa ocupan la mayor parte del tiempo de ejecución. Tales mediciones normalmente incluirían usar el reloj del sistema para calcular el tiempo que se invierte en cada procedimiento, contabilizar el número de veces que cada ciclo se ejecuta, y acciones similares.

Por ejemplo, supongamos que el 10% del programa da cuenta del 90% del tiempo de ejecución. Esto implica que, de un trabajo de 100 s, 90 s se dedican a este 10% crítico y 10 s se dedican al 90% restante del programa. Ahora puede mejorarse el 10% crítico reescribiéndolo en lenguaje ensamblador. Este proceso se llama **afinación** y se ilustra en la figura 7-1. Aquí se requieren otros 5 años-programador para reescribir los procedimientos críticos pero su tiempo de ejecución se reduce de 90 s a 30 s.

Es provechoso comparar la estrategia mixta de lenguaje de alto nivel/lenguaje ensamblador con la versión únicamente en lenguaje ensamblador (vea la figura 7-1). La segunda es aproximadamente 20% más rápida (33 s *versus* 40 s) pero con un costo de más del triple (50 años-programador *versus* 15 años-programador). Además, la ventaja de la estrategia mixta es en realidad mayor que la indicada, porque la recodificación en lenguaje ensamblador de un procedimiento en lenguaje de alto nivel ya depurado es en realidad mucho más fácil que escribir el mismo procedimiento en lenguaje ensamblador desde cero. En otras palabras, el

	Años-programador para producir el programa	Tiempo de ejecución del programa en segundos
Lenguaje ensamblador	50	33
Lenguaje de alto nivel	10	100
Estrategia mixta antes de afinar		
Crítico 10%	1	90
Otro 90%	9	10
Total	10	100
Estrategia mixta después de afinar		
Crítico 10%	6	30
Otro 90%	9	10
Total	15	40

Figura 7-1. Comparación de programación en lenguaje ensamblador y lenguaje de alto nivel, con y sin afinación.

estimado de 5 años-programador para reescribir los procedimientos críticos en excesivamente conservador. Si esta recodificación en realidad sólo requiriera un año-programador, la relación de costos entre la estrategia mixta y la estrategia de lenguaje ensamblador pura sería de más de 4 a 1 en favor de la estrategia mixta.

Un programador que usa un lenguaje de alto nivel no está absorto en pasar bits de aquí para acá y a veces ve aspectos del problema que permiten efectuar mejoras *reales* en el desempeño. Esta situación pocas veces se presenta con los programadores de lenguaje ensamblador, quienes por lo regular se la pasan haciendo juegos malabares con las instrucciones a fin de ahorrarse unos cuantos ciclos.

Dos experiencias bien documentadas que ocurrieron durante la creación de MULTICS destacan este punto. Graham (1970) informó de un procedimiento en PL/I que se reescribió en tres meses; la nueva versión era 26 veces más pequeña que el original y 50 veces más rápida. Otro procedimiento se volvió 20 veces más pequeño y 40 veces más rápido con dos meses de trabajo. Corbató (1969) describió un procedimiento en PL/I que se redujo de 50,000 a 10,000 palabras de código compilado en menos de un mes, y un controlador de E/S que se encogió de 65,000 a 30,000 palabras de código compilado, con una mejora de la rapidez de un factor de 8 en cuatro meses. La cuestión aquí es que, como los programadores en lenguaje de alto nivel tienen una perspectiva más global de lo que están haciendo, tienen más posibilidades de generar ideas que conduzcan a algoritmos totalmente distintos e incomparablemente mejores.

A pesar de estas experiencias (y muchas otras similares), sigue habiendo por lo menos cuatro buenas razones para estudiar lenguaje ensamblador. Primera, dado que el éxito o fracaso de un proyecto grande podría depender de la capacidad para exprimir algún procedi-

miento crítico hasta mejorar su desempeño en un factor de 2 o 3, es importante poder escribir buen código en lenguaje ensamblador cuando realmente es necesario.

Segunda, el código de ensamblador a veces es la única alternativa debido a la escasez de memoria. Las tarjetas inteligentes contienen una CPU, pero pocas tienen un megabyte de memoria, y más pocas aún tienen un disco duro para paginar. A pesar de ello, tales tarjetas tienen que efectuar complejos cálculos criptográficos con recursos limitados. Los procesadores incorporados en los aparatos domésticos a menudo tienen un mínimo de memoria por razones de costo. Los asistentes digitales personales y otros dispositivos electrónicos inalámbricos de baterías suelen tener memorias pequeñas a fin de ahorrar energía, por lo que aquí también es indispensable un código pequeño y eficiente.

Tercera, un compilador debe producir salidas que un ensamblador pueda usar, o bien realizar el proceso de ensamblado él mismo. Por ello, entender el lenguaje ensamblador es indispensable para entender cómo funcionan los compiladores. Después de todo, alguien tiene que escribir el compilador (y su ensamblador).

Por último, el estudio del lenguaje ensamblador pone al descubierto la máquina real. Para los estudiantes de arquitectura de las computadoras, escribir algo de código de ensamblador es la única forma de ver realmente cómo son las máquinas en el nivel de arquitectura.

7.1.3 Formato de un enunciado en lenguaje ensamblador

Aunque la estructura de un enunciado en lenguaje ensamblador refleja la estructura de la instrucción de máquina que representa, los lenguajes de ensamblador para diferentes máquinas y diferentes niveles se parecen lo suficiente unos a otros como para poder hablar del lenguaje ensamblador en general. La figura 7-2 muestra fragmentos de programas en lenguaje ensamblador para el Pentium II, el Motorola 680x0 y el (Ultra)SPARC que realizan el cálculo $N = I + J$. En los tres ejemplos los enunciados que están arriba de la línea en blanco efectúan el cálculo. Los enunciados que están abajo de la línea en blanco son comandos para el ensamblador, que le ordenan reservar memoria para las variables *I*, *J* y *N*, y no son representaciones simbólicas de instrucciones de máquina.

Existen varios ensambladores para la familia Intel, cada uno con una sintaxis distinta. En este capítulo usaremos el lenguaje ensamblador MASM de Microsoft para nuestros ejemplos. Aunque nos estamos concentrando en el Pentium II, todo lo que digamos al respecto aplica igualmente al 386, 486, Pentium y Pentium Pro. Para el SPARC basaremos nuestros ejemplos en el ensamblador de Sun. Aquí también todo lo que digamos aplica también a las versiones anteriores (de 32 bits) de SPARC. Por uniformidad, usaremos mayúsculas para los códigos de operación y los registros en todos los casos (convención de Pentium II) aunque el ensamblador de Sun espera minúsculas.

Los enunciados en lenguaje ensamblador tienen cuatro partes: un campo de etiqueta, un campo de operación (código de operación), un campo de operandos y un campo de comentarios. Las etiquetas, que sirven para asignar nombres simbólicos a direcciones de memoria, se necesitan en enunciados ejecutables para poder saltar a esos enunciados. También se necesitan en las palabras de datos para poder accesar con un nombre simbólico a los datos ahí almacenados. Si un enunciado está etiquetado, la etiqueta (generalmente) inicia en la columna 1.

Etiqueta	Cód. op.	Operandos	Comentarios
FÓRMULA:	MOV	EAX,I	; registro EAX = I
	ADD	EAX,J	; registro EAX = I + J
	MOV	N,EAX	; N = I + J
I	DW	3	; reservar 4 bytes inicializados a 3
J	DW	4	; reservar 4 bytes inicializados a 4
N	DW	0	; reservar 4 bytes inicializados a 0

(a)

Etiqueta	Cód. op.	Operandos	Comentarios
FÓRMULA	MOVE.L	I, D0	; registro D0 = I
	ADD.L	J, D0	; registro D0 = I + J
	MOVE.L	D0, N	; N = 1 + J
I	DC.L	3	; reservar 4 bytes inicializados a 3
J	DC.L	4	; reservar 4 bytes inicializados a 4
N	DC.L	0	; reservar 4 bytes inicializados a 0

(b)

Etiqueta	Cód. op.	Operandos	Comentarios
FÓRMULA:	SETHI	%HI(I),%R1	! R1 = bits de orden alto de la dir. de I
	LD	[%R1+%LO(I)],%R1	! R1 = I
	SETHI	%HI(J),%R2	! R2 = bits de orden alto de la dir. de J
	LD	[%R2+%LO(J)],%R2	! R2 = J
	NOP		! esperar que J llegue de la memoria
	ADD	%R1,%R2,%R2	! R2 = R1 + R2
	SETHI	%HI(N),%R1	! R1 = bits de orden alto de la dir. de N
	ST	%R2,[%R1+%LO(N)]	
I:	.WORD 3		! reservar 4 bytes inicializados a 3
J:	.WORD 4		! reservar 4 bytes inicializados a 4
N:	.WORD 0		! reservar 4 bytes inicializados a 0

(c)

Figura 7-2. Cálculo de $N = I + J$. (a) Pentium II. (b) Motorola 680x0. (c) SPARC.

Cada una de las tres partes de la figura 7-2 tiene cuatro etiquetas: *FÓRMULA*, *I*, *J* y *N*. Observe que el lenguaje ensamblador de SPARC requiere un signo de dos puntos después de cada etiqueta, pero el de Motorola no. El de Intel requiere los dos puntos en las etiquetas de código pero no en las de datos. Esta diferencia nada tiene de fundamental. Los diseñadores de diferentes ensambladores suelen tener diferentes gustos. Nada en la arquitectura subyacente sugiere una opción o la otra. Una ventaja de la notación con dos puntos es que cuando se usa una etiqueta puede aparecer sola en una línea, con el código de operación en la columna 1 de la siguiente línea.

Tal estilo a veces es cómodo para los compiladores. Sin los dos puntos, no habría forma de distinguir un rótulo solo en una línea de un código de operación solo en una línea.

Una característica deplorable de algunos ensambladores es que los rótulos están restringidos a seis u ocho caracteres. En contraste, la mayor parte de los lenguajes de alto nivel permite usar nombres arbitrariamente largos. Los nombres largos bien elegidos hacen que los programas sean mucho más fáciles de entender por otras personas.

Cada una de las máquinas tiene algunos registros, pero reciben nombres muy diferentes. Los registros del Pentium II tienen nombres como EAX, EBX, ECX, etc. Los registros del Motorola se llaman D0, D1, D2, entre otros nombres. Los registros del SPARC tienen diversos nombres. Aquí usamos %R1 y %R2 para ellos.

El campo de código de operación contiene una abreviatura simbólica del código de operación —si el enunciado es una representación simbólica de una instrucción de máquina— o un comando para el ensamblador. La selección de un nombre apropiado es cuestión de gusto, y diferentes diseñadores de lenguaje ensamblador a menudo toman decisiones distintas. Los diseñadores del ensamblador de Intel decidieron usar MOV tanto para cargar un registro con el contenido de la memoria como para almacenar el contenido de un registro en la memoria. Los diseñadores del ensamblador de Motorola escogieron MOVE para las dos operaciones. En contraste, los diseñadores del ensamblador de SPARC decidieron usar LD para la primera y ST para la segunda. En este caso, también, las decisiones nada tienen que ver con la máquina subyacente.

En cambio, la necesidad de usar dos instrucciones de máquina, comenzando con SETHI, para accesar a la memoria, es una propiedad inherente a la arquitectura SPARC porque las direcciones virtuales tienen 32 bits (SPARC Versión 8) o 44 bits (SPARC Versión 9) y las instrucciones pueden contener cuando más 22 bits de datos intermedios. Por ello, siempre se requieren dos instrucciones para proporcionar todos los bits de una dirección virtual completa. Lo que

```
SETHI %HI(I),%R1
```

hace es poner en ceros los 32 bits superiores y los 10 bits inferiores del registro (de 64 bits) R1 y luego colocar los 22 bits superiores de la dirección de 32 bits de *I* en las posiciones de bit 10 a 31 de R1. La siguiente instrucción,

```
LD [%R1+%LO(I)],%R1
```

suma R1 y los 10 bits de orden bajo de la dirección de *I* para formar la dirección completa de *I*, obtener esa palabra de la memoria y colocarla en R1. En un concurso de belleza en el que se usara una escala del 1 al 10, estas instrucciones obtendrían una puntuación de aproximadamente –20, pero el SPARC no se diseñó pensando en la belleza de su lenguaje ensamblador; se diseñó pensando en la rapidez de ejecución, y esa meta sí se alcanza.

Las familias Pentium, 680x0 y SPARC permiten operandos de byte, palabra y largos. ¿Cómo sabe el ensamblador qué longitud debe usar? Una vez más, los diseñadores de ensambladores escogieron diferentes soluciones. En el Pentium II los registros con diferente longitud tienen nombres distintos; así, EAX se usa para elementos de 32 bits, AX sirve para mover elementos de 16 bits y, AL y AH se usan para elementos de 8 bits. Los diseñadores del ensamblador Motorola, en cambio, decidieron añadir a cada código de operación un sufijo: *.L* para largo, *.W* para palabra o *.B* para byte, en lugar de dar a subconjuntos de D0, etc., diferentes nombres. El SPARC usa diferentes códigos de operación para las distintas longitudes (por ejemplo, LDSB, LDSH y LDSW para cargar bytes con signo, medias palabras y palabras en un registro de 64 bits, respectivamente). Las tres técnicas son válidas, y destacan una vez más la naturaleza arbitraria del diseño de lenguajes.

Los tres ensambladores también difieren en la forma en que reservan espacio para los datos. Los diseñadores del lenguaje ensamblador de Intel escogieron DW (*Define Word*, definir palabra), aunque posteriormente se añadió .WORD como alternativa. A los de Motorola les gustó DC (Definir Constante). La gente de SPARC prefirió .WORD desde el principio. Una vez más, las diferencias son arbitrarias.

El campo de operandos de un enunciado en lenguaje ensamblador sirve para especificar las direcciones y registros que la instrucción de máquina usará como operandos. El campo de operandos de una instrucción de suma de enteros indica qué se sumará a qué. El campo de operandos de una instrucción de ramificación indica a dónde se saltará. Los operandos pueden ser registros, constantes, localidades de memoria, etcétera.

El campo de comentarios ofrece a los programadores un lugar para incluir explicaciones de cómo funciona el programa, para beneficio de otros programadores que podrían usar o modificar posteriormente el programa (o para el beneficio del programador original un año después). Un programa en lenguaje ensamblador sin tal documentación es casi incomprensible para todos los programadores, a menudo incluido el autor. El campo de comentarios es exclusivamente para consumo humano; no afecta el proceso de ensamblado ni el programa que se genera.

7.1.4 Seudoinstrucciones

Además de especificar qué instrucciones de máquina deben ejecutarse, un programa en lenguaje ensamblador puede contener comandos para el ensamblador mismo, por ejemplo para pedirle que asigne espacio en la memoria o que saque una nueva página de listado. Los comandos para el ensamblador se llaman **seudoinstrucciones** o a veces **directrices de ensamblador**. Ya vimos una seudoinstrucción típica en la figura 7-2(a): DW. En la figura 7-3 se dan algunas otras seudoinstrucciones tomadas del ensamblador Microsoft MASM para la familia Intel.

La seudoinstrucción SEGMENT inicia un nuevo segmento, y ENDS termina uno. Está permitido iniciar un segmento de texto, con código, luego iniciar un segmento de datos, luego volver al segmento de texto, etcétera.

ALIGN hace que la siguiente línea, casi siempre de datos, se coloque en una dirección que sea un múltiplo de su argumento. Por ejemplo, si el segmento actual ya tiene 61 bytes de datos, después de ALIGN 4 la siguiente dirección asignada será la 64.

EQU sirve para asignar un nombre simbólico a una expresión. Por ejemplo, después de la seudoinstrucción

```
BASE EQU 1000
```

puede usarse el símbolo BASE en cualquier punto en vez de 1000. La expresión que sigue a EQU puede contener varios símbolos definidos combinados con operadores aritméticos o de otro tipo, como en

Seudoinstrucción	Significado
SEGMENT	Iniciar un nuevo segmento (texto, datos, etc.) con ciertos atributos
ENDS	Terminar el segmento actual
ALIGN	Controlar la alineación de la siguiente instrucción o datos
EQU	Definir un nuevo símbolo igual a una expresión dada
DB	Asignar memoria para uno o más bytes (inicializados)
DD	Asignar memoria para una o más medias palabras de 16 bits (inicializadas)
DW	Asignar memoria para una o más palabras de 32 bits (inicializadas)
DQ	Asignar memoria para una o más palabras dobles de 64 bits (inicializadas)
PROC	Iniciar un procedimiento
ENDP	Terminar un procedimiento
MACRO	Iniciar una definición de macro
ENDM	Terminar una definición de macro
PUBLIC	Exportar un nombre definido en este módulo
EXTERN	Importar un nombre de otro módulo
INCLUDE	Traer e incluir otro archivo
IF	Iniciar ensamblado condicional con base en una expresión dada
ELSE	Iniciar ensamblado condicional si la condición IF previa fue falsa
ENDIF	Terminar ensamblado condicional
COMMENT	Definir nuevo carácter de inicio de comentario
PAGE	Generar un salto de página en el listado
END	Terminar el programa de ensamblador

Figura 7-3. Algunas de las seudoinstrucciones con que cuenta el ensamblador de Pentium II (MASM).

```
LIMIT EQU 4 * BASE + 2000
```

Casi todos los ensambladores, incluido MASM, exigen que un símbolo se defina antes de usarse en una expresión como ésta.

Las siguientes cuatro seudoinstrucciones, DB, DD, DW y DQ, asignan espacio para una o más variables de 1, 2, 4 u 8 bytes, respectivamente. Por ejemplo,

```
TABLE DB 11, 23, 49
```

asigna espacio para 3 bytes y los inicializa con los valores 11, 23 y 49, respectivamente. También define el símbolo *TABLE* y lo hace igual a la dirección donde está almacenado 11.

Las seudoinstrucciones PROC y ENDP definen el principio y el final de procedimientos en lenguaje ensamblador, respectivamente. Los procedimientos en lenguaje ensamblador

tienen la misma función que en otros lenguajes de programación. Así mismo, MACRO y ENDM delimitan una definición de macro. Estudiaremos las macros más adelante en este capítulo.

Las siguientes dos seudoinstrucciones, PUBLIC y EXTERN, controlan la visibilidad de los símbolos. Es común escribir programas como una colección de archivos. Con frecuencia, un procedimiento de un archivo necesita invocar un procedimiento o accesar una palabra de datos definidos en otro archivo. Para hacer posibles estas referencias de un archivo a otro, un símbolo que estará disponible para otros archivos se exporta con PUBLIC. Así mismo, para evitar que el ensamblador proteste por el uso de un símbolo que no está definido en el archivo en curso, dicho símbolo puede declararse como EXTERN, lo que indica al ensamblador que se definirá en algún otro archivo. Los símbolos que no se declaran en ninguna de estas seudoinstrucciones tienen como alcance el archivo local. Este *default* implica que usar, digamos, *FOO* en varios archivos no genera conflictos porque cada definición es local respecto a su propio archivo.

La seudoinstrucción INCLUDE hace que el ensamblador traiga otro archivo y lo incluya entero dentro del archivo actual. Tales archivos incluidos a menudo contienen definiciones, macros y otras cosas que se necesitan en múltiples archivos.

Muchos ensambladores, incluido MASM, manejan ensamblado condicional. Por ejemplo,

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:    DW   32
ELSE
WSIZE:    DW   16
ENDIF
```

asigna una sola palabra de 32 bits y llama a su dirección *WSIZE*. La palabra recibe el valor inicial 32 o 16, dependiendo del valor de *WORDSIZE*, que en este caso es 16. Esta construcción se usaría típicamente para escribir un programa que podría ensamblarse en máquinas de 16 bits (como el 8088) o de 32 bits (como el Pentium II). Al encerrar todo el código dependiente de la máquina entre IF y ENDIF, bastará con modificar una sola definición, la de *WORDSIZE*, para que el programa se ensamble automáticamente para cualquiera de los dos tamaños de palabra. Con este enfoque es posible mantener un programa fuente para varias máquinas objetivo (distintas), lo que facilita el desarrollo y mantenimiento del software. En muchos casos, todas las definiciones que dependen de la máquina, como *WORDSIZE*, se reúnen en un solo archivo, con diferentes versiones para diferentes máquinas. Si incluimos el archivo de definiciones correcto, podremos recompilar fácilmente el programa para diferentes máquinas.

La seudoinstrucción COMMENT permite al usuario cambiar el delimitador de comentarios a algo distinto del signo de punto y coma. PAGE sirve para controlar el listado que el ensamblador puede producir si se le solicita. Por último, END marca el final del programa.

Hay muchas otras seudoinstrucciones en MASM. Otros ensambladores para Pentium II cuentan con un conjunto diferente de seudoinstrucciones porque éstas no dependen de la arquitectura subyacente, sino del gusto de quien escribió el ensamblador.

7.2 MACROS

Los programadores en lenguaje ensamblador a menudo necesitan repetir sucesiones de instrucciones varias veces dentro de un programa. La forma más fácil de hacerlo es simplemente escribir las instrucciones requeridas donde sea que se necesiten. Sin embargo, si una secuencia es larga, o debe usarse muchas veces, escribirla una y otra vez resulta tedioso.

Una estrategia alternativa sería convertir la secuencia en un procedimiento e invocarlo cada vez que se necesite. Esta estrategia tiene la desventaja de que requiere la ejecución de una instrucción de llamada a procedimiento y una de retorno cada vez que se necesita la secuencia. Si la secuencia es corta —digamos, dos instrucciones— pero se usa con mucha frecuencia, el gasto extra de la llamada a procedimiento podría hacer al programa apreciablemente más lento. Las macros son una solución fácil y eficiente al problema de necesitar repetidamente la misma o casi la misma secuencia de instrucciones.

7.2.1 Definición, llamada y expansión de macros

Una **definición de macro** es una forma de asignar un nombre a un fragmento de texto. Una vez definida una macro, el programador puede escribir el nombre de la macro en lugar del fragmento de programa. Una macro es, de hecho, una abreviatura para un fragmento de texto. La figura 7-4(a) muestra un programa en lenguaje ensamblador para el Pentium II que intercambia dos veces el contenido de las variables *p* y *q*. Estas sucesiones podrían definirse como una macro, como se hace en la figura 7-4(b). Después de su definición, cada ocurrencia de *SWAP* hace que sea sustituida por las cuatro líneas:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

El programador definió *SWAP* como una abreviatura de los cuatro enunciados anteriores.

Aunque diferentes ensambladores tienen notaciones un poco distintas para definir macros, todas requieren las mismas partes básicas en una definición de macro:

1. Una cabecera de macro que da el nombre de la macro que se está definiendo.
2. El texto que constituye el cuerpo de la macro.
3. Una seudoinstrucción que marca el final de la definición (por ejemplo, ENDM).

Cuando el ensamblador se topa con una definición de macro, la guarda en una tabla de definición de macros para usarla posteriormente. A partir de ese punto, cada vez que el nombre de la macro (*SWAP* en el ejemplo de la figura 7-4) aparezca como código de operación, el ensamblador la sustituirá por el cuerpo de la macro. El uso de un nombre de macro como código de operación se denomina **llamada a macro** y su sustitución por el cuerpo de la macro se denomina **expansión de macro**.


```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

Figura 7-4. Código en lenguaje ensamblador para intercambiar P y Q dos veces. (a) Sin una macro. (b) Con una macro.

La expansión de macros se efectúa durante el proceso de ensamblado y no durante la ejecución del programa. Este punto es importante. El programa de la figura 7-4(a) y el de la figura 7-4(b) producen exactamente el mismo código en lenguaje de máquina. Si examinamos sólo el programa en lenguaje de máquina, es imposible saber si se usaron o no macros para generarlo. La razón es que una vez finalizada la expansión de macros, el ensamblador desecha las definiciones de macros. En el programa generado no queda ni rastro de ellas.

Las llamadas a macros no deben confundirse con llamadas a procedimientos. La diferencia básica es que una llamada a macro es una instrucción dirigida al ensamblador para que sustituya el nombre de la macro por el cuerpo de la misma. Una llamada a procedimiento es una instrucción de máquina que se inserta en el programa objeto y que posteriormente se ejecutará para llamar al procedimiento. En la figura 7-5 se comparan las llamadas a macros con las llamadas a procedimientos.

Aspecto	Llamada a macro	Llamada a procedimiento
¿Cuándo se hace la llamada?	Al ensamblar	Al ejecutarse el programa
¿El cuerpo se inserta en el programa objeto en cada lugar en el que se efectúa la llamada?	Sí	No
¿Se inserta una instrucción de llamada a procedimiento en el programa objeto y luego se ejecuta?	No	Sí
¿Se debe usar una instrucción de retorno después de efectuarse la llamada?	No	Sí
¿Cuántas copias del cuerpo aparecen en el programa objeto?	Una por llamada a la macro	1

Figura 7-5. Comparación de llamadas a macros y llamadas a procedimientos.

En lo conceptual, es mejor pensar que el proceso de ensamblado ocurre en dos pasadas. En la primera pasada, todas las definiciones de macros se guardan y las llamadas a macros se expanden. En la segunda pasada, el texto resultante se procesa como si estuviera en el programa original. Desde este punto de vista, el programa fuente se lee y se transforma en otro programa en el cual se han eliminado todas las definiciones de macros y en el que todas las llamadas a macros han sido sustituidas por sus cuerpos. La salida resultante, un programa en lenguaje ensamblador que no contiene macros, se alimenta al ensamblador.

Es importante tener en cuenta que un programa es una cadena de caracteres que incluye letras, dígitos, espacios, signos de puntuación y "retornos de carro" (cambio a una nueva línea). La expansión de macros consiste en sustituir ciertas subcadenas de esta cadena por otras cadenas de caracteres, sin fijarse en su significado.

7.2.2 Macros con parámetros

El recurso de macros que acabamos de describir puede servir para acortar programas en los que ocurre la misma secuencia exacta de instrucciones una y otra vez. Sin embargo, es común que un programa contenga varias secuencias de instrucciones que son casi idénticas pero no idénticas, como se ilustra en la figura 7-6(a). Aquí la primera secuencia intercambia *P* y *Q*, y la segunda secuencia intercambia *R* y *S*.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,R
MOV   EBX,S
MOV   S,EAX
MOV   R,EBX
```

(a)

```
CAMBIAR   MACRO P1, P2
          MOV EAX,P1
          MOV EBX,P2
          MOV P2,EAX
          MOV P1,EBX
          ENDM

          CAMBIAR P, Q

          CAMBIAR R, S
```

(b)

Figura 7-6. Sucesiones casi idénticas de enunciados. (a) Sin una macro. (b) Con una macro.

Los ensambladores de macros manejan el caso de secuencias casi idénticas permitiendo que las definiciones de macros incluyan **parámetros formales** y que las llamadas a macros incluyan **parámetros reales**. Cuando una macro se expande, cada parámetro formal que aparece en el cuerpo de la macro es sustituido por el parámetro real correspondiente. Los parámetros reales se colocan en el campo de operandos de la llamada a la macro. En la figura 7-6(b) se muestra el programa de la figura 7-6(a) reescrito empleando una macro con dos parámetros. Los símbolos *P1* y *P2* son los parámetros formales. Cada ocurrencia de *P1* dentro del cuerpo de una macro es sustituida por el primer parámetro real durante la expansión de la macro. Así mismo, *P2* es sustituido por el segundo parámetro real. En la llamada a macro

```
CAMBIAR P, Q
```

P es el primer parámetro real y *Q* es el segundo parámetro real. Así, los programas ejecutables que ambas partes de la figura 7-6 producen son idénticos.

7.2.3 Características avanzadas

Casi todos los procesadores de macros tienen una serie de características avanzadas que facilitan el trabajo del programador en lenguaje ensamblador. En esta sección examinaremos unas cuantas de las características avanzadas de MASM. Un problema que ocurre con todos los ensambladores que manejan macros es la duplicación de etiquetas. Suponga que una macro contiene una instrucción de ramificación condicional y una etiqueta a la cual se salta. Si la macro se llama dos o más veces, la etiqueta se duplicará y causará un error de ensamblado. Una solución es hacer que el programador proporcione como parámetro una etiqueta distinta en cada llamada. Una solución distinta (empleada por MASM) es permitir que una etiqueta se declare como LOCAL, y el ensamblador generará automáticamente una distinta en cada expansión de la macro. Algunos otros ensambladores tienen la regla de que las etiquetas numéricas son automáticamente locales.

MASM y casi todos los demás ensambladores permiten definir macros dentro de otras macros. Esta característica es útil sobre todo en combinación con el ensamblado condicional. Por lo regular, se define la misma macro en ambas partes de un enunciado IF, así:

```
M1   MACRO
     IF WORDSIZE GT 16
M2        MACRO
          ...
          ENDM
     ELSE
M2        MACRO
     ...
          ENDM
     ENDIF
          ENDM
```

En cualquier caso, se definirá la macro *M2*, pero la definición dependerá de si el programa se está ensamblando en una máquina de 16 bits o en una de 32 bits. Si no se invoca *M1*, *M2* no se definirá.

Por último, las macros pueden llamar a otras macros, incluidas ellas mismas. Si una macro es recursiva, es decir, si se llama a sí misma, deberá pasarse a sí misma un parámetro que se modificará en cada expansión, y la macro deberá probar el parámetro y terminar la recursión cuando llegue a cierto valor. En caso contrario, el ensamblador podría ingresar en un ciclo infinito. Si esto sucede, el usuario deberá suspender explícitamente el ensamblador.

7.2.4 Implementación de un recurso de macros en un ensamblador

Para implementar un recurso de macros, un ensamblador debe poder realizar dos funciones: guardar definiciones de macros y expandir llamadas de macros. Examinaremos ambas funciones por turno.

El ensamblador debe mantener una tabla de todos los nombres de macros y, junto con cada nombre, un apuntador a su definición almacenada para poder recuperarla cuando la necesite. Algunos ensambladores tienen una tabla aparte para nombres de macros y algunos tienen una tabla de códigos de operación combinada en la que se guardan todas las instrucciones de máquina, seudoinstrucciones y nombres de macros.

Cuando se llega a una definición de macro, se crea una entrada de tabla que da el nombre de la macro, el número de parámetros formales y un apuntador a otra tabla —la tabla de definiciones de macros— en la que se guardará el cuerpo de la macro. En este momento también se construye una lista de los parámetros formales que se usarán al procesar la definición. Luego se lee el cuerpo de la macro y se guarda en la tabla de definiciones de macros. Los parámetros formales que ocurren dentro del cuerpo se indican con algún símbolo especial. Por ejemplo, la representación interna de la definición de la macro *CAMBIAR* con un signo de punto y coma como "retorno de carro" y el signo "&" como símbolo de parámetro formal es:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Dentro de la tabla de definiciones de macros el cuerpo de la macro no es más que una cadena de caracteres.

Durante la primera pasada del ensamblador, se buscan en tablas los códigos de operación y las macros se expanden. Cada vez que se encuentra una definición de macro, se guarda en la tabla de macros. Cuando se invoca una macro, el ensamblador suspende temporalmente la lectura del dispositivo de entrada y comienza a leer del cuerpo almacenado de la macro. Los parámetros formales extraídos del cuerpo almacenado de la macro se sustituyen por los parámetros reales proporcionados en la llamada. La presencia de un "&" antes de los parámetros formales permite al ensamblador reconocerlos fácilmente.

7.3 EL PROCESO DE ENSAMBLADO

En las secciones que siguen describiremos brevemente cómo funciona un ensamblador. Aunque cada máquina tiene un lenguaje ensamblador distinto, el proceso de ensamblado tiene suficientes similitudes en las distintas máquinas que es posible describirlo en términos generales.

7.3.1 Ensambladores de dos pasadas

Dado que un programa en lenguaje ensamblador consta de una serie de enunciados de una sola línea, al principio podría parecer natural tener un ensamblador que lea un enunciado, lo traduzca a lenguaje de máquina y luego escriba la instrucción de lenguaje de máquina generada en un archivo, escribiendo al mismo tiempo la línea correspondiente del listado, si se va a producir, en otro archivo. Este proceso se repetiría hasta haber traducido todo el programa. Lo malo es que tal estrategia no funciona.

Considere la situación en la que el primer enunciado es un salto a *L*. El ensamblador no podrá ensamblar este enunciado hasta que no conozca la dirección del enunciado *L*. Dicho enunciado podría estar cerca del final del programa, y para el ensamblador sería imposible encontrar la dirección sin leer antes casi todo el programa. Esto se conoce como el **problema de la referencia adelantada (hacia adelante)**, porque se ha usado un símbolo, *L*, antes de definirse; es decir, se ha hecho referencia a un símbolo cuya definición ocurrirá posteriormente.

Las referencias adelantadas se pueden manejar de dos maneras. Primera, el ensamblador podría leer el programa fuente dos veces. Cada lectura del programa fuente es una **pasada**; cualquier traductor que lee dos veces el programa de entrada es un **traductor de dos pasadas**. En la primera pasada de un ensamblador de dos pasadas, se reúnen las definiciones de símbolos, incluidas las etiquetas de enunciados, y se guardan en una tabla. Para cuando se inicia la segunda pasada, ya se conocen los valores de todos los símbolos, así que no quedan ya referencias adelantadas y cada enunciado puede leerse, ensamblarse y enviarse hacia la salida. Aunque este enfoque requiere una pasada extra por las entradas, es conceptualmente sencillo.

La segunda estrategia consiste en leer el programa en lenguaje ensamblador una vez, convertirlo en una forma intermedia, y guardar esta forma intermedia en una tabla en la memoria. Luego se efectúa otra pasada, pero ahora por la tabla, no por el programa fuente. Si hay suficiente memoria (o memoria virtual), este enfoque ahorra tiempo de E/S. Si se va a producir un listado, entonces hay que guardar todo el enunciado fuente, incluidos los comentarios. Si no se requiere un listado, la forma intermedia puede reducirse a lo indispensable.

En cualquier caso, otra tarea de la primera pasada es guardar todas las definiciones de macros y expandir las llamadas conforme se llega a ellas. Por ello, la definición de los símbolos y la expansión de las macros generalmente se combina en una sola pasada.

7.3.2 Primera pasada

La principal función de la primera pasada es construir una tabla llamada **tabla de símbolos**, que contiene los valores de todos los símbolos. Un símbolo es una etiqueta o bien un valor al que se le asigna un nombre simbólico mediante una seudoinstrucción como

```
BUFSIZE EQU 8192
```

Al asignar un valor a un símbolo en el campo de etiqueta de una instrucción, el ensamblador debe saber qué dirección tendrá esa instrucción durante la ejecución del programa. Para seguir la pista a la dirección de tiempo de ejecución de la instrucción que se está ensamblando, el ensamblador mantiene una variable llamada **contador de posiciones de instrucciones** (**ILC**, *Instruction Location Counter*). Esta variable se pone en 0 al principio de la primera pasada y se incrementa en una cantidad igual a la longitud de la instrucción cada vez que se procesa una instrucción, como se muestra en la figura 7-7. Este ejemplo es para el Pentium II.

En adelante no daremos ejemplos de SPARC (ni Motorola) porque las diferencias entre los lenguajes ensambladores no son muy importantes, y deberá bastar con un ejemplo. Además, si hubiera un concurso para hallar el lenguaje ensamblador más incomprensible del mundo, el SPARC sería uno de los favoritos.

Etiqueta	Cód. op.	Operandos	Comentarios	Longitud	ILC
MARÍA:	MOV	EAX,I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	Saltar a DONE	5	129

Figura 7-7. El contador de localidades de instrucciones (ILC) sigue la pista a las direcciones donde las instrucciones se cargarán en la memoria. En este ejemplo, los enunciados anteriores a MARÍA ocupan 100 bytes.

La primera pasada de casi todos los ensambladores utiliza por lo menos tres tablas: la tabla de símbolos, la tabla de seudoinstrucciones y la tabla de códigos de operación. Si se necesita, también se mantiene una tabla de literales. La tabla de símbolos tiene una entrada para cada símbolo, como se muestra en la figura 7-8. Los símbolos se definen ya sea usándolos como etiquetas o por definición explícita (por ejemplo, EQU). Cada entrada de la tabla de símbolos contiene el símbolo mismo (o un apuntador a él), su valor numérico y a veces otra información. Esta información adicional podría incluir

1. La longitud del campo de datos asociado al símbolo.
2. Los bits de reubicación. (¿El símbolo cambia de valor si el programa se carga en una dirección diferente de la que el ensamblador supuso?)
3. Si el símbolo estará o no accesible afuera del procedimiento.

Símbolo	Valor	Otra información
MARÍA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Figura 7-8. Tabla de símbolos para el programa de la figura 7-7.

La tabla de códigos de operación contiene al menos una entrada para cada código de operación simbólico (mnemónico) del lenguaje ensamblador. En la figura 7-9 se muestra parte de una tabla de códigos de operación. Cada entrada contiene el código de operación simbólico, dos operandos, el valor numérico del código de operación, la longitud de la instrucción y un número de tipo que divide los códigos de operación en grupos dependiendo del número y el tipo de operandos.

Cód. op.	Primer operando	Segundo operando	Cód. op. hexadecimal	Longitud de instrucción	Clase de instrucción
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figura 7-9. Extractos de la tabla de códigos de operación para un ensamblador de Pentium II.

Por ejemplo, consideremos el código de operación ADD. Si una instrucción ADD contiene EAX como primer operando y una constante de 32 bits (immed32) como segundo operando, entonces se usa el código de operación 0x05 y la longitud de la instrucción es de 5 bytes. (Las constantes que se pueden expresar en 8 o 16 bits usan diferentes códigos de operación, que no se muestran.) Si se usa ADD con dos registros como operandos, la instrucción ocupa 2 bytes y tiene el código de operación 0x01. La clase de instrucción (arbitraria) 19 se asigna a todas las combinaciones código de operación-operando que siguen las mismas reglas y deben procesarse de la misma manera que ADD con dos registros como operandos. La clase de instrucción designa en realidad un procedimiento dentro del ensamblador que se invoca para procesar todas las instrucciones de un tipo dado.

Algunos ensambladores permiten a los programadores escribir instrucciones empleando direccionamiento inmediato aunque no exista una instrucción correspondiente en el lenguaje objetivo. Tales instrucciones "seudoinmediatas" se manejan como sigue. El ensamblador asigna memoria para el operando inmediato al final del programa y genera una instrucción que hace referencia a ella. Por ejemplo, la *mainframe* IBM 3090 no tiene instrucciones inmediatas; no obstante, los programadores pueden escribir

```
L 14,=F'5'
```

para cargar el registro 14 con una constante de palabra completa (Full) con valor de 5. De este modo, el programador evita escribir explícitamente una seudoinstrucción para asignar una palabra que se inicializa con 5, darle una etiqueta y luego usarla en la instrucción L. Las constantes para las que el ensamblador automáticamente reserva memoria se llaman **literales**. Además de ahorrar al programador un poco de trabajo, las literales hacen más comprensibles los programas porque hacen evidente el valor de la constante en el enunciado fuente. La primera pasada del ensamblador debe construir una tabla de todas las literales empleadas en el programa. Nuestras tres computadoras de ejemplo tienen instrucciones inmediatas, por lo que sus ensambladores no manejan literales. Las instrucciones inmediatas son muy comunes hoy día, pero antes eran raras. Es probable que lo extendido del uso de literales hizo que los diseñadores de máquinas se dieran cuenta de que el direccionamiento

inmediato era una buena idea. Si se necesitan literales, se mantiene una tabla de literales durante el ensamblado, y se crea una nueva entrada cada vez que se encuentra una literal. Después de la primera pasada, esta tabla se ordena y se eliminan los duplicados.

La figura 7-10 muestra un procedimiento que podría servir como base para la primera pasada de un ensamblador. El estilo de programación es notable por derecho propio. Los nombres de procedimiento se escogieron de modo que den una idea clara de lo que los procedimientos hacen. Lo más importante es que la figura 7-10 representa un bosquejo de la primera pasada que, aunque no es completo, es un buen punto de partida. Es lo bastante corto como para entenderse fácilmente y deja bien claro cuál debe ser el siguiente paso: escribir los procedimientos que se usan en él.

Algunos de estos procedimientos serán relativamente cortos, como *verif_simbolo*, que simplemente devuelve el símbolo como cadena de caracteres si lo hay y null si no lo hay. Otros procedimientos, como *obt_long_de_tipo1* y *obt_long_de_tipo2*, podrían ser más largos e invocar otros procedimientos. En general, el número de tipos no será dos, sino que dependerá del lenguaje que se ensambla y del número de tipos de instrucciones que tiene.

Estructurar los programas de este modo tiene otras ventajas además de la facilidad de programación. Si el ensamblador está siendo escrito por un grupo de personas, los distintos procedimientos pueden repartirse entre los programadores. Todos los detalles (desagradables) de la obtención de las entradas se ocultan en *leer_sig_linea*. Si llegaran a cambiar —por ejemplo, por un cambio de sistema operativo— sólo se verá afectado un procedimiento subsidiario; el procedimiento *primera_pasada* en sí no requerirá modificaciones.

A medida que lee el programa, la primera pasada tiene que analizar sintácticamente cada línea para encontrar el código de operación (digamos ADD), buscar su tipo (básicamente, el patrón de operandos) y calcular la longitud de la instrucción. Esta información también se necesita en la segunda pasada, así que podríamos registrarla explícitamente para no tener que analizar la línea desde cero la próxima vez. Sin embargo, la reescritura del archivo de entrada causa más E/S. La ventaja relativa de hacer más E/S para eliminar el análisis sintáctico o hacer menos E/S y más análisis depende de las velocidades relativas de la CPU y el disco, la eficiencia del sistema de archivos y otros factores. En este ejemplo escribiremos un archivo temporal que contiene el tipo, código de operación, longitud y la línea de entrada real. Esta línea es la que se leerá en la segunda pasada en lugar del archivo de entrada original.

Cuando se lee la seudoinstrucción END, termina la primera pasada. En este momento se ordenan la tabla de símbolos y la de literales si es necesario. También se buscan y se eliminan entradas repetidas en la tabla de literales.

7.3.3 Segunda pasada

La función de la segunda pasada es generar el programa objeto y posiblemente imprimir el listado de ensamblado. Además, la segunda pasada debe producir cierta información que el enlazador necesita para enlazar procedimientos ensamblados en diferentes momentos para producir un solo archivo ejecutable. La figura 7-11 muestra un bosquejo de procedimiento para la segunda pasada.


```
public static void primera_pasada( ) {
   // Este procedimiento es un bosquejo de la primera pasada de un ensamblador sencillo.
   boolean mas_entradas = true;                    // bandera que detiene la primera pasada
   String linea, simbolo, literal, codop;          // campos de la instrucción
   int contador_posic, longitud, valor, tipo;      // variables diversas
   final int ENUNC_FINAL = –2;                     // indica fin de las entradas

   contador_posic = 0;                             // ensamblar la primera instrucción en 0
   inicializar_tablas( );                          // inicialización general

   while (mas_entradas) {                          // END asigna false a mas_entradas
   linea = leer_sig_linea( );                      // obtener una línea de entrada
      longitud = 0;                                // número de bytes en la instrucción
      tipo = 0;                                    // tipo (formato) de la instrucción

      if (linea_no_es_comentario(linea)) {
         simbolo = verif_simbolo(linea);           // ¿tiene rótulo esta línea?
         if (simbolo != null)                      // en tal caso, registrar símbolo y valor
            ingresar_simbolo_nuevo(simbolo, contador_posic);
         (literal = verif_literal(linea);          // ¿la línea contiene una literal?
         if (literal != null)                      // en tal caso, ingresarla en la tabla
            ingresar_literal_nueva(literal);

         // Ahora determinar el tipo del cód. op. –1 indica código no válido.
         codop = extraer_codop(linea);             // buscar mnemónico del código de operación
         tipo = buscar_tabla_cods(codop);          // buscar formato, por ejemplo, OP REG1, REG2
         if (tipo < 0)                             // si no es cód. op., ¿es seudoinstrucción?
            tipo = buscar_tabla_seudo(codop);
         switch(tipo) {                            // determinar longitud de esta instrucción
            case 1: longitud = obt_long_de_tipo1(linea); break;
            case 2: longitud = obt_long_de_tipo2(linea); break;
            // otros casos aquí
         }
      }

      escrib_arch_temp(tipo, codop, longitud, linea); // inf. útil p/segunda pasada
      contador_posic = contador_posic + longitud   // actualizar cont. posic.
      if (tipo == ENUNC_FINAL) {                   // ¿terminamos con las entradas?
         mas_entradas = false;                     // entonces, realizar tareas de aseo
         rebob_temp_segunda_pasada( );             // como rebobinar el archivo temporal
         ordenar_tabla_lit( );                     // y ordenar la tabla de literales
         quitar_literales_rep( );                  // y eliminar de ella los duplicados
      }
   }
}
```

Figura 7-10. Primera pasada de un ensamblador sencillo.

El funcionamiento de la segunda pasada es parecido al de la primera: se leen las líneas una por una y se procesan una por una. Dado que ya escribimos el tipo, código de operación y longitud al principio de cada línea (en el archivo temporal), se leen todos estos datos para

```
public static void segunda_pasada( ) {
    // Este procedimiento es un bosquejo de la segunda pasada de un ensamblador sencillo.
    boolean mas_entradas = true;                    // bandera que detiene la pasada
    String línea, codop;                            // campos de la instrucción
    int contador_posic, longitud, tipo;             // variables diversas
    final int ENUNC_FINAL = –2;                     // indica fin de las entradas
    final int MAX_CODIGO = 16;                      // límite de bytes de código por instruc.
    byte código[ ] = new byte[MAX_CODIGO];          // contiene el código generado por instruc.

    contador_posic = 0;                             // ensamblar primera instrucción en 0

    while (mas_entradas) {                          // END asigna false a mas_entradas
        tipo = leer_tipo( );                        // obtener campo de tipo de la sig. linea
        codop = leer_codop( );                      // obtener campo de cód. op. de la sig. linea
        longitud = leer_long( );                    // obtener campo de longit. de la sig. linea
        linea = leer_linea( );                      // obtener la linea de entrada real

        if (tipo != 0) {                            // el tipo 0 es para comentarios
            switch(tipo) {                          // generar código de salida
                case 1: eval_tipo1(codop, longitud, linea, codigo); break;
                case 2: eval_tipo2(codop, longitud, linea, codigo); break;
                // otros casos aquí
            }
        }

        escribir_salida(codigo);                    // escribir el código binario
        escribir_listado(codigo, linea)             // imprimir una línea del listado
        contador_posic = contador_posic + longitud; // actualizar cont. pos.
        if (tipo == ENUNC_FINAL) {                  // ¿terminamos con las entradas?
            mas_entradas = false;                   // entonces, realizar tareas de aseo
            finalizar( );                           // diversas cosas
        }
    }
}
```

Figura 7-11. Segunda pasada de un ensamblador sencillo.

ahorrar algo de análisis. El trabajo principal de la generación de código corre por cuenta de los procedimientos *eval_tipo1*, *eval_tipo2*, etc. Cada uno maneja un patrón específico, como el de un código de operación y dos operandos en registros, genera el código binario para la instrucción y lo devuelve en *código*. Luego se escribe el código. Lo más probable es que *escribir_código* se limite a colocar en un buffer el código binario acumulado y escriba el archivo en disco en fragmentos grandes a fin de reducir el tráfico con el disco.

El enunciado fuente original y el código objeto generado a partir de él (en hexadecimal) pueden imprimirse o colocarse en un buffer para imprimirse posteriormente. Una vez que se ha ajustado el ILC, se adquiere el siguiente enunciado.

Hasta ahora hemos supuesto que el programa fuente no contiene errores. Cualquiera que haya escrito alguna vez un programa, en cualquier lenguaje, sabe qué tan realista es ese supuesto. Entre los errores más comunes están:

1. Se usó un símbolo pero no se definió.
2. Se definió un símbolo más de una vez.
3. El nombre que está en el campo de código de operación no corresponde a un código válido.
4. Un código de operación no tiene suficientes operandos.
5. Un código de operación tiene demasiados operandos.
6. Un número octal contiene un 8 o un 9.
7. Uso incorrecto de registros (por ejemplo, un salto a un registro).
8. Falta la instrucción END.

Los programadores son muy ingeniosos y siempre se les ocurren nuevos errores. Los errores de símbolo no definido a menudo se deben a errores de dedo, por lo que un ensamblador inteligente podría tratar de deducir cuál de los símbolos definidos se parece más al que no está definido y usarlo en vez de éste. Poco puede hacerse para corregir casi todos los demás errores. Lo mejor que puede hacer el ensamblador con un enunciado descarriado es imprimir un mensaje de error y tratar de continuar con el ensamblado.

7.3.4 La tabla de símbolos

Durante la primera pasada del proceso de ensamblado, el ensamblador acumula información acerca de los símbolos y sus valores, la cual debe guardarse en la tabla de símbolos para consultarla durante la segunda pasada. Hay varias formas de organizar la tabla de símbolos, y a continuación describiremos brevemente algunos de ellos. En todos los casos se intenta simular una **memoria asociativa**, que conceptualmente es un conjunto de pares (símbolo, valor). Dado el símbolo, la memoria asociativa debe producir el valor.

La técnica más sencilla es implementar la tabla de símbolos como un arreglo de pares cuyo primer elemento es el símbolo (o apunta a él) y cuyo segundo elemento es el valor (o apunta a él). Dado un símbolo qué buscar, la rutina de la tabla de símbolos simplemente recorre la tabla linealmente hasta que encuentra una entrada que coincide. Este método es fácil de programar pero es lento porque, en promedio, habrá que examinar media tabla en cada búsqueda.

Otra forma de organizar la tabla de símbolos consiste en ordenarla por símbolo y usar el algoritmo de **búsqueda binaria** para localizar un símbolo. Este algoritmo opera comparando la entrada que está a la mitad de la tabla con el símbolo. Si el símbolo es alfabéticamente anterior a dicha entrada, quiere decir que se encuentra en la primera mitad de la tabla; si no, deberá estar en la segunda mitad. Si el símbolo es igual a la entrada media, la búsqueda termina.

Suponiendo que la entrada media no es igual al símbolo buscado, al menos sabemos en qué mitad de la tabla debemos buscarlo. Ahora podemos aplicar la búsqueda binaria a la mitad correcta y, o bien encontraremos el símbolo o sabremos en qué cuarto de la tabla está.

Si aplicamos este algoritmo recursivamente, una tabla con n entradas puede examinarse en aproximadamente $\log_2 n$ intentos. Obviamente, este método es mucho más rápido que una búsqueda lineal, pero requiere mantener ordenada la tabla.

Una forma totalmente distinta de simular una memoria asociativa es usar una técnica llamada **codificación por dispersión (*hash coding*)**. Este enfoque necesita una función "*hash*" que establece una correspondencia entre los símbolos y enteros dentro del intervalo 0 a $k - 1$. Una posible función sería multiplicar entre sí los códigos ASCII de los caracteres de los símbolos, hacer caso omiso del desbordamiento, y obtener el residuo de la división del resultado entre k, o dividir el resultado entre un número primo. De hecho, se puede usar casi cualquier función que dé una distribución uniforme de los valores *hash*. Los símbolos pueden guardarse en una tabla que consiste en k **cubetas** numeradas de 0 a $k - 1$. Todos los pares (símbolo, valor) cuyo símbolo da i por *hashing* se almacenan en una lista enlazada a la que apunta la entrada i de la tabla *hash*. Con n símbolos y k entradas en la tabla *hash*, una lista promedio tendrá una longitud n/k. Si escogemos k aproximadamente igual a n, generalmente podremos localizar los símbolos con una sola consulta. Si ajustamos k podremos reducir el tamaño de la tabla a expensas de hacer más lentas las búsquedas. La codificación *hash* se ilustra en la figura 7.12.

7.4 ENLAZADO Y CARGA

Casi todos los programas consisten en más de un procedimiento. Los compiladores y ensambladores generalmente traducen un procedimiento a la vez y colocan la salida traducida en disco. Antes de que el programa pueda ejecutarse, es preciso encontrar todos los procedimientos traducidos y enlazarlos en forma correcta. Si no se cuenta con memoria virtual, el programa enlazado también se deberá cargar explícitamente en la memoria principal. Los programas que realizan estas funciones reciben diversos nombres, como **enlazador** (*linker*), **cargador de enlace** (*linking loader*) y **editor de enlace** (*linkage editor*). La traducción completa de un programa fuente requiere dos pasos, como se muestra en la figura 7-13:

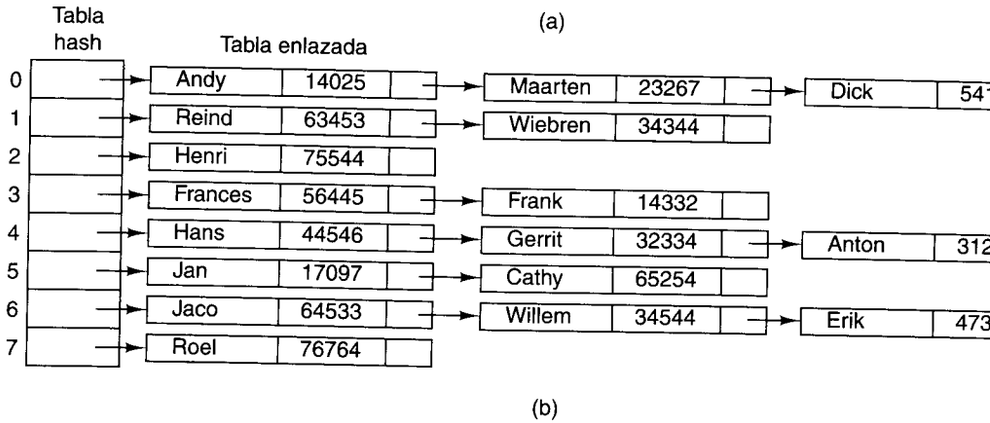
1. Compilación o ensamblado de los procedimientos fuente.
2. Enlazado de los módulos objeto.

El compilador o ensamblador se encarga del primer paso, y el segundo corre por cuenta del enlazador.

La traducción de procedimiento fuente a módulo objeto representa un cambio de nivel porque el lenguaje fuente y el lenguaje objetivo tienen diferentes instrucciones y notación. El proceso de enlazado, en cambio, no representa un cambio de nivel, ya que tanto las entradas del enlazador como sus salidas son programas para la misma máquina virtual. La función del enlazador es reunir procedimientos traducidos por separado y enlazarlos para que se ejecuten como una unidad llamada **programa binario ejecutable**. En MS-DOS Windows 95/98 y NT, los módulos objeto tienen la extensión *.obj* y los programas binarios ejecutables tienen la extensión *.exe*. En UNIX, los módulos objetos tienen la extensión *.o*; los programas binarios ejecutables no tienen extensión.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

Figura 7-12. Codificación *hash o por dispersión*. (a) Símbolos, valores y códigos *hash* derivados de los símbolos. (b) Tabla *hash* de ocho entradas con listas enlazadas de símbolos y valores.

Los compiladores y ensambladores traducen cada procedimiento fuente como entidad individual por una buena razón. Si un compilador o ensamblador leyera una serie de procedimientos fuente y produjera directamente un programa en lenguaje de máquina listo para ejecutarse, la modificación de un enunciado de un procedimiento fuente requeriría volver a traducir todos los procedimientos fuente.

Si se usa la técnica de módulos objeto separados de la figura 7-13, sólo es necesario volver a traducir el procedimiento que se modificó y no los demás, aunque sí es necesario volver a enlazar todos los módulos objeto. Sin embargo, el enlazado suele ser mucho más rápido que la traducción; así, el proceso de dos pasos de traducir y enlazar puede ahorrar mucho tiempo durante el desarrollo de un programa. Esta ganancia es importante sobre todo si los programas tienen cientos o miles de módulos.

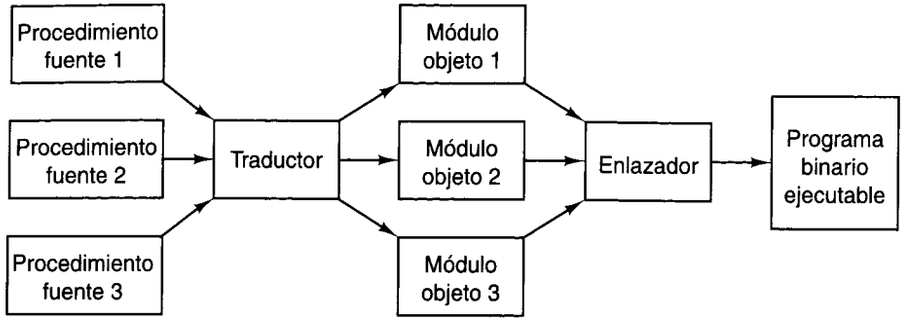


Figura 7-13. La generación de un programa binario ejecutable a partir de una colección de procedimientos fuente traducidos de forma independiente requiere un enlazador.

7.4.1 Tareas que realiza el enlazador

Al principio de la primera pasada del proceso de ensamblado, el contador de posiciones de instrucciones se pone en 0. Este paso equivale a suponer que el módulo objeto estará en la dirección (virtual) 0 durante su ejecución. En la figura 7-14 se muestran cuatro módulos objeto para una máquina genérica. En este ejemplo, cada módulo comienza con una instrucción de SALTAR a una instrucción de PASAR dentro del módulo.

Para ejecutar el programa, el enlazador obtiene los módulos objeto de la memoria principal a fin de formar la imagen de programa binario ejecutable, como se muestra en la figura 7-15(a). De lo que se trata es de crear una imagen exacta del espacio de direcciones virtual del programa ejecutable dentro del enlazador y colocar todos los módulos objeto en sus posiciones correctas. Si no hay suficiente memoria (virtual) para formar la imagen, puede usarse un archivo en disco. Por lo regular se usa una sección pequeña de la memoria que comienza en la dirección cero para vectores de interrupción, comunicación con el sistema operativo, atrapar apuntadores no inicializados u otras cosas, por lo que los programas casi siempre comienzan arriba de 0. En esta figura hemos iniciado (arbitrariamente) los programas en la dirección 100.

El programa de la figura 7-15(a), aunque ya se cargó en la imagen del archivo binario ejecutable, todavía no está listo para ejecutarse. Considere qué sucedería si la ejecución comenzara con la instrucción que está al principio del módulo *A*. El programa no saltaría a la instrucción PASAR como debería, porque esa instrucción ahora está en 300. De hecho, todas las instrucciones que hacen referencia a la memoria fallarían por la misma razón.

Este problema, llamado **problema de reubicación**, ocurre porque cada módulo objeto de la figura 7-14 representa un espacio de direcciones individual. En una máquina que tiene espacio de direcciones segmentado, como la Pentium II, cada módulo objeto podría en teoría tener su propio espacio de direcciones si se le coloca en su propio segmento. Sin embargo, OS/2 es el único sistema operativo para el Pentium II que reconoce este concepto. Todas las versiones de Windows y UNIX sólo reconocen un espacio de direcciones lineal, así que todos los módulos objeto deben fusionarse en un solo espacio de direcciones.

Además, las instrucciones de llamada a procedimiento de la figura 7-15(a) tampoco funcionan. En la dirección 400, la intención del programador era llamar al módulo objeto *B*, pero

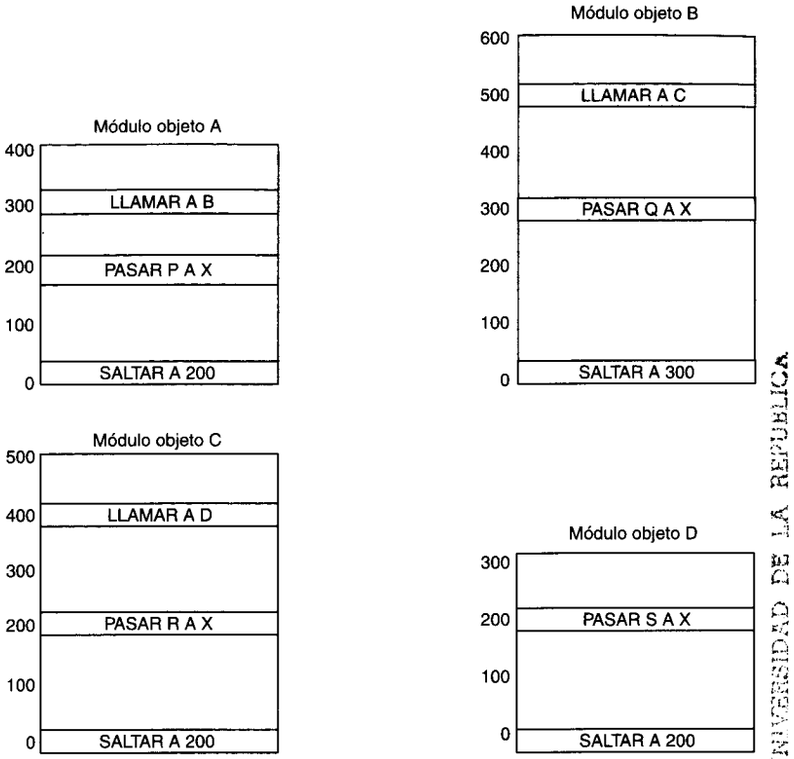


Figura 7-14. Cada módulo tiene su propio espacio de direcciones, que comienza en 0.

dado que cada procedimiento se traduce por su cuenta, el ensamblador no tiene forma de saber qué dirección debe insertar en la instrucción LLAMAR A B. La dirección del módulo objeto *B* no se conoce antes del enlazado. Este problema se denomina **problema de referencia externa**. El enlazador puede resolver ambos problemas.

El enlazador fusiona los espacios de direcciones de los módulos objeto en un solo espacio de direcciones lineal siguiendo estos pasos:

1. Construcción de una tabla con todos los módulos objeto y sus longitudes.
2. Con base en esta tabla, se asigna una dirección de inicio a cada módulo objeto.
3. Se buscan todas las instrucciones que hacen referencia a la memoria y se suma a cada una de ellas una **constante de reubicación** igual a la dirección de inicio de su módulo.
4. Se buscan todas las instrucciones que hacen referencia a otros procedimientos y se inserta la dirección del procedimiento en cuestión.

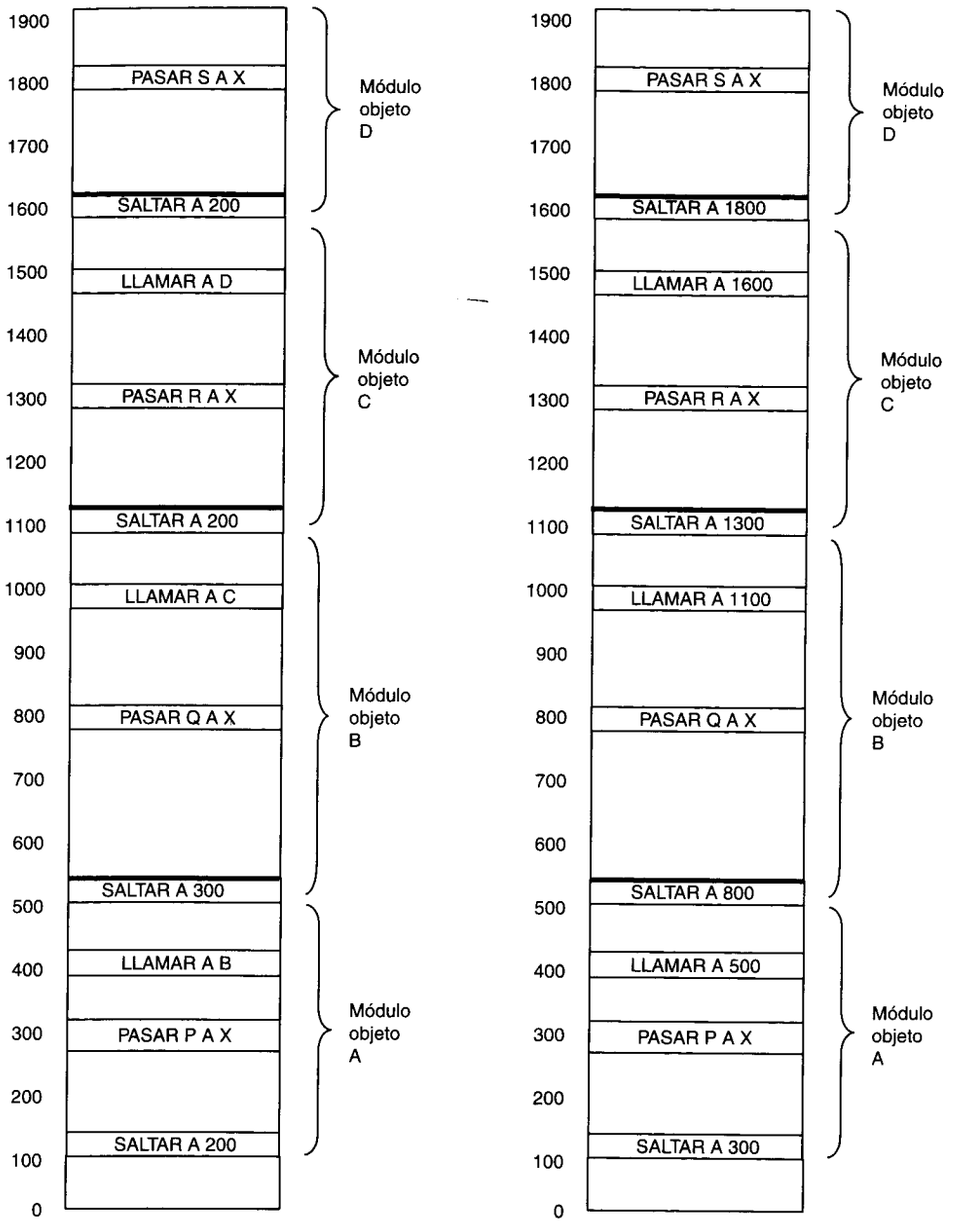


Figura 7-15. (a) Los módulos objeto de la figura 7-14 después de colocarse en la imagen binaria pero antes de reubicarse y enlazarse. (b) Los mismos módulos objeto una vez efectuados el enlazado y la reubicación. Juntos forman un programa binario ejecutable, listo para trabajar.

A continuación se muestra la tabla de módulos objeto que se construye en el paso 1 para los módulos de la figura 7-15. La tabla da el nombre, longitud y dirección de inicio de cada módulo.

Módulo	Longitud	Dirección de inicio
A	400	100
B	600	500
C	500	1100
D	300	1600

La figura 7-15(b) muestra el aspecto que el espacio de direcciones de la figura 7-15(a) tiene después de que el enlazador ha llevado a cabo estos pasos.

7.4.2 Estructura de un módulo objeto

Los módulos objeto a menudo contienen seis partes, que se muestran en la figura 7-16. La primera parte contiene el nombre del módulo, cierta información que el enlazador necesita, como las longitudes de las distintas partes del módulo, y a veces la fecha de ensamblado.

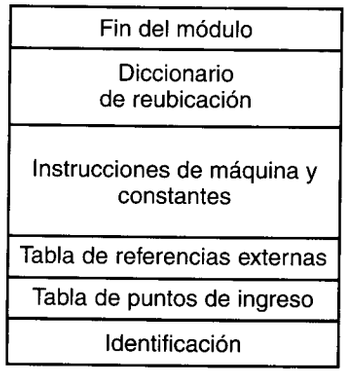


Figura 7-16. Estructura interna de un módulo objeto producido por un traductor.

La segunda parte del módulo objeto es una lista de los símbolos definidos en el módulo a los que otros módulos podrían hacer referencia, junto con sus valores. Por ejemplo, si el módulo consiste en un procedimiento llamado *bigbug*, la tabla de puntos de ingreso contendría la cadena de caracteres "bigbug" seguida de la dirección a la que corresponde. El programador en lenguaje ensamblador indica qué símbolos deben declararse como **puntos de ingreso** empleando una seudoinstrucción como PUBLIC de la figura 7-3.

La tercera parte del módulo objeto consiste en una lista de los símbolos que se usan en el módulo pero se definen en otros módulos, junto con una lista que indica cuáles instrucciones

de máquina usan cuáles símbolos. El enlazador necesita la segunda lista para poder insertar las direcciones correctas en las instrucciones que usan símbolos externos. Un procedimiento puede llamar a otros procedimientos que se tradujeron de forma independiente declarando como externos los nombres de los procedimientos a invocar. El programador en lenguaje ensamblador indica cuáles símbolos se declararán como **símbolos externos** usando una seudoinstrucción como EXTERN de la figura 7-3. En algunas computadoras los puntos de ingreso y las referencias externas se combinan en una sola tabla.

La cuarta parte del módulo objeto es el código ensamblado y las constantes. Esta parte del módulo objeto es la única que se cargará en la memoria para ejecutarse. Las otras cinco partes serán utilizadas por el enlazador y se desecharán antes de que se inicie la ejecución.

La quinta parte del módulo objeto es el diccionario de reubicación. Como se muestra en la figura 7-15, es preciso sumar una constante de reubicación a las instrucciones que contienen direcciones de memoria. Puesto que el enlazador no tiene forma de saber por inspección cuáles de las palabras de datos de la parte 4 contienen instrucciones de máquina y cuáles contienen constantes, esta tabla proporciona información acerca de cuáles direcciones deben reubicarse. La información podría estar en forma de una tabla de bits, con un bit por cada dirección potencialmente reubicable, o una lista explícita de direcciones que deben reubicarse.

La sexta parte es una indicación de fin de módulo, a veces una suma de verificación para detectar errores ocurridos durante la lectura del módulo, y la dirección en la que debe iniciarse la ejecución.

Casi todos los enlazadores requieren dos pasadas. Durante la primera pasada el enlazador lee todos los módulos objeto y construye una tabla de nombres y longitudes de módulos, y una tabla de símbolos global que consiste en todos los puntos de ingreso y referencias externas. Durante la segunda pasada se leen los módulos objeto, se reubican, y se enlazan uno por uno.

7.4.3 Tiempo de ligado y reubicación dinámica

En un sistema de multiprogramación un programa puede leerse de la memoria principal, ejecutarse durante un rato, escribirse en disco, y luego almacenarse nuevamente en la memoria principal para ejecutarse otra vez. En un sistema grande con muchos programas es difícil asegurar que un programa dado se colocará en las mismas localidades en cada ocasión.

La figura 7-17 muestra lo que sucedería si el programa ya reubicado de la figura 7-15(b) se reubicara en la dirección 400 en lugar de la dirección 100 donde el enlazador lo colocó originalmente. Todas las direcciones de memoria son incorrectas; además, la información de reubicación hace mucho que se desechó. Incluso si todavía estuviera disponible dicha información, el costo de reubicar todas las direcciones cada vez que el programa se trae a la memoria sería prohibitivo.

El problema de desplazar programas que ya se enlazaron y reubicaron está íntimamente relacionado con el momento en que se lleva a cabo el ligado final de los nombres simbólicos con direcciones de memoria física absolutas. Cuando se escribe un programa, contiene nombres simbólicos para las direcciones de memoria, por ejemplo, BR L. El momento en que se

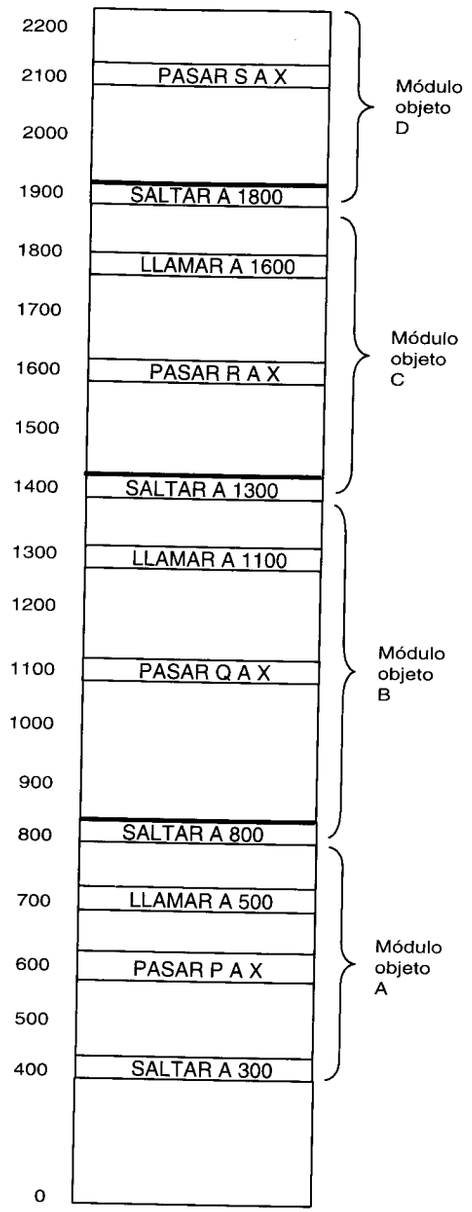


Figura 7-17. El programa binario reubicado de la figura 7-15(b) desplazado hacia arriba 300 direcciones. Ahora muchas instrucciones hacen referencia a una dirección de memoria incorrecta.

determina la dirección real en la memoria principal que corresponde a L se denomina **tiempo de ligado**. Hay por lo menos seis posibilidades para el tiempo de ligado:

1. Cuando se escribe el programa.
2. Cuando se traduce el programa.
3. Cuando se enlaza el programa pero antes de cargarlo.
4. Cuando se carga el programa.
5. Cuando se carga un registro base que se usa para direccionamiento.
6. Cuando se ejecuta la instrucción que contiene la dirección.

Si una instrucción que contiene una dirección de memoria se desplaza después del ligado, será incorrecta (suponiendo que el objeto al que hace referencia también cambió de lugar). Si el traductor produce un binario ejecutable como salida, el ligado habrá ocurrido en el momento de la traducción, y el programa deberá ejecutarse en la dirección en la que el traductor esperaba que se ejecutara. El método de enlazado que se describió en la sección anterior liga los nombres simbólicos a direcciones absolutas durante el enlazado, y es por ello que el desplazamiento de los programas después de enlazados causa errores, como se muestra en la figura 7-17.

Aquí intervienen dos cuestiones relacionadas. La primera es la cuestión de cuándo los nombres simbólicos se ligan a direcciones virtuales. La segunda es cuándo las direcciones virtuales se ligan a direcciones físicas. Sólo si ya se llevaron a cabo ambas operaciones es completo el ligado. Cuando el enlazador fusiona los espacios de direcciones discretos de los módulos objeto en un solo espacio de direcciones lineal, está creando de hecho un espacio de direcciones virtual. La reubicación y el enlazado sirven para ligar nombres simbólicos a direcciones virtuales específicas. Esta observación es cierta sea que se use o no memoria virtual.

Supongamos por el momento que el espacio de direcciones de la figura 7-15(b) se pagina. Es evidente que las direcciones virtuales que corresponden a los nombres simbólicos A, B, C y D ya están determinados, aunque sus direcciones físicas en la memoria principal dependerán del contenido de la tabla de páginas en el momento en que se usen. Un programa binario ejecutable es en realidad un ligado de nombres simbólicos a direcciones virtuales.

Cualquier mecanismo que permita modificar fácilmente la correspondencia entre direcciones virtuales y direcciones físicas de la memoria principal facilitará el desplazamiento de los programas dentro de la memoria principal, aun después de haberse ligado a un espacio de direcciones virtual. Uno de esos mecanismos es la paginación. Una vez que un programa se ha cambiado de lugar en la memoria principal sólo hay que modificar su tabla de páginas, no el programa mismo.

Un segundo mecanismo es el uso de un registro de reubicación de tiempo de ejecución. El CDC 6600 y sus sucesores tenían un registro semejante. En las máquinas que usan esta técnica de reubicación, el registro siempre apunta a la dirección física en la memoria donde comienza el programa actual. El hardware suma a todas las direcciones de memoria el contenido del registro de ubicación antes de enviarlas a la memoria. Todo el proceso de reubicación es transparente para los programas de usuario; ni siquiera saben que está ocurriendo. Cuando un programa cambia de lugar, el sistema operativo debe actualizar el registro de reubicación.

Este mecanismo es menos general que la paginación porque es preciso mover todo el programa como una unidad (a menos que haya registros de reubicación distintos para el código y los datos, como en el Intel 8088, en cuyo caso tendrá que moverse como dos unidades).

En las máquinas que pueden hacer referencia a la memoria relativa al contador de programa se puede usar un tercer mecanismo. Cada vez que un programa cambia de lugar en la memoria principal sólo es preciso actualizar el contador de programa. Un programa en el que todas las referencias a la memoria son relativas al contador de programa o bien absolutas (por ejemplo, registros de dispositivos de E/S en direcciones absolutas) se dice que es **independiente de la posición**. Un procedimiento independiente de la posición se coloca en cualquier lugar del espacio de direcciones virtual sin necesidad de reubicación.

7.4.4 Enlazado dinámico

La estrategia de enlazado que vimos en la sección 7.4.1 tiene la propiedad de que todos los procedimientos que un programa podría invocar se enlazan antes de que el programa pueda iniciar su ejecución. En una computadora con memoria virtual, si se lleva a cabo todo el enlazado antes de iniciar la ejecución no se aprovecharán todas las ventajas de la memoria virtual. Muchos programas tienen procedimientos que sólo se invocan en condiciones extraordinarias. Por ejemplo, los compiladores tienen procedimientos para compilar enunciados que raras veces se usan, además de procedimientos para manejar condiciones de error que casi nunca se presentan.

Una forma más flexible de enlazar procedimientos compilados por separado es enlazar cada procedimiento en el momento en que se invoca por primera vez. Este proceso se llama **enlazado dinámico** y se usó por primera vez en MULTICS, cuya implementación todavía no ha sido superada en algunos aspectos. En las siguientes secciones examinaremos el enlazado dinámico en varios sistemas.

Enlazado dinámico en MULTICS

En la modalidad MULTICS del enlazado dinámico, cada programa está asociado a un segmento, llamado **segmento de enlace**, que contiene un bloque de información para cada procedimiento que podría invocarse. Este bloque de información inicia con una palabra reservada para la dirección virtual del procedimiento y va seguida del nombre del procedimiento, que se almacena como cadena de caracteres.

Cuando se está usando enlazado dinámico, las llamadas a procedimientos en el lenguaje fuente se traducen en instrucciones que direccionan indirectamente la primera palabra del bloque de enlace correspondiente, como se muestra en la figura 7-18(a). El compilador coloca en esta palabra una dirección no válida o bien un patrón de bits especial que hace que ocurra una trampa.

Cuando se invoca un procedimiento que está en un segmento distinto, el intento por direccionar la palabra no válida indirectamente hace que ocurra una trampa al enlazador dinámico. Entonces, el enlazador encuentra la cadena de caracteres en la palabra que sigue a la dirección no válida y busca un procedimiento compilado con este nombre en el directorio

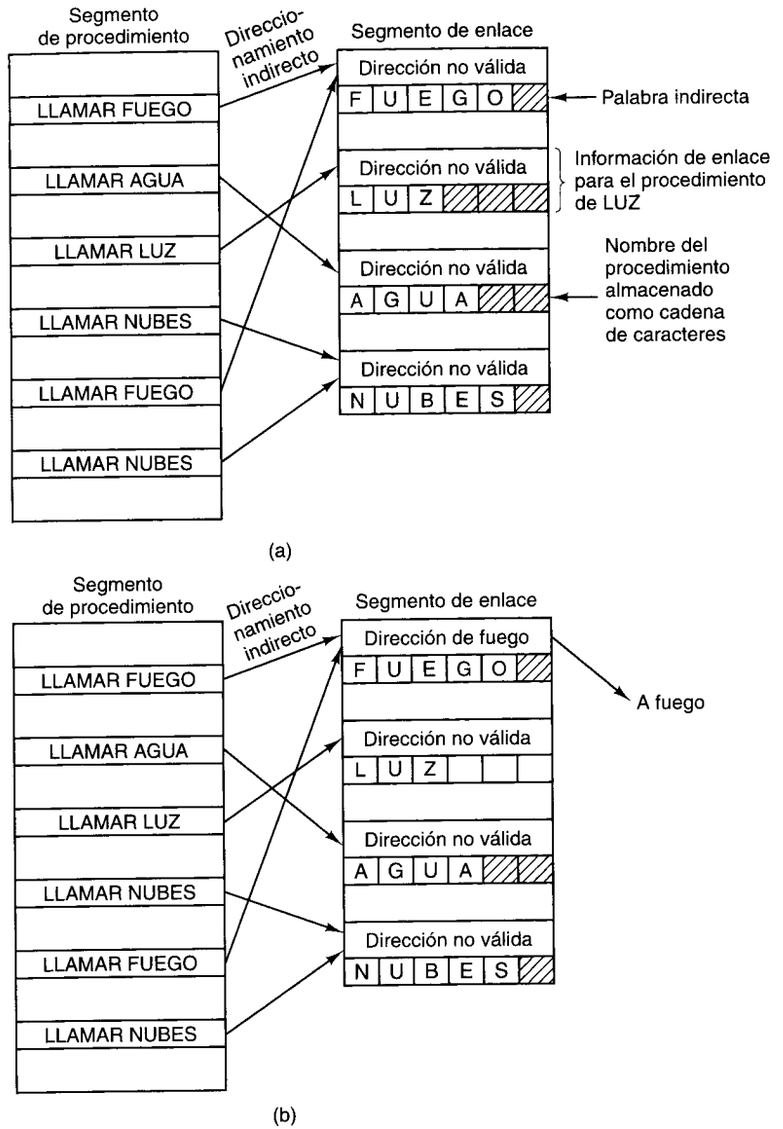


Figura 7-18. Enlazado dinámico. (a) Antes de llamar a *FUEGO*. (b) Después de llamar a *FUEGO* y enlazarlo.

de archivos del usuario. A continuación, se asigna una dirección virtual a ese procedimiento, casi siempre en su propio segmento privado, y esta dirección virtual sobreescribe la dirección no válida del segmento de enlace, como se indica en la figura 7-18(b). Después se vuelve a ejecutar la instrucción que causó el fallo de enlace, y el programa puede continuar a partir del punto en el que estaba antes de la trampa.

Todas las referencias subsecuentes a ese procedimiento se ejecutarán sin causar un fallo de enlace, pues la palabra indirecta ahora contiene una dirección virtual válida. Por consiguiente, el enlazador dinámico sólo se invoca la primera vez que se llama a un procedimiento, y después ya no.

Enlazado dinámico en Windows

Todas las versiones del sistema operativo Windows, incluido NT, manejan enlazado dinámico y se apoyan mucho en él. El enlazado dinámico emplea un formato de archivo especial llamado **DLL** (**biblioteca de enlace dinámico**, *Dynamic Link Library*). Las DLL pueden contener procedimientos, datos o ambas cosas, y es común usarlos para que dos o más procesos puedan compartir procedimientos o datos de biblioteca. Muchas DLL tienen la extensión *.dll*, pero también se usan otras extensiones como *.drv* (para bibliotecas de *drivers*) y *.fon* (para bibliotecas de tipos de letra).

La forma más común de DLL es una biblioteca que consiste en una colección de procedimientos que se pueden cargar en la memoria y a los que varios procesos pueden accesar al mismo tiempo. La figura 7-19 ilustra dos programas que comparten un archivo DLL que contiene cuatro procedimientos, *A*, *B*, *C* y *D*. El programa 1 usa el procedimiento *A*; el programa 2 usa el procedimiento *C*, aunque bien podrían haber usado el mismo procedimiento.

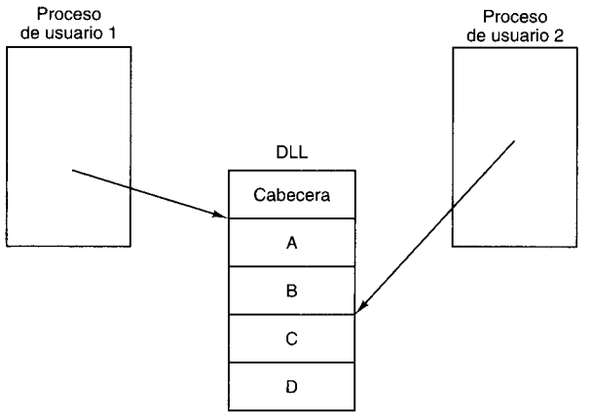


Figura 7-19. Uso de un archivo DLL por dos procesos.

El enlazador construye una DLL a partir de una colección de archivos de entrada. De hecho, la construcción de un archivo DLL se parece mucho a la construcción de un programa binario ejecutable, excepto que se proporciona una bandera especial al enlazador para indicarle que cree una DLL. Las DLL suelen construirse a partir de colecciones de procedimientos de biblioteca que es probable que muchos procedimientos necesiten. Los procedimientos de interfaz con la biblioteca de llamadas del sistema Windows y las grandes bibliotecas de gráficos son ejemplos comunes de DLL. La ventaja de usar las DLL es el ahorro de espacio en la memoria y en el disco. Si se ligara estáticamente una biblioteca común a cada programa

que la usa, aparecería en muchos binarios ejecutables en el disco y en la memoria, y se desperdiciaría espacio. Con las DLL, cada biblioteca aparece una sola vez en el disco y una vez en la memoria.

Además de ahorrar espacio, este enfoque facilita la actualización de los procedimientos de biblioteca aun después de que los programas que los usan han sido compilados y enlazados. En el caso de los paquetes de software comerciales, de los cuales los usuarios casi nunca tienen el código fuente, usar las DLL implica que el proveedor de software puede corregir los errores en las bibliotecas con sólo distribuir archivos DLL nuevos por Internet, sin tener que modificar los binarios del programa principal.

La principal diferencia entre una DLL y un binario ejecutable es que una DLL no puede iniciarse ni ejecutarse sola (porque no tiene programa principal). La información de su cabecera también es diferente. Además, la DLL incluye varios procedimientos extra que no están relacionados con los procedimientos de la biblioteca. Por ejemplo, hay un procedimiento que se invoca automáticamente cada vez que un proceso nuevo se liga a la DLL y otro que se llama automáticamente cada vez que un proceso se desliga de ella. Estos procedimientos pueden apartar y liberar memoria o administrar otros recursos que la DLL necesita.

Un programa se puede ligar a una DLL de dos maneras. En la primera, llamada **enlazado implícito**, el programa del usuario se enlaza estáticamente con un archivo especial llamado **biblioteca de importación** generado por un programa utilitario que extrae cierta información de la DLL. La biblioteca de importación es el mecanismo que permite al programa de usuario acceder a la DLL. Un programa de usuario se puede enlazar a varias bibliotecas de importación. Cuando un programa que usa enlazado implícito se carga en la memoria para ejecutarse, Windows lo examina para ver cuáles DLL usa y verifica si todas ellas ya están en la memoria. Las que no están en la memoria se cargan de inmediato (pero no necesariamente en su totalidad, ya que se paginan). Luego se hacen ciertos cambios a las estructuras de datos de las bibliotecas de importación para poder localizar los procedimientos invocados, algo análogo a los cambios que se muestran en la figura 7-18. Los procedimientos también tienen que mapearse en el espacio de direcciones virtual del programa. En este punto, el programa de usuario está listo para ejecutarse y puede llamar a los procedimientos de las DLL como si se hubieran enlazado estáticamente a él.

La alternativa al enlazado implícito es (¡oh sorpresa!) el **enlazado explícito**. Este enfoque no requiere bibliotecas de importación y no hace que las DLL se carguen al mismo tiempo que el programa de usuario. En vez de ello, el programa de usuario emite una llamada explícita en el momento de la ejecución para ligarse a una DLL, y luego efectúa llamadas adicionales para obtener las direcciones de los procedimientos que necesita. Una vez encontrados, pueden invocarse. Por último, el programa efectúa una llamada final para desligarse de la DLL. Cuando el último proceso se desliga de una DLL, ésta puede removerse de la memoria.

Es importante darse cuenta de que un procedimiento en una DLL no tiene identidad propia (como un hilo o proceso); se ejecuta en el hilo del invocador y usa la pila del invocador para sus variables locales. El procedimiento puede tener datos estáticos específicos para el proceso (así como datos compartidos), y por lo demás se comporta igual que un procedimiento enlazado estáticamente. La única diferencia básica está en la forma en que se efectúa el ligado.

Enlazado dinámico en UNIX

El sistema UNIX tiene un mecanismo idéntico en lo esencial a las DLL de Windows; se llama **biblioteca compartida**. Al igual que un archivo DLL, una biblioteca compartida es un archivo permanente que contiene varios procedimientos o módulos de datos que están presentes en la memoria durante la ejecución y se pueden ligar a varios procesos al mismo tiempo. La biblioteca estándar de C y gran parte del código para trabajo con redes son bibliotecas compartidas.

UNIX sólo maneja enlazado implícito, por lo que una biblioteca compartida consta de dos partes: una **biblioteca anfitriona**, que se enlaza estáticamente con el archivo ejecutable, y una **biblioteca objetivo**, que se llama en el momento de la ejecución. Aunque los detalles difieren, los conceptos son prácticamente idénticos a los de las DLL.

7.5 RESUMEN

Aunque casi todos los programas pueden y deben escribirse en un lenguaje de alto nivel, hay situaciones ocasionales en las que se necesita lenguaje ensamblador, al menos en parte. Los programas para computadoras portátiles escasas de recursos, como las tarjetas inteligentes, procesadores incorporados a aparatos domésticos y asistentes digitales portátiles inalámbricos son posibles candidatos. Un programa en lenguaje ensamblador es una representación simbólica de un programa en lenguaje de máquina subyacente, y se traduce al lenguaje de máquina con un programa llamado ensamblador.

Cuando una ejecución extremadamente rápida es crucial para el éxito de alguna aplicación, una estrategia mejor que escribir todo en lenguaje ensamblador es escribir primero todo el programa en un lenguaje de alto nivel, luego determinar dónde pasa más tiempo el programa, y finalmente reescribir en lenguaje ensamblador sólo aquellas localidades del programa que se usen mucho. En la práctica, una fracción pequeña del código representa una fracción importante del tiempo de ejecución.

Muchos ensambladores tienen un recurso de macros que permite al programador asignar nombres simbólicos a secuencias de código de uso común para después incluirlas. Por lo regular estas macros pueden usar parámetros sin complicación. Las macros se implementan con una especie de algoritmo de procesamiento de cadenas literales.

Casi todos los ensambladores son de dos pasadas. La primera pasada se dedica a construir una tabla de símbolos para etiquetas, literales e identificadores declarados explícitamente. Los símbolos se pueden guardar sin ordenarse y luego buscarse linealmente, ordenarse primero y luego buscarse empleando búsqueda binaria, o procesarse con *hashing (codificación por dispersión)*. Si no es necesario eliminar símbolos durante la primera pasada, el mejor método suele ser el *hashing*. La segunda pasada realiza la generación de código. Algunas seudoinstrucciones se ejecutan durante la primera pasada, y otras, durante la segunda.

Programas ensamblados de forma independiente se pueden enlazar para formar un programa binario ejecutable. El enlazador se encarga de este trabajo. Sus tareas primordiales son la reubicación y el ligado de nombres. El enlazado dinámico es una técnica en la que ciertos procedimientos no se enlazan sino hasta que se llaman realmente. Las DLL de Windows y las bibliotecas compartidas de UNIX usan enlazado dinámico.

PROBLEMAS

1. En cierto programa, el 1% del código ocupa el 50% del tiempo de ejecución. Compare las siguientes tres estrategias respecto al tiempo de programación y el tiempo de ejecución. Suponga que se requerirían 100 meses-programador para escribirlo en C, y que el código de ensamblador es 10 veces más difícil de escribir y cuatro veces más eficiente.

   a. Todo el programa en C.
   b. Todo el programa en lenguaje ensamblador.
   c. Primero todo en C, y luego se reescribe el 1% crítico en ensamblador.

2. ¿Las consideraciones que son válidas para los ensambladores de dos pasadas también son válidas para los compiladores?

   a. Suponga que los compiladores producen módulos objeto, no código de ensamblador.
   b. Suponga que los compiladores producen lenguaje ensamblador simbólico.

3. Sugiera cómo los programadores en lenguaje ensamblador podrían definir sinónimos para códigos de operación. ¿Cómo podría implementarse esto?

4. Todos los ensambladores para las CPU Intel tienen la dirección de destino como primer operando y la dirección de origen como segundo operando. ¿Qué problemas tendrían que resolverse para hacerlo al revés?

5. ¿El siguiente programa puede ensamblarse en dos pasadas? EQU es una seudoinstrucción que iguala el rótulo a la expresión que está en el campo de operandos.

```
A EQU B
B EQU C
C EQU D
D EQU 4
```

6. La Compañía de Software Baratija está planeando producir un ensamblador para una computadora que maneja palabras de 48 bits. A fin de reducir los costos, el gerente del proyecto, el doctor Amarrete, decidió limitar la longitud de los símbolos permitidos de modo que cada símbolo se pueda almacenar en una sola palabra. Amarrete declaró que los símbolos sólo pueden consistir en letras, con excepción de la letra Q, que está prohibida (con objeto de hacer patente ante los clientes su preocupación por la eficiencia). ¿Qué longitud máxima puede tener un símbolo? Describa su esquema de codificación.

7. ¿Qué diferencia hay entre una instrucción y una seudoinstrucción?

8. ¿Qué diferencia hay entre el contador de localidades de instrucciones y el contador de programa, si es que la hay? Después de todo, los dos siguen la pista a la siguiente instrucción de un programa.

9. Muestre la tabla de símbolos después de encontrarse los siguientes enunciados para el Pentium II. El primer enunciado se asigna a la dirección 1000.

```
EVEREST:    POP BX          (1 BYTE)
K2:         PUSH BP         (1 BYTE)
WHITNEY:    MOV BP,SP       (2 BYTES)
MCKINLEY:   PUSH X          (3 BYTES)
FUJI:       PUSH SI         (1 BYTE)
KIBO:       SUB SI,300      (3 BYTES)
```


10. ¿Puede imaginar circunstancias en las que un lenguaje ensamblador permita que una etiqueta sea igual a un código de operación (por ejemplo, *MOV* como rótulo)? Comente.

11. Muestre los pasos necesarios para encontrar Berkeley usando búsqueda binaria con la siguiente lista: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood y Yellow Springs. Al calcular el elemento de la mitad de una lista con número par de elementos, use el elemento que está inmediatamente después del índice de en medio.

12. ¿Es posible usar búsqueda binaria con una tabla cuyo tamaño es primo?

13. Calcule el código *hash* para cada uno de los siguientes símbolos sumando todas las letras (A = 1, B = 2, etc.) y obteniendo el residuo de dividir el resultado entre el tamaño de la tabla *hash*. La tabla tiene 19 entradas, numeradas del 0 al 18.

    els, jan, jelle, maaike

    ¿Cada símbolo genera un valor *hash* único? Si no es así, ¿cómo puede resolverse el problema de la colisión?

14. El método de codificación por *hashing* descrito en el texto vincula todas las entradas que tienen el mismo código *hash* en una lista enlazada. Un método alternativo es tener una sola tabla con n entradas, cada una de las cuales tiene espacio para una clave y su valor (o apuntadores a ellos). Si el algoritmo de *hashing* genera una entrada que ya está llena, se usa un segundo algoritmo de *hashing* para intentarlo otra vez. Si esa entrada también está llena, se usa otro algoritmo, y así hasta encontrar una entrada vacía. Si la fracción de las entradas llenas es R, ¿cuántos intentos se necesitarán, en promedio, para introducir un símbolo nuevo?

15. A medida que avanza la tecnología, podría llegar el día en que sea posible colocar miles de CPU idénticas en un chip, cada una con unas cuantas palabras de memoria local. Si todas las CPU pueden leer y escribir tres registros compartidos, ¿cómo podría implementarse una memoria asociativa?

16. El Pentium II tiene una arquitectura segmentada, con varios segmentos independientes. Un ensamblador para esta máquina bien podría tener una seudoinstrucción SEG N que ordenaría al ensamblador colocar el código y los datos subsecuentes en el segmento N. ¿Un esquema así afectaría al ILC?

17. Los programas a menudo se enlazan con varias DLL. ¿No sería más eficiente poner todos los procedimientos en una DLL grande y luego enlazarse con ella?

18. ¿Una DLL se puede mapear con los espacios de direcciones virtuales de dos procesos en diferentes direcciones virtuales? Si así se hiciera, ¿qué problemas podrían surgir? ¿Se pueden resolver? Si no, ¿qué puede hacerse para eliminarlos?

19. Una forma de efectuar enlazado (estático) es la siguiente. Antes de explorar la biblioteca, el enlazador construye una lista de los procedimientos requeridos, es decir, de nombres definidos como EXTERN en los módulos que se están enlazando. Luego el enlazador recorre la biblioteca linealmente, extrayendo todos los procedimientos que están en la lista de nombres requeridos. ¿Funciona este plan? Si no, ¿por qué no y cómo puede remediarse?

20. ¿Un registro puede usarse como parámetro real en una llamada a macro? ¿Y una constante? ¿Por qué sí o por qué no?

21. Usted debe implementar un ensamblador con macros. Por razones estéticas, su jefe ha decidido que las definiciones de las macros no tienen que preceder a sus llamadas. ¿Qué implicaciones tiene esta decisión para la implementación?

22. Piense en una forma de poner un ensamblador con macros en un ciclo infinito.

23. Un enlazador lee cinco módulos, cuyas longitudes son 200, 800, 600, 500 y 700 palabras, respectivamente. Si los módulos se cargan en ese orden, ¿cuáles son las constantes de reubicación?

24. Escriba un paquete de tabla de símbolos que conste de dos rutinas: *ingresar(simbolo, valor)* y *buscar(simbolo, valor)*. La primera ingresa símbolos nuevos en la tabla y la segunda los busca. Use alguna forma de codificación *hash*.

25. Escriba un ensamblador sencillo para la computadora Mic-1 del capítulo 4. Además de procesar las instrucciones de máquina, incluya un mecanismo para asignar constantes a símbolos durante el ensamblado, y una forma de ensamblar una constante en una palabra de máquina.

26. Añada un recurso de macros sencillo al ensamblador del problema anterior.

8

ARQUITECTURAS DE COMPUTADORAS PARALELAS

Aunque las computadoras cada vez son más rápidas, lo que se les exige está creciendo por lo menos al mismo ritmo. Los astrónomos quieren simular toda la historia del Universo, desde el *big bang* hasta que acabe la función. A los ingenieros farmacéuticos les encantaría poder diseñar medicamentos a la medida de enfermedades específicas en sus computadoras en lugar de tener que sacrificar legiones de ratas. Los diseñadores de aviones podrían crear productos con mejor rendimiento de combustible si las computadoras pudieran efectuar todo el trabajo, sin necesidad de construir prototipos para túneles de viento. En pocas palabras, por más potencia de cómputo que tengamos al alcance, para muchos usuarios, sobre todo en las ciencias, la ingeniería y la industria, nunca es suficiente.

Aunque la frecuencia de reloj es cada vez más alta, la velocidad de los circuitos no se puede incrementar indefinidamente. La velocidad de la luz ya representa un problema importante para los diseñadores de computadoras del extremo superior, y las posibilidades de lograr que los electrones y fotones se muevan a mayor velocidad no son muy halagüeñas. Los problemas de disipación de calor están convirtiendo a las supercomputadoras en acondicionadores de aire de vanguardia. Por último, aunque los transistores son cada vez más pequeños, llegará el momento en que cada transistor tendrá tan pocos átomos que los efectos de la mecánica cuántica (por ejemplo, el principio de incertidumbre de Heisenberg) podrían convertirse en un problema importante.

Por ello, para poder atacar problemas cada vez más grandes, los arquitectos de computadoras están recurriendo a las computadoras paralelas. Aunque tal vez nunca sea posible construir una computadora con una sola CPU y un tiempo de ciclo de 0.001 ns, bien

podría ser factible construir una con 1000 CPU, cada una de ellas con un tiempo de ciclo de 1 ns. Si bien este último diseño usa CPU más lentas que el primero, en teoría su capacidad de cómputo total es la misma. Es en esto en lo que están fincadas las esperanzas.

Podemos introducir paralelismo en diversos niveles. En el nivel de instrucciones, por ejemplo, el uso de filas de procesamiento y los diseños superescalares pueden elevar el desempeño en un factor de 10 respecto a los diseños puramente secuenciales. Sin embargo, para lograr un mejoramiento por un factor de cien o mil o un millón es necesario repetir CPU enteras, o al menos porciones sustanciales de ellas, y lograr que todas colaboren de forma eficiente. Aquí es donde está el reto.

En este capítulo examinaremos los principios de diseño de computadoras paralelas y estudiaremos varios ejemplos. Todas estas máquinas consisten en elementos de procesamiento y elementos de memoria. Los diseños difieren en la cantidad de cada elemento que está presente, en su constitución y en su interconexión. Algunos diseños emplean un número relativamente bajo de componentes de gran potencia, mientras que otras usan cantidades enormes de componentes más débiles. También son comunes los diseños intermedios. Se ha trabajado mucho en el área de las arquitecturas paralelas; en este capítulo apenas podemos rascar la superficie. El lector puede encontrar información adicional en (Loshin, 1994; Pfister, 1998; Sima *et al.*, 1997; y Wilkinson, 1994).

8.1 ASPECTOS DEL DISEÑO DE COMPUTADORAS PARALELAS

Al analizar un nuevo sistema de cómputo paralelo, las tres preguntas fundamentales que debemos hacer son:

1. ¿Cuántos elementos de procesamiento hay, de qué tamaño y de qué tipo?
2. ¿Cuántos módulos de memoria hay, de qué tamaño y de qué tipo?
3. ¿Cómo se interconectan los elementos de procesamiento y de memoria?

Examinemos brevemente cada uno de estos puntos. Los elementos de procesamiento pueden variar desde las ALU mínimas hasta las CPU completas, con tamaños que van desde una pequeña fracción de un chip hasta un metro cúbico de circuitos electrónicos por elemento. Como cabría esperar, cuando el elemento de procesamiento es una fracción de un chip, es posible equipar una computadora con una gran cantidad de tales elementos, quizá hasta un millón de ellos. Si el elemento de procesamiento es una computadora completa, con su propia memoria y equipo de E/S, el número de elementos es naturalmente más reducido, aunque se han instalado sistemas con casi 10,000 CPU. Cada vez se están construyendo más computadoras paralelas a partir de piezas comerciales, sobre todo CPU. Las capacidades y limitaciones de estas piezas a menudo influyen considerablemente en el diseño.

Los sistemas de memoria a menudo se dividen en módulos que operan de forma independiente unos de otros, en paralelo para que muchas CPU puedan accesar a ellos al mismo tiempo. Dichos módulos pueden ser pequeños (kilobytes) o grandes (megabytes) y estar integrados a la CPU o situados en una tarjeta de circuitos distinta. Puesto que las memorias dinámicas (DRAM) grandes suelen ser mucho más lentas que las CPU, es común utilizar

complejos esquemas de caché para agilizar el acceso a la memoria. Con frecuencia se usan dos, tres y hasta cuatro niveles de caché.

Aunque existe cierta variación entre los diseños de CPU y de memoria, el área en la que más difieren los sistemas paralelos es en la forma en que se combinan las piezas. Los esquemas de interconexión se pueden dividir a grandes rasgos en dos categorías: estáticos y dinámicos. Los esquemas estáticos simplemente conectan todos los componentes en una configuración fija, como estrella, anillo o cuadrícula. En los esquemas de interconexión dinámica todas las piezas se conectan a una red de conmutación que puede encaminar dinámicamente mensajes entre los componentes. Los dos esquemas tienen puntos fuertes y débiles, como en breve veremos.

Ver las computadoras paralelas como una colección de chips conectados entre sí de una forma u otra es básicamente una perspectiva ascendente del mundo. En un enfoque descendente la pregunta que se haría sería: "¿Qué es lo que se va a ejecutar en paralelo?" Aquí también tenemos toda una gama de posibilidades. Algunas computadoras paralelas se diseñan de modo que puedan ejecutar varios trabajos independientes al mismo tiempo. Dichos trabajos nada tienen que ver unos con otros y no se comunican. Un ejemplo típico es una computadora con entre 8 y 64 CPU creada para un gran sistema UNIX de tiempo compartido que atiende a miles de usuarios remotos. Los sistemas de procesamiento de transacciones de los bancos (por ejemplo, cajeros automáticos), líneas aéreas (por ejemplo, sistemas de reservaciones) y grandes servidores de Web también pertenecen a esta categoría, lo mismo que las simulaciones independientes que se ejecutan de forma concurrente empleando diferentes conjuntos de parámetros.

En una zona diferente de este espectro se encuentran las computadoras paralelas que se usan para ejecutar un solo trabajo que consiste en muchos procesos paralelos. Por ejemplo, consideremos un programa de ajedrez que analiza un tablero dado generando una lista de movimientos válidos y luego genera procesos paralelos para analizar (recursivamente) cada nuevo tablero en paralelo. Lo que se busca con el paralelismo aquí no es atender a más usuarios, sino resolver con mayor rapidez un solo problema.

Pasando a otra zona del espectro, nos encontramos con máquinas en las que el paralelismo proviene del uso intensivo de filas de procesamiento o de muchas ALU que operan con el mismo flujo de instrucciones al mismo tiempo. Las supercomputadoras numéricas con hardware especial para procesar vectores pertenecen a esta categoría. Aquí no sólo se está resolviendo un solo problema principal, sino que todas las partes de la computadora están trabajando en muy estrecha colaboración en casi el mismo aspecto del problema (por ejemplo, diferentes elementos de los mismos dos vectores se están sumando en paralelo).

Aunque es difícil definirlo con exactitud, estos tres ejemplos difieren en lo que algunos llaman **tamaño de grano**. En los sistemas de tiempo compartido con múltiples CPU, la unidad de paralelismo es grande: todo un programa de usuario. La ejecución de programas grandes en paralelo con poca o ninguna comunicación entre los programas es **paralelismo de grano grueso**. El extremo opuesto, que ilustramos con el procesamiento de vectores, se llama **paralelismo de grano fino**.

El tamaño de grano se refiere a los algoritmos y al software, pero tiene un análogo directo en el hardware. Los sistemas con un número reducido de CPU grandes e independientes que tienen conexiones de baja velocidad entre sí se dice que están **débilmente acoplados**. Lo

opuesto son los sistemas **fuertemente acoplados**, en los que los componentes generalmente son más pequeños, más juntos e interactúan unos con otros con frecuencia a través de redes de comunicación con gran ancho de banda. En casi todos los casos, los problemas que tienen paralelismo de grano grueso se resuelven mejor en sistemas débilmente acoplados; así mismo, los problemas con paralelismo de grano fino se resuelven mejor en sistemas fuertemente acoplados. No obstante, es tanta la variedad de algoritmos, software y hardware, que en el mejor de los casos ésta es sólo una guía general. La amplia gama de tamaños de grano de los problemas y de posibilidades para el acoplamiento de los sistemas ha dado pie a la amplia variedad de arquitecturas que estudiaremos en este capítulo.

En las secciones que siguen examinaremos algunos de los aspectos del diseño de computadoras paralelas, comenzando con los modelos de comunicación y las redes de interconexión. Luego analizaremos cuestiones de desempeño y de software. Por último, concluiremos con una taxonomía de las arquitecturas de computadoras paralelas que determinará la organización del resto del capítulo.

8.1.1 Modelos de comunicación

En cualquier sistema de cómputo paralelo, las CPU que trabajan en diferentes partes del mismo problema se deben comunicar entre sí para intercambiar información. La forma precisa en la que lo deben hacer es tema de muchos debates en la comunidad arquitectónica. Se han propuesto e implementado dos diseños distintos, multiprocesadores y multicomputadoras. En esta sección examinaremos ambos diseños.

Multiprocesadores

En el primer diseño, todas las CPU comparten una misma memoria física, como se ilustra en la figura 8-1(a). Un sistema basado en memoria compartida como éste se llama **multiprocesador** o a veces simplemente **sistema con memoria compartida**.

El modelo del multiprocesador se extiende al software. Todos los procesos que están ejecutándose en un multiprocesador pueden compartir un solo espacio de direcciones virtual mapeado en la memoria común. Cualquier proceso puede leer o escribir una palabra de memoria con sólo ejecutar una instrucción LOAD o STORE. No se necesita nada más. Dos procesos pueden comunicarse con sólo hacer que uno de ellos escriba datos en la memoria y que el otro los lea después.

La capacidad de dos (o más) procesos para comunicarse con sólo leer y escribir en la memoria es la razón por la que los multiprocesadores son tan populares. Se trata de un modelo fácil de entender para los programadores que se puede aplicar a una amplia gama de problemas. Consideremos, por ejemplo, un programa que examina una imagen de mapa de bits y enumera todos los objetos que contiene. Una copia de la imagen se guarda en la memoria, como se muestra en la figura 8-1(b). Cada una de las 16 CPU ejecuta un solo proceso, al cual se ha asignado una de las 16 secciones para que la analice. Sin embargo, cada proceso tiene acceso a toda la imagen, lo cual es indispensable porque algunos objetos ocupan varias secciones. Si un proceso descubre que uno de sus objetos rebasa la frontera de su sección, simplemente sigue al objeto a la siguiente sección leyendo las palabras de esa sección. En este ejemplo, varios procesos descubrirán el mismo objeto, por lo que se requiere cierta coordinación al final para determinar cuántas, casas, árboles y aviones hay.

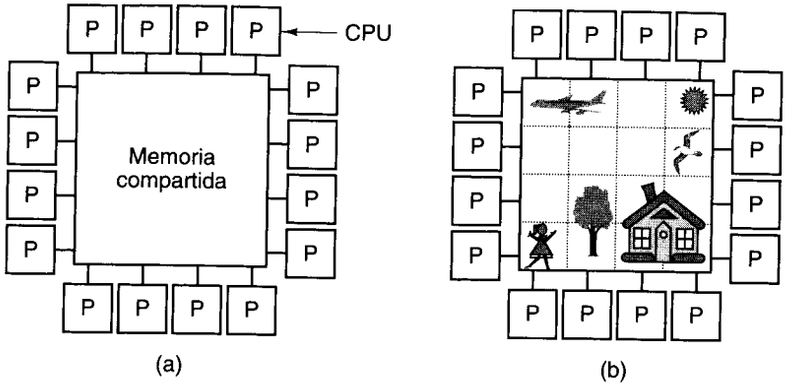


Figura 8-1. (a) Multiprocesador con 16 CPU que comparten una misma memoria. (b) Imagen dividida en 16 secciones, cada una de las cuales se analiza con una CPU distinta.

Muchos fabricantes de computadoras venden multiprocesadores. Como ejemplos podemos citar el Sun Enterprise 10000, el Sequent NUMA-Q, el SGI Origin 2000 y el HP/Convex Exemplar.

Multicomputadoras

El segundo diseño de arquitectura paralela es aquel en el que cada CPU tiene su propia memoria privada, accesible únicamente a ella y a ninguna otra CPU. Un diseño así se llama **multicomputadora** o **sistema de memoria distribuida**, y se ilustra en la figura 8-2(a). Las multicomputadoras suelen estar débilmente acopladas (aunque no siempre). El aspecto clave de una multicomputadora que la distingue de un multiprocesador es que cada CPU de una multicomputadora tiene su propia memoria local privada a la que puede acceder con sólo ejecutar instrucciones LOAD y STORE, pero a la que ninguna otra CPU puede acceder usando esas instrucciones. Así pues, los multiprocesadores tienen un solo espacio de direcciones físico compartido por todas las CPU, mientras que las multicomputadoras tienen un espacio de direcciones físico por CPU.

Puesto que las CPU de una multicomputadora no pueden comunicarse con sólo leer y escribir en la memoria común, necesitan un mecanismo de comunicación distinto. Lo que hacen es pasarse mensajes entre sí utilizando la red de interconexión. Como ejemplos de multicomputadoras podemos mencionar al IBM SP/2, la Intel/Sandia Option Red y la Wisconsin COW.

La ausencia de memoria compartida en hardware en una multicomputadora tiene importantes implicaciones para la estructura del software. Tener un solo espacio de direcciones virtual en el que todos los procesos pueden leer y escribir con sólo ejecutar instrucciones LOAD

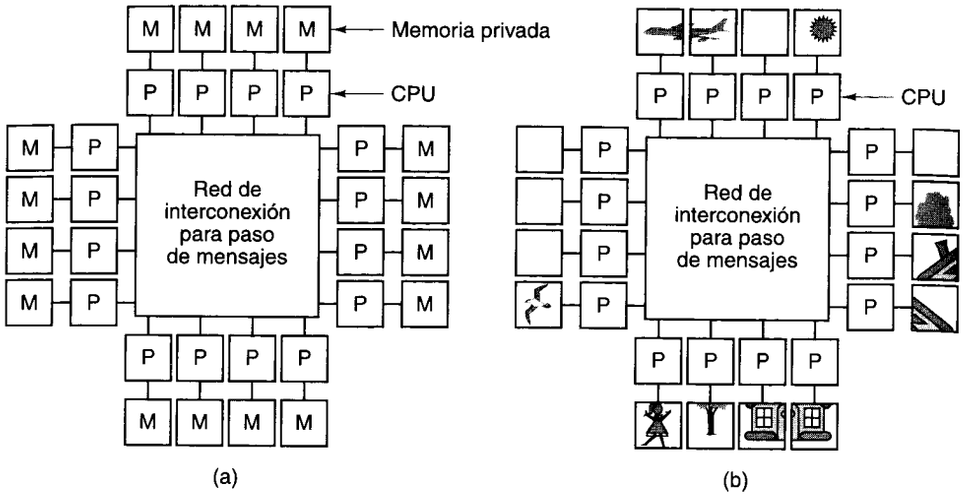


Figura 8-2. (a) Multicomputadora con 16 CPU, cada una de las cuales tiene su propia memoria privada. (b) La imagen de mapa de bits de la figura 8-1 dividida entre las 16 memorias.

y STORE es imposible en una multicomputadora. Por ejemplo, si la CPU 0 (la de la esquina superior izquierda) de la figura 8-1(b) descubre que parte de su objeto se extiende hacia la sección asignada a la CPU 1, podrá seguir leyendo la memoria para acceder a la cola del avión. Por otra parte, si la CPU 0 de la figura 8-2(b) efectúa el mismo descubrimiento, no puede leer simplemente la memoria de la CPU 1; tiene que hacer algo muy distinto para obtener los datos que necesita.

En particular, la CPU 0 tiene que descubrir (de alguna manera) cuál CPU tiene los datos que necesita y enviar a esa CPU un mensaje en el que solicita una copia de los datos. Por lo regular, lo que haría entonces la CPU 0 sería bloquearse (es decir, esperar) hasta recibir una respuesta a su solicitud. Cuando el mensaje llega a la CPU 1, el software de esa CPU debe analizarlo y devolver los datos solicitados. Cuando el mensaje de respuesta llega a la CPU 0, el software se desbloquea y puede continuar su ejecución.

En una multicomputadora, la comunicación entre procesos a menudo utiliza primitivas en software como send (enviar) y receive (recibir). Esto hace que el software tenga una estructura diferente, mucho más complicada, que en un multiprocesador, y también implica que la división correcta de los datos y su colocación en los lugares óptimos es una cuestión importante en una multicomputadora. Esto es menos importante en un multiprocesador porque la colocación no afecta la corrección ni la programabilidad, aunque sí podría afectar el desempeño. En pocas palabras, programar una multicomputadora es mucho más difícil que programar un multiprocesador.

En estas condiciones, ¿para qué construir multicomputadoras, si los multiprocesadores son más fáciles de programar? La respuesta es sencilla: las multicomputadoras grandes son mucho más sencillas y económicas de construir que los multiprocesadores con el mismo

número de CPU. Implementar una memoria compartida por unos cuantos cientos de CPU es una empresa de gran magnitud, pero es relativamente fácil construir una multicomputadora con 10,000 CPU o más.

Así pues, tenemos un dilema: los multiprocesadores son difíciles de construir pero fáciles de programar, mientras que las multicomputadoras son fáciles de construir pero difíciles de programar. Esta observación ha dado pie a muchos intentos por construir sistemas híbridos que sean relativamente fáciles de construir y relativamente fáciles de programar. Tales trabajos han conducido a la revelación de que hay varias formas de implementar una memoria compartida, cada una con sus ventajas y desventajas. De hecho, gran parte de las investigaciones actuales sobre arquitecturas paralelas tiene que ver con la convergencia de las arquitecturas de multiprocesador y multicomputadora en formas híbridas que combinan las ventajas de cada una. El Santo Grial aquí es encontrar diseños **escalables**, es decir, que sigan funcionando bien a medida que se añaden más y más CPU.

Una estrategia para construir sistemas híbridos se basa en el hecho de que los sistemas de cómputo modernos no son monolíticos sino que se construyen en una serie de capas: el tema de este libro. Este concepto abre la posibilidad de implementar la memoria compartida en cualquiera de varios niveles, como se muestra en la figura 8-3. En la figura 8-3(a) vemos cómo el hardware implementa la memoria compartida como verdadero multiprocesador. En este diseño, hay una sola copia de sistema operativo con un solo conjunto de tablas, en particular la tabla de asignación de memoria. Cuando un proceso necesita más memoria, transfiere el control al sistema operativo a través de una trampa, éste busca en su tabla una página libre y hace corresponder la página con el espacio de direcciones del proceso. En lo que al sistema operativo concierne, hay una sola memoria y él lleva la contabilidad de qué proceso posee qué página en software. Hay muchas formas de implementar la memoria compartida en hardware, como veremos más adelante.

Una segunda posibilidad es usar hardware de multicomputadora y hacer que el sistema operativo simule la memoria compartida proporcionando un solo espacio de direcciones virtual paginado compartido para todo el sistema. En esta estrategia, llamada **memoria compartida distribuida** (DSM, *Distributed Shared Memory*) (Li, 1988; Li y Hudak, 1986, 1989), cada página se ubica en una de las memorias de la figura 8-2(a). Cada máquina tiene su propia memoria virtual y sus propias tablas de páginas. Cuando una CPU efectúa una LOAD o STORE con una página que no tiene, ocurre una trampa al sistema operativo. Éste localiza entonces la página y pide a la CPU que actualmente la tiene que anule su mapeo y envíe la página por la red de interconexión. Cuando la página llega, se mapea y la instrucción que causó el fallo se reinicia. En realidad, lo que el sistema operativo está haciendo es atender los fallos de página con la memoria remota en lugar de con el disco. Para el usuario, parece como si la máquina tuviera memoria compartida. Examinaremos DSM más adelante en este capítulo.

Una tercera posibilidad es hacer que un sistema de tiempo de ejecución en el nivel de usuario implemente una forma (tal vez específica para el lenguaje) de memoria compartida. Con esta estrategia, el lenguaje de programación proporciona algún tipo de abstracción de memoria compartida, que luego el compilador y el sistema de tiempo de ejecución implementan. Por ejemplo, el modelo Linda se basa en la abstracción de un espacio de tuplas (registros de datos que contienen una colección de campos) compartido. Los procesos de cualquier

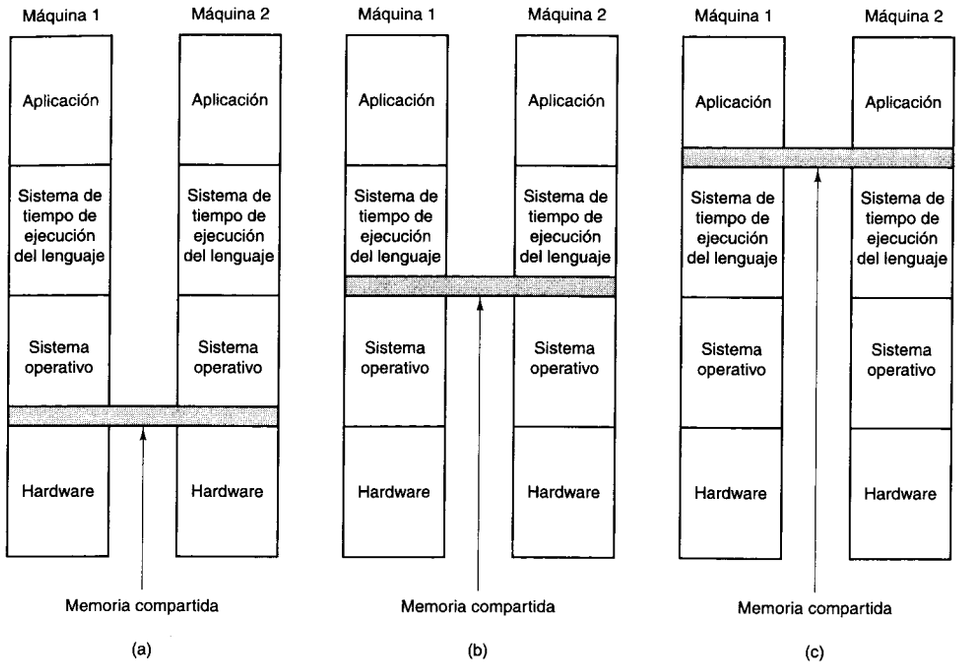


Figura 8-3. Diversas capas en las que puede implementarse la memoria compartida. (a) El hardware. (b) El sistema operativo. (c) El sistema de tiempo de ejecución del lenguaje.

máquina pueden traer una tupla del espacio de tuplas compartido o enviar una tupla a ese espacio. Dado que el acceso al espacio de tuplas se controla totalmente en software (por el sistema de tiempo de ejecución Linda), no se requiere hardware especial ni apoyo del sistema operativo.

Otro ejemplo de memoria compartida específica para un lenguaje implementada por el sistema de tiempo de ejecución es el modelo Orca de objetos de datos compartidos. En Orca, los procesos comparten objetos genéricos, no sólo tuplas, y pueden ejecutar métodos específicos para cada objeto con ellos. Cuando un método modifica el estado interno de un objeto, corresponde al sistema de tiempo de ejecución asegurarse de que todas las copias del objeto en todas las máquinas se actualicen simultáneamente. Una vez más, puesto que los objetos son un concepto estrictamente de software, el sistema de tiempo de ejecución puede efectuar totalmente la implementación, sin ayuda del hardware ni del sistema operativo. Examinaremos tanto a Linda como a Orca más adelante en este capítulo.

8.1.2 Redes de interconexión

En la figura 8-2 vimos que las multicomputadoras deben su cohesión a las redes de interconexión. Ha llegado el momento de examinar con mayor detenimiento esas redes. Resulta interesante y sorprendente lo parecido que son los multiprocesadores y las multicomputadoras

en este sentido, porque los multiprocesadores a menudo tienen varios módulos de memoria que también deben conectarse entre sí y con las CPU. Por ello, el material de esta sección a menudo se aplica a ambos tipos de sistemas.

La razón fundamental por la que las redes de interconexión de los multiprocesadores y las multicomputadoras son similares es que en lo más fundamental ambas usan transferencia de mensajes. Incluso en una máquina con una sola CPU, si el procesador necesita leer o escribir una palabra, lo que normalmente hace es acertar ciertas líneas del bus y esperar una respuesta. Esta acción es fundamentalmente similar a la transferencia de mensajes: el iniciador envía una solicitud y espera una respuesta. En los multiprocesadores grandes, la comunicación entre las CPU y la memoria remota casi siempre consiste en que la CPU envía un mensaje explícito, llamado **paquete**, a la memoria solicitándole ciertos datos, y la memoria devuelve un paquete de respuesta.

Las redes de interconexión pueden tener hasta cinco componentes:

1. CPU.
2. Módulos de memoria.
3. Interfaces.
4. Enlaces.
5. Conmutadores.

Ya examinamos las CPU y las memorias con cierto detalle en otros capítulos y no lo haremos más aquí. Básicamente, se trata de los puntos finales de toda la comunicación. Las interfaces son los dispositivos que sacan los mensajes de las CPU y las memorias y los introducen en ellas. En muchos diseños, una interfaz es un chip o tarjeta que está conectada al bus local de cada CPU y puede hablar con ella y con la memoria local, si la hay. Es común que la interfaz incluya un procesador programable así como algo de RAM privada. Por lo regular, la interfaz puede leer y escribir en diversas memorias, a fin de trasladar bloques de datos de un lugar a otro.

Los enlaces son los canales físicos por los cuales se desplazan los bits; pueden ser eléctricos o de fibra óptica y seriales (anchura de un bit) o paralelos (anchura de más de un bit). Cada enlace tiene un ancho de banda máximo, que es el número de bits por segundo que puede transferir. Los enlaces pueden ser símplex (unidireccionales), semidúplex (en un sentido a la vez) o dúplex (en ambos sentidos a la vez).

Los conmutadores son dispositivos con varios puertos de entrada y varios puertos de salida. Cuando un paquete llega a un conmutador por un puerto de entrada, ciertos bits del paquete se usan para seleccionar el puerto de salida al que se enviará el paquete. Un paquete podría tener sólo 2 o 4 bytes, pero también podría ser mucho más largo (digamos, 8 KB).

Existe cierta analogía entre una red de interconexión y las calles de una ciudad. Las calles son como enlaces; cada una tiene una direccionalidad (de uno o dos sentidos), una tasa de datos máxima (límite de velocidad) y una anchura (número de carriles). Las intersecciones

son como conmutadores. En cada intersección, un paquete que llega (peatón o vehículo) puede escoger cuál puerto de salida (calle) usará, dependiendo de cuál es su destino final.

Al diseñar o analizar una red de interconexión, varias áreas destacan por su importancia. En primer lugar está la cuestión de topología, es decir, la forma en que están dispuestos los componentes. En segundo lugar está la forma en que el conmutador funciona y cómo maneja la lucha por recursos. En tercer lugar está el algoritmo de enrutamiento que se usa para que los mensajes lleguen a su destino de forma eficiente. A continuación examinaremos brevemente cada uno de estos temas.

Topología

La topología de una red de interconexión describe la forma como están dispuestos los enlaces y los conmutadores; por ejemplo, en forma de anillo o de cuadrícula. Los diseños topológicos pueden modelarse como grafos, en los que los enlaces son aristas y los conmutadores son nodos, como se muestra en la figura 8-4. Cada nodo de una red de interconexión (o de su grafo) tiene cierto número de enlaces conectados a él. Los matemáticos llaman al número de enlaces **grado** del nodo; los ingenieros lo llaman **abanico de salida**. En general, cuanto mayor es el abanico de salida, más opciones de enrutamiento hay y mayor es la tolerancia de fallos, es decir, la capacidad para seguir funcionando aunque falle un enlace, desviando el tráfico por otros caminos. Si cada nodo tiene k aristas y el cableado es el debido, es posible diseñar la red de modo que permanezca plenamente conectada aunque fallen $k - 1$ enlaces.

Otra propiedad de una red de interconexión (o su grafo) es su **diámetro**. Si medimos la distancia entre dos nodos por el número de aristas que es preciso recorrer para llegar de uno al otro, el título de una gráfica es la distancia entre los dos nodos que están más separados (es decir, entre los que la distancia es máxima). El diámetro de una red de interconexión se relaciona con el retraso de peor caso al enviar paquetes de una CPU a otra o de una CPU a la memoria porque cada brinco por un enlace toma una cantidad de tiempo finita. Cuanto más pequeño sea el diámetro, mejor será el desempeño de peor caso. Otra cosa importante es la distancia media entre dos nodos, la cual se relaciona con el tiempo de tránsito medio de los paquetes.

Otra propiedad importante de una red de interconexión es su capacidad de transmisión, es decir, cuántos datos puede transferir por segundo. Una medida útil de esta capacidad es el **ancho de banda bisectriz**. Para calcular esta cantidad, primero tenemos que dividir (conceptualmente) la red en dos partes iguales (en términos del número de nodos) pero disconexas eliminando un conjunto de aristas de su grafo. Luego calculamos el ancho de banda total de las aristas que se eliminaron. Puede haber muchas formas distintas de dividir la red en dos partes iguales. El ancho de banda bisectriz es el mínimo de todas las divisiones posibles. La importancia de este número radica en que si el ancho de banda bisectriz es, digamos, de 800 bits/s, entonces si hay mucha comunicación entre las dos mitades el rendimiento total podría estar limitado a sólo 800 bits/s, en el peor de los casos. Muchos diseñadores creen que el ancho de banda bisectriz es la métrica más importante de una red de interconexión. Muchas redes de interconexión se diseñan con el objetivo de maximizar el ancho de banda bisectriz.

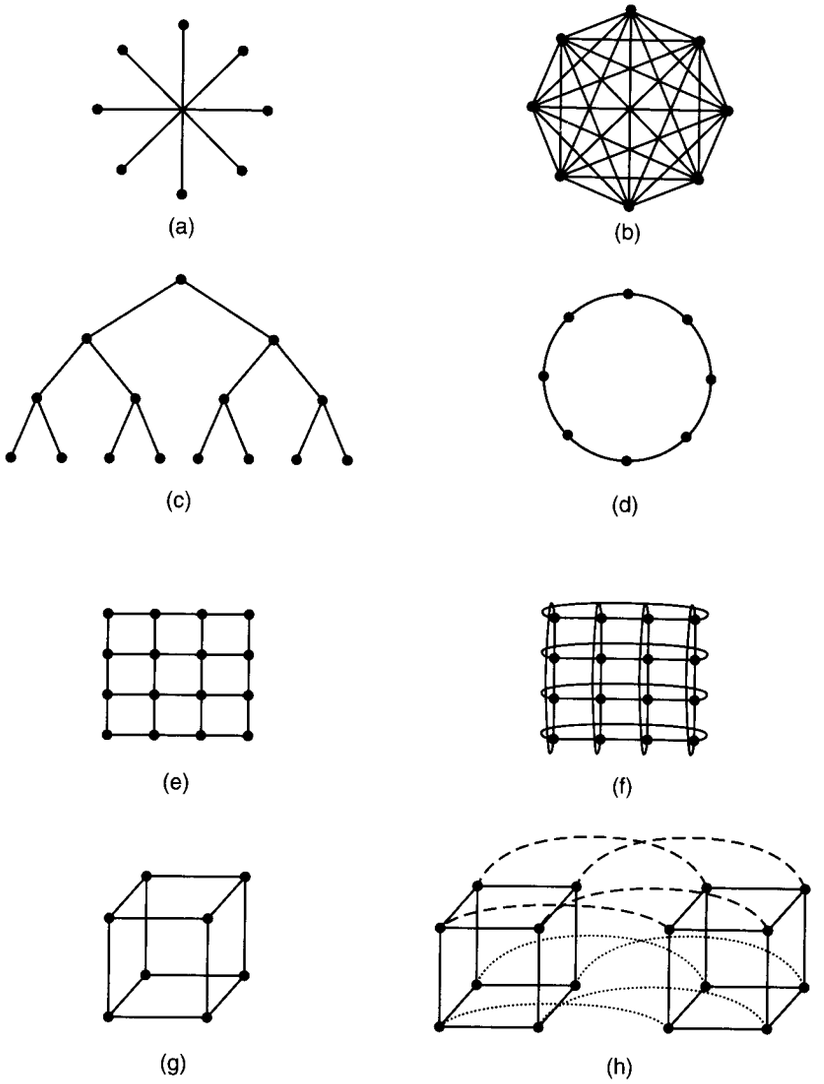


Figura 8-4. Diversas topologías. Los puntos gruesos representan conmutadores. No se muestran las CPU ni las memorias. (a) Una estrella. (b) Una interconexión total. (c) Un árbol. (d) Un anillo. (e) Una cuadrícula. (f) Un toroide doble. (g) Un cubo. (h) Un hipercubo 4D.

Las redes de interconexión se pueden caracterizar por su **dimensionalidad**. Para nuestros fines, la dimensionalidad está determinada por el número de opciones que hay para llegar del origen al destino. Si nunca hay opciones (es decir, sólo hay un camino de cada origen a cada destino), la red es cero-dimensional. Si hay una dimensión en la que se puede tomar una decisión, por ejemplo, ir al oriente o ir al poniente, la red es unidimensional. Si hay dos ejes,

de modo que un paquete puede ir al oriente o al poniente, o bien ir al norte o al sur, la red es bidimensional, etcétera.

En la figura 8-4 se muestran varias topologías. Sólo se han incluido los enlaces (líneas) y los conmutadores (puntos). Las memorias y las CPU (que no se muestran) por lo regular estarían conectadas a los conmutadores mediante interfaces. En la figura 8-4(a) tenemos una configuración de **estrella** cero-dimensional, en la que las CPU y memorias estarían conectadas a los nodos exteriores, y el nodo central sólo realizaría conmutación. Aunque su diseño es sencillo, si el sistema es grande es muy probable que el conmutador central sea un importante cuello de botella. Además, desde el punto de vista de la tolerancia de fallos, este diseño es deficiente porque un solo fallo en el conmutador central destruye por completo el sistema.

En la figura 8-4(b) tenemos otro diseño cero-dimensional que está en el otro extremo del espectro, una **interconexión total**. Aquí cada nodo tiene una conexión directa con todos los demás nodos. Este diseño maximiza el ancho de banda bisectriz, minimiza el diámetro y es en extremo tolerante de fallos (puede perder cualquier enlace y aun así seguir plenamente conectado). Lo malo es que el número de enlaces requeridos para k nodos es $k(k-1)/2$, cifra que pronto se vuelve inmanejable cuando k es grande.

Una tercera topología cero-dimensional es el **árbol**, que se ilustra en la figura 8-4(c). Un problema con este diseño es que el ancho de banda bisectriz es igual a la capacidad de un enlace. Puesto que normalmente hay mucho tráfico cerca de la parte alta del árbol, los nodos de arriba se convertirán en cuellos de botella. Una forma de remediar este problema es incrementar el ancho de banda bisectriz dando a los enlaces superiores mayor ancho de banda. Por ejemplo, los enlaces del nivel más bajo podrían tener una capacidad de b, el siguiente nivel podría tener una capacidad de $2b$, y los enlaces del nivel superior podrían tener $4b$ cada uno. Un diseño así se denomina **árbol grueso** y se ha usado en multicomputadoras comerciales como la (ahora difunta) CM-5 de Thinking Machines.

El **anillo** de la figura 8-4(d) es una topología unidimensional según nuestra definición porque cada paquete que se envía tiene la opción de ir a la derecha o ir a la izquierda. La **cuadrícula** o **malla** de la figura 8-4(e) es un diseño bidimensional que se ha usado en muchos sistemas comerciales. Es muy regular, fácil de cambiar de escala hasta tamaños muy grandes, y tiene un diámetro que aumenta según la raíz cuadrada del número de nodos. Una variante de la cuadrícula es el **doble toroide** de la figura 8-4(f), que es una cuadrícula cuyas aristas están conectadas. Esta topología no sólo es más tolerante de fallos que la cuadrícula, sino que su diámetro también es menor porque las esquinas opuestas ahora pueden comunicarse con sólo dos pasos.

El **cubo** de la figura 8-4(g) es una topología tridimensional regular. Hemos ilustrado un cubo de $2 \times 2 \times 2$, pero en el caso general podría ser un cubo de $k \times k \times k$. En la figura 8-4(h) tenemos un cubo tetradimensional construido a partir de dos cubos tridimensionales cuyas aristas correspondientes están conectadas. Podríamos crear un cubo pentadimensional clonando la estructura de la figura 8-4(h) y conectando los nodos correspondientes para formar un bloque de cuatro cubos. Para llegar a seis dimensiones podríamos repetir el bloque de cuatro cubos e interconectar los nodos correspondientes, y así. Un cubo n-dimensional formado de esta manera se denomina **hipercubo**. Muchas computadoras paralelas usan esta topología porque el diámetro crece linealmente con la dimensionalidad. En otras palabras, el diámetro

es el logaritmo base 2 del número de nodos, de modo que, por ejemplo, un hipercubo 10-dimensional tiene 1024 nodos pero un diámetro de sólo 10, lo que da excelentes propiedades de retraso. Observe que, en contraste, 1024 nodos dispuestos en una cuadrícula de 32×32 tiene un diámetro de 62, más de seis veces más grande que el del hipercubo. El precio que se paga por el diámetro más pequeño es que el abanico de salida y por tanto el número de enlaces (y el costo) es mucho mayor para el hipercubo. No obstante, el hipercubo es una opción común en sistemas de alto desempeño.

Conmutación

Una red de interconexión consiste en conmutadores y alambres que los conectan. En la figura 8-5 vemos una red pequeña con cuatro conmutadores. Cada conmutador de este ejemplo tiene cuatro puertos de entrada y cuatro puertos de salida. Además, cada conmutador tiene algunas CPU y circuitos de interconexión (que no se muestran en su totalidad). La tarea del conmutador es aceptar paquetes que llegan por cualquier puerto de entrada y enviarlos por el puerto de salida correcto.

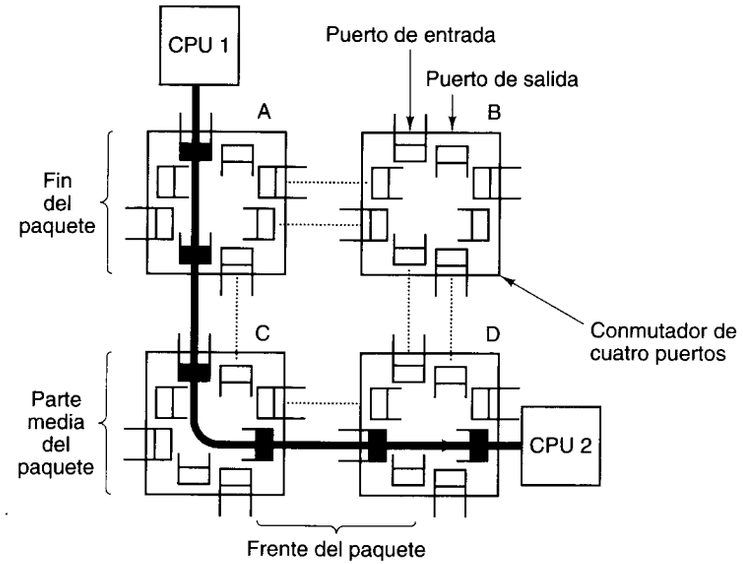


Figura 8-5. Red de interconexión en forma de cuadrícula de cuatro conmutadores. Sólo se muestran dos de las CPU.

Cada puerto de salida está conectado a un puerto de entrada que pertenece a otro conmutador por medio de un enlace serial o paralelo, que se muestra como líneas punteadas en la figura 8-5. Los enlaces seriales transfieren un bit a la vez. Los enlaces paralelos transfieren varios bits a la vez y por tanto tienen también señales para controlar el enlace. Los enlaces paralelos son más rápidos que los enlaces seriales con la misma frecuencia de reloj, pero

tienen el problema del **sesgo** (asegurarse de que todos los bits lleguen al mismo tiempo) y son mucho más costosos.

Hay varias posibles estrategias de conmutación. En una de ellas, llamada **conmutación de circuitos**, antes de enviar un paquete se reserva por adelantado el camino completo desde el origen hasta el destino. Se aseguran todos los puertos y buffers, de modo que cuando la transmisión se inicie esté garantizado que todos los recursos están disponibles y los bits pueden viajar a toda velocidad desde el origen, a través de todos los conmutadores, hasta el destino. La figura 8-5 ilustra la conmutación de circuitos, con un circuito reservado de la CPU 1 a la CPU 2, que se indica con la flecha negra curva. Aquí se han reservado tres puertos de entrada y tres puertos de salida.

La conmutación de circuitos puede compararse con reservar la ruta de un desfile a través de una ciudad, pidiendo a la policía que bloquee todas las calles laterales con barricadas. Se requiere planificación por adelantado, pero una vez que se ha efectuado, el desfile puede avanzar a toda velocidad sin interferencia por parte de otro tráfico. La desventaja es que se requiere planificación anticipada y se prohíbe cualquier otro tráfico, aunque de momento no haya ningún integrante del desfile (paquete) a la vista.

Una segunda estrategia de conmutación es la **conmutación de paquetes con almacenamiento y reenvío**. Aquí no es necesario reservar nada con anticipación. Más bien, el origen envía un paquete completo al primer conmutador, donde se almacena en su totalidad. En la figura 8-6(a), la CPU 1 es el origen, y el paquete entero, destinado a la CPU 2, se coloca primero en un buffer dentro del conmutador A. Una vez que el paquete se ha acumulado cabalmente en el conmutador A, se transfiere al conmutador C, como se muestra en la figura 8-6(b). Ya que el paquete entero ha llegado a C, se transfiere al conmutador D, como se muestra en la figura 8-6(c). Por último, el paquete se envía al destino, la CPU 2. Observe que no se requiere preparación y no se reservan recursos por adelantado.

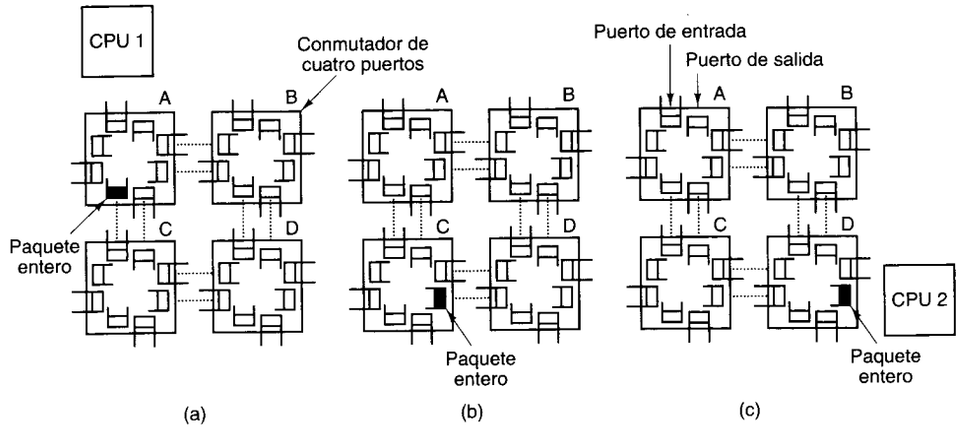


Figura 8-6. Conmutación de paquetes de almacenamiento y reenvío.

Los conmutadores de almacenamiento y reenvío deben colocar los paquetes en buffers porque cuando un origen (una CPU, memoria o conmutador) presenta un paquete, cabe la

posibilidad de que el puerto de salida requerido esté ocupado transmitiendo otro paquete. Si no se usaran buffers, los paquetes que llegaran y necesitaran un puerto ocupado tendrían que desecharse, y la red de interconexión sería muy poco confiable. Se usan tres estrategias de buffers. En la primera, **uso de buffers en la entrada**, cada puerto de entrada tiene asociados uno o más buffers en forma de cola de "primero que entra, primero que sale". Si el paquete que está a la cabeza de la cola no puede transmitirse porque el puerto de salida que necesita está ocupado, simplemente espera.

El problema con este diseño es que, si un paquete está esperando que se desocupe un puerto de salida, el paquete que viene atrás tiene que esperar, aunque esté destinado para un puerto desocupado. Esta situación se llama **bloqueo de cabeza de línea**, y se puede comparar a una sucesión de automóviles en una carretera de dos carriles que tiene que esperar porque el primer automóvil quiere dar vuelta a la izquierda pero no puede hacerlo a causa del tráfico en el otro sentido. Aunque el segundo automóvil y los que le siguen quieran seguir de frente, el automóvil que está a la cabeza los está bloqueando.

El bloqueo de cabeza de línea puede eliminarse con **uso de buffers en la salida**. En este diseño, los buffers están asociados a los puertos de salida, no a los de entrada. A medida que llegan los bits de un paquete, se almacenan en un buffer asociado al puerto de salida correcto. Así, los paquetes destinados al puerto m no pueden bloquear a los paquetes destinados al puerto n.

El número de buffers asociado a un puerto tanto de entrada como de salida es limitado. Si es preciso almacenar más paquetes de los que caben en los buffers, habrá que desechar paquetes. Una forma de mejorar esta situación es el uso de **buffers comunes**, en el que una sola reserva de buffers se reparte dinámicamente entre los puertos conforme se necesitan. Sin embargo, este esquema requiere una administración más compleja para llevar la contabilidad de los buffers, y también permite a una conexión con mucho tráfico acaparar todos los buffers y así bloquear otras conexiones. Además, cada conmutador necesita poder contener el paquete más grande, y probablemente varios paquetes de tamaño máximo, lo que tiende a incrementar las necesidades de memoria y reducir el tamaño máximo de paquete.

Aunque la conmutación de paquetes con almacenamiento y reenvío es flexible y eficiente, tiene el problema de aumentar la latencia (retraso) de la red de interconexión. Supongamos que el tiempo requerido para que un paquete avance un tramo en la figura 8-6 es de T ns. Puesto que el paquete debe copiarse cuatro veces para que llegue de la CPU 1 a la CPU 2 (en A, en C, en D y en la CPU de destino), y ningún copiado puede iniciarse antes de que termine el anterior, la latencia de la red de interconexión es de $4T$. Una solución es diseñar una red híbrida, con algunas de las propiedades de la conmutación de circuitos y algunas de la conmutación de paquetes. Por ejemplo, cada paquete podría dividirse lógicamente en unidades más pequeñas. Tan pronto como la primera unidad llega a un conmutador, se le puede transferir al siguiente conmutador, aun antes de que haya llegado la cola del paquete.

Semejante estrategia difiere de la conmutación de circuitos en cuanto a que no se reservan con anticipación recursos de extremo a extremo, y puede haber competencia por los recursos (puertos y buffers). En el **enrutamiento virtual de corte a través**, si la primera unidad de un paquete no puede avanzar, el resto del paquete sigue llegando. En el peor de los casos, este esquema se degrada hasta convertirse en conmutación de paquetes con almacenamiento y reenvío. En una estrategia alternativa, el **enrutamiento por túnel**, si la primera

unidad no puede avanzar, se le pide al origen que deje de transmitir, y así el paquete podría quedar tendido a lo largo de dos o tal vez aún más conmutadores. Una vez que están disponibles los recursos necesarios, el paquete puede continuar.

Vale la pena señalar que ambas estrategias de enrutamiento utilizan algo análogo a las instrucciones de filas de procesamientos de una CPU. En un instante dado, cada conmutador sólo está efectuando una fracción pequeña del trabajo, pero juntos logran un mejor desempeño que el que cada uno podría lograr individualmente.

Algoritmos de enrutamiento

En cualquier red que no sea cero-dimensional hay que tomar decisiones acerca de la ruta que los paquetes han de seguir del origen al destino. En muchos casos existen varias rutas. La regla que determina por cuál sucesión de nodos debe viajar un paquete para ir del origen al destino se denomina **algoritmo de enrutamiento**.

Se requieren buenos algoritmos de enrutamiento porque es común que haya varios caminos disponibles. Un buen algoritmo de enrutamiento puede distribuir la carga entre varios enlaces a fin de aprovechar al máximo el ancho de banda disponible. Además, el algoritmo de enrutamiento debe evitar los bloqueos mutuos dentro de la red de interconexión. Ocurre un **bloqueo mutuo** (o bloqueo mortal) cuando varios paquetes que están en tránsito al mismo tiempo se han apoderado de recursos de tal manera que ninguno de ellos puede avanzar y todos quedan bloqueados indefinidamente.

En la figura 8-7 se da un ejemplo de bloqueo mutuo en una red de interconexión de conmutación de circuitos. (También puede haber bloqueos mutuos en las redes de conmutación de paquetes, pero es más fácil ilustrarlo en las de conmutación de circuitos.) Aquí cada CPU está tratando de enviar un paquete a la CPU que está en la posición diagonalmente opuesta. Cada una ha logrado reservar los puertos de entrada y de salida de su conmutador local, así como un puerto de entrada en el conmutador que sigue, pero no puede obtener el puerto de salida del segundo conmutador, por lo que tiene que esperar hasta que ese puerto se desocupe. Lo malo es que si las cuatro CPU inician este proceso simultáneamente, todas ellas se bloquearán y la red se suspenderá indefinidamente.

Los algoritmos de enrutamiento se pueden clasificar como de enrutamiento de origen o de enrutamiento distribuido. En el **enrutamiento de origen**, el origen determina con anticipación la ruta completa que se seguirá por la red de interconexión, expresada como una lista de números de puerto que se usarán en cada conmutador del camino. Por lo regular, si la ruta pasa por k conmutadores, los primeros k bytes de cada paquete contendrán los k números de puerto de salida requeridos, un byte por puerto. Cuando el paquete llega a un puerto, el primer byte se separa y se usa para determinar el puerto de salida que se usará. El resto del paquete se encamina entonces al puerto correcto. En cada paso del camino, el paquete se acorta en un byte, y expone un nuevo número de puerto que será el siguiente en seleccionarse.

En el enrutamiento distribuido, cada conmutador toma una decisión respecto al puerto al que enviará cada paquete que llegue. Si toma la misma decisión para cada paquete dirigido a un destino dado, se dice que el enrutamiento es **estático**. Si el conmutador toma en cuenta el gráfico actual, se dice que el enrutamiento es **adaptativo**.

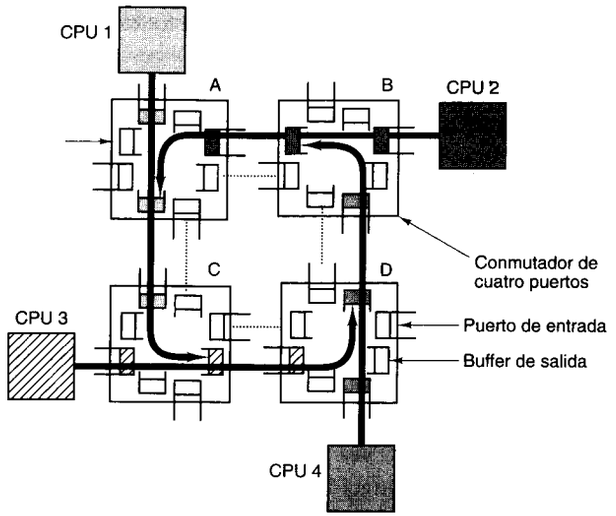


Figura 8-7. Bloqueo mutuo en una red de interconexión con conmutación de circuitos.

Un algoritmo de enrutamiento muy utilizado en cuadrículas rectangulares con cualquier cantidad de dimensiones y que se sabe no causa bloqueos mutuos es el **enrutamiento dimensional**. En este algoritmo, el paquete primero se desplaza a lo largo del eje x hasta la coordenada correcta, y luego a lo largo del eje y hasta la coordenada correcta, y así sucesivamente si hay más dimensiones. Por ejemplo, para ir de (3, 7, 5) a (6, 9, 8) el paquete primero iría de $x = 3$ a $x = 6$ pasando por (4, 7, 5), (5, 7 5) y (6, 7, 5). Luego viajaría por el eje y viajando a (6, 8, 5) y (6, 9, 5). Por último, el paquete viajaría por el eje z a (6, 9, 6), (6, 9, 7) y (6, 9, 8). Este algoritmo evita los ciclos que causan un bloqueo mutuo.

8.1.3 Desempeño

Lo que se busca al construir una computadora paralela es que opere más rápidamente que una máquina uniprocesador. Si no logra este sencillo objetivo, no tiene razón de existir. Además, la meta debe alcanzarse con eficacia de costos. Una máquina que es dos veces más rápida que un uniprocesador pero cuesta 50 veces más probablemente no tendrá muchas ventas. En esta sección examinaremos algunos de los aspectos de desempeño asociados a las arquitecturas de computadoras paralelas.

Métricas de hardware

Desde el punto de vista del hardware, las métricas de desempeño que interesan son la velocidad de la CPU y de E/S, y el desempeño de la red de interconexión. Las velocidades de la CPU y de E/S son las mismas que en el uniprocesador, de modo que los parámetros clave de

interés en un sistema paralelo son los asociados a la interconexión. Dos son los elementos clave: la latencia y el ancho de banda, que examinaremos a continuación.

La latencia de viaje redondo es el tiempo que una CPU tarda en enviar un paquete y recibir una respuesta. Si el paquete se envía a una memoria, la latencia mide el tiempo que toma leer o escribir una palabra o un bloque de palabras. Si el paquete se envía a otra CPU, la latencia mide el tiempo de comunicación interprocesador para paquetes de ese tamaño. Por lo regular, la latencia que interesa es la correspondiente a paquetes mínimos, que a menudo son una palabra o una línea de caché pequeña.

La latencia comprende varios factores, y es diferente para las interconexiones de conmutación de circuitos, de almacenamiento y reenvío, de corte a través virtual y de enrutamiento por túnel. En el caso de la conmutación de circuitos, la latencia es la suma del tiempo de preparación y el tiempo de transmisión. Para preparar un circuito, es preciso enviar un paquete "de sondeo" que reserve los recursos e informe de los resultados de su gestión. Después, hay que armar el paquete de datos. Una vez que está listo, sus bits pueden fluir a toda velocidad, de modo que si el tiempo de preparación total es T_s, el tamaño del paquete es de p bits y el ancho de banda es de b bits/s, la latencia en un sentido será $T_s + p/b$. Si el circuito es dúplex, no habrá tiempo de preparación para la respuesta, de modo que la latencia mínima para enviar un paquete de p bits y obtener una respuesta de p bits es de $T_s + 2p/b$ segundos.

En el caso de la conmutación de paquetes no es necesario enviar un paquete de sondeo al destino con anticipación, pero de todos modos hay cierto tiempo de preparación interno para armar el paquete, T_a. Aquí el tiempo de transmisión en un sentido es $T_a + p/b$, pero éste es sólo el tiempo requerido para que el paquete llegue al primer conmutador. Hay un retardo finito dentro del conmutador, digamos T_d y luego el proceso se repite con el siguiente conmutador, etc. El retraso T_d se compone de un tiempo de procesamiento y un retraso de la cola de espera, mientras se desocupa el puerto de salida. Si hay n conmutadores, la latencia total en un sentido será $T_a + n(p/b + T_d) + p/b$, donde el término final se debe al copiado del último conmutador al destino.

Las latencias en un sentido para el corte virtual a través y el enrutamiento por túnel en el mejor de los casos son cercanas a $T_a + p/b$ porque no se requiere un paquete de sondeo para establecer un circuito, y tampoco hay retraso de almacenamiento y reenvío. Básicamente, la latencia consiste en el tiempo de preparación inicial mientras se arma el paquete, más el tiempo que toma "sacar los bits a la calle". En todos los casos hay que sumar el retraso de propagación, pero éste suele ser pequeño.

La otra métrica de hardware es el ancho de banda. Muchos programas paralelos, sobre todo en las ciencias naturales, transfieren de aquí para allá una gran cantidad de datos, por lo que el número de bytes/s que el sistema puede transferir es crítico para el desempeño. Existen varias métricas para el ancho de banda. Ya vimos una de ellas, el ancho de banda bisectriz. Otra es el **ancho de banda agregado**, que se calcula por la simple suma de las capacidades de todos los enlaces. Este número proporciona la cantidad máxima de bits que pueden estar en tránsito en un momento dado. Otra métrica importante es el ancho de banda medio en la salida de cada CPU. Si cada CPU puede transmitir 1 MB/s, no sirve de mucho que la interconexión tenga un ancho de banda bisectriz de 100 GB/s. La comunicación estará limitada a la cantidad de datos que cada CPU puede enviar.

En la práctica es muy difícil acercarse siquiera al ancho de banda teórico. Muchas fuentes de gasto extra reducen la capacidad. Por ejemplo, siempre hay cierto gasto extra asociado a cada paquete: para armarlo, construir su cabecera y ponerlo en marcha. El envío de 1024 paquetes de 4 bytes nunca logrará el mismo ancho de banda que el envío de un paquete de 4096 bytes. Lamentablemente, el uso de paquetes pequeños es mejor para lograr latencias bajas, puesto que los paquetes grandes bloquean demasiado tiempo las líneas y los conmutadores. Así, hay un conflicto inherente entre lograr una latencia baja y aprovechar al máximo el ancho de banda. En algunas aplicaciones, una cosa es más importante que la otra, y en otras aplicaciones es al revés. No obstante, cabe señalar que siempre es posible comprar más ancho de banda (instalando más cables o cables más anchos), pero no es posible comprar latencias más bajas. Por ello, generalmente es mejor tratar de hacer que las latencias sean lo más bajas posibles, y preocuparse por el ancho de banda después.

Métricas de software

Las métricas de hardware como la latencia y el ancho de banda se ocupan de lo que el hardware hace. Sin embargo, los usuarios tienen un punto de vista distinto. Ellos quieren saber qué tanto va a aumentar la rapidez de ejecución de sus programas en una computadora paralela, en comparación con un uniprocesador. Para ellos, la métrica clave es la aceleración: cuántas veces más rápidamente se ejecuta un programa en un sistema de n procesadores que en uno de un solo procesador. Por lo regular, estos resultados se presentan en gráficas como la de la figura 8-8. En ella vemos la ejecución de varios programas paralelos distintos en una multicomputadora formada por 64 CPU Pentium Pro. Cada curva muestra la aceleración de un programa con k CPU en función de k. La línea punteada indica la aceleración perfecta, en la que el uso de k CPU hace que el programa sea k veces más rápido, para cualquier k. Pocos programas logran una aceleración perfecta, pero algunos se acercan. El problema de N cuerpos se adapta muy bien a una ejecución paralela; awari (un juego de mesa africano) obtiene resultados razonables; pero la inversión de cierta matriz de perfilado no tarda menos de cinco veces menos por más CPU que estén disponibles. Los programas y los resultados se analizan en (Bal *et al.*, 1998).

Una parte de la razón por la que es casi imposible lograr una aceleración perfecta es que casi todos los programas tienen algún componente secuencial, como la fase de inicialización, la lectura de datos o la reunión de los resultados. Tener muchas CPU no ayuda mucho a estas actividades. Suponga que un programa se ejecuta durante T segundos en un uniprocesador, y que una fracción f de este tiempo corresponde a código secuencial y una fracción $(1 - f)$ es potencialmente paralelizable, como se muestra en la figura 8-9(a). Si este último código se puede ejecutar en n CPU sin gasto extra, su tiempo de ejecución podrá reducirse de $(1 - f)T$ a $(1 - f)T/n$ en el mejor de los casos, como se muestra en la figura 8-9(b). Esto da un tiempo de ejecución total para las partes secuencial y paralela de $fT + (1 - f)T/n$. La aceleración no es más que el tiempo de ejecución del programa original, T, dividido entre este nuevo tiempo de ejecución:

$$Aceleración = \frac{n}{1 + (n - 1)f}$$

Con $f = 0$ podemos obtener una aceleración lineal, pero con $f > 0$ no es posible lograr la aceleración perfecta a causa del componente secuencial. Este resultado se conoce como **ley de Amdahl**.

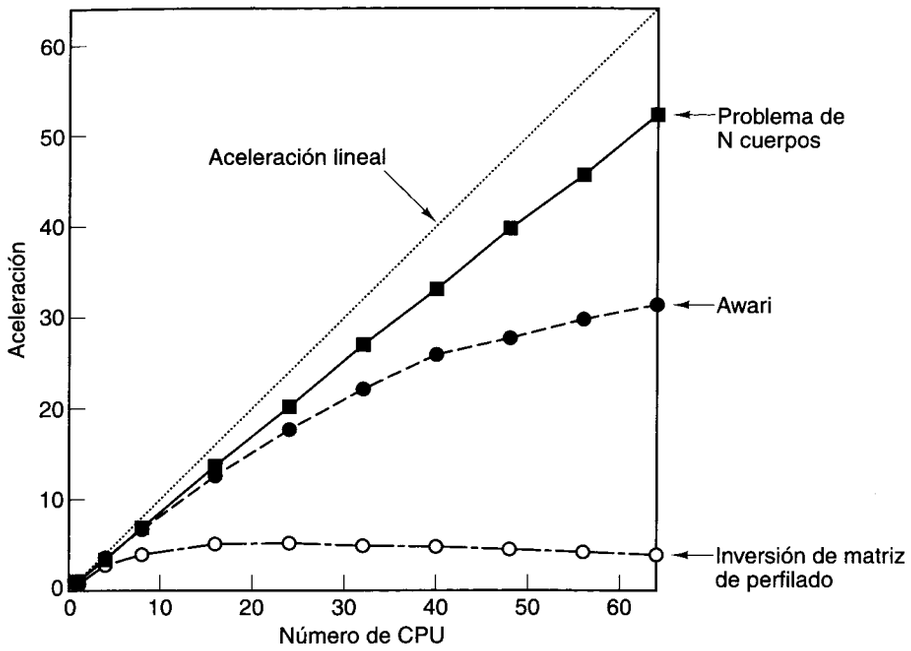


Figura 8-8. Los programas reales logran una aceleración menor que la perfecta indicada por la línea punteada.

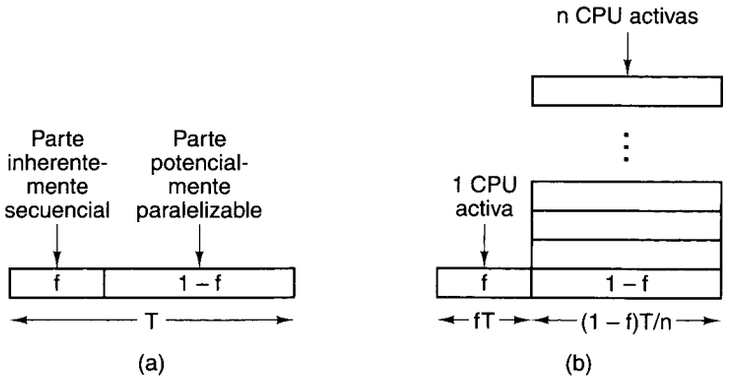


Figura 8-9. (a) Programa que tiene una parte secuencial y una parte paralelizable. (b) Efecto de ejecutar una parte del programa en paralelo.

La ley de Amdahl no es la única razón por la que es casi imposible lograr una aceleración perfecta. Las latencias de comunicación distintas de cero, los anchos de banda de comunicación finitos y las ineficiencias algorítmicas también son factores. Además, aunque se contara

con 1000 CPU, no todos los programas pueden escribirse de modo que aprovechen tantas CPU, y el gasto extra de ponerlas todas en marcha puede ser considerable. Por añadidura, es común que el algoritmo mejor conocido no se pueda paralelizar bien, y sea necesario usar un algoritmo subóptimo en el caso paralelo. A pesar de todo esto, para muchas aplicaciones es altamente deseable lograr que el programa funcione con una rapidez n veces mayor, aunque se necesiten $2n$ CPU para ello. Después de todo, las CPU no son tan caras, y muchas compañías se conforman con una eficiencia muy por debajo del 100% en otras facetas de su operación.

Cómo lograr un buen desempeño

La forma más sencilla de mejorar el desempeño es agregar más CPU al sistema. Sin embargo, esta adición debe efectuarse de tal manera que se evite la creación de cuellos de botella. Un sistema en el que es posible añadir más CPU e incrementar de manera acorde la potencia de cómputo es un sistema **escalable**.

Para ver algunas de las implicaciones de la escalabilidad, considere cuatro CPU conectadas por un bus, como se ilustra en la figura 8-10(a). Ahora imagine aumentar la escala del sistema a 16 CPU añadiendo 12 más, como se muestra en la figura 8-10(b). Si el ancho de banda del bus es de b MB/s, entonces si cuadruplicamos el número de CPU también habremos reducido el ancho de banda disponible por CPU de $b/4$ MB/s a $b/16$ MB/s. Un sistema así no es escalable.

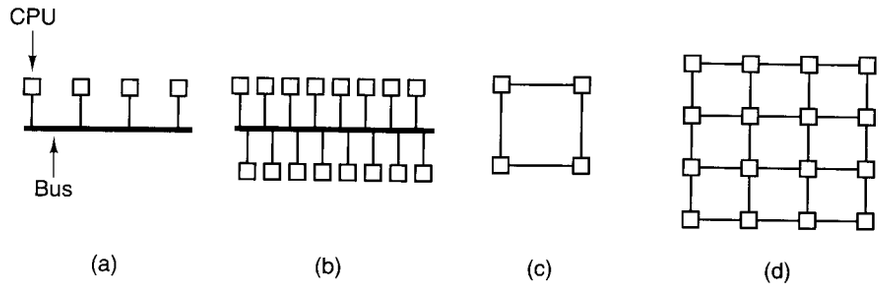


Figura 8-10. (a) Sistema con cuatro CPU basado en bus. (b) Sistema de 16 CPU basado en bus. (c) Sistema de cuatro CPU basado en cuadrícula. (d) Sistema de 16 CPU basado en cuadrícula.

Hagamos ahora lo mismo con un sistema basado en una cuadrícula, como el que se muestra en la figura 8-10(c) y 8-10(d). Con esta topología, la adición de nuevas CPU también implica la adición de nuevos enlaces, por lo que el aumento de escala del sistema no hace que el ancho de banda por CPU se reduzca, como en el caso de un bus. De hecho, la proporción enlaces/CPU aumenta de 1.0 con 4 CPU (4 CPU, 4 enlaces) a 1.5 con 16 CPU (16 CPU, 24 enlaces), así que la adición de CPU mejora el ancho de banda colectivo por CPU.

Desde luego, el ancho de banda no es la única consideración. La adición de CPU al bus no aumenta el diámetro de la red de interconexión ni la latencia en ausencia de tráfico, pero la adición de CPU a la cuadrícula sí lo hace. En el caso de una cuadrícula de $n \times n$, el diámetro es de $2(n-1)$, de modo que la latencia de peor caso (y la latencia media) aumenta aproxima-

damente con la raíz cuadrada del número de CPU. Si hay 400 CPU, el diámetro es de 38, mientras que si hay 1600 CPU el diámetro es de 78. Cuadruplicar el número de CPU duplica aproximadamente el diámetro y por tanto la latencia media.

Idealmente, un sistema escalable debe mantener el mismo ancho de banda medio por CPU y una latencia media constante a medida que se añaden más CPU. En la práctica, es posible mantener suficiente ancho de banda por CPU, pero en todos los diseños prácticos la latencia crece al aumentar el tamaño. Lo mejor que puede hacerse es que el crecimiento sea logarítmico, como en un hipercubo.

El aumento de la latencia al aumentar la escala del sistema es un problema porque la latencia a menudo es fatal para el desempeño en las aplicaciones de grano mediano y fino. Si un programa necesita datos que no están en la memoria local, a menudo su obtención implica un retraso sustancial, y cuanto mayor es el sistema, más largo es el retraso, como acabamos de ver. Este problema se presenta tanto en multiprocesadores como en multicomputadoras, porque en ambos casos la memoria física se divide en módulos dispersos.

Una consecuencia de esta observación es que los diseñadores de sistemas a menudo hacen hasta lo imposible por reducir, o al menos ocultar, la latencia, empleando varias técnicas que mencionaremos a continuación. La primera técnica para ocultar la latencia es la repetición de datos. Si es posible mantener copias de un bloque de datos en varios lugares, será posible acelerar los accesos desde esos lugares. Una de las técnicas de repetición es el uso de cachés, en el que una o más copias de algunos bloques de datos se mantienen cerca de donde se están usando, y también cerca de donde "deben estar". Otra estrategia consiste en mantener varias copias pares —copias que tienen la misma categoría— en contraposición a la relación asimétrica primario/secundario que se usa en las cachés. Cuando se mantienen varias copias, en cualquier forma, las cuestiones fundamentales son dónde se colocan los bloques de datos, cuándo y quién los coloca ahí. Las respuestas varían desde la colocación dinámica por demanda efectuada por hardware hasta la colocación intencional en el momento de la carga siguiendo directrices del compilador. En todos los casos, la coherencia de gestión es importante.

Una segunda técnica para ocultar la latencia es la **prebúsqueda**. Si se puede traer un dato antes de que se necesite, el proceso de búsqueda podrá traslaparse con la ejecución normal para que cuando el dato se necesite, ya esté ahí. La prebúsqueda puede ser automática o estar bajo el control del programa. Cuando una caché carga no sólo la palabra a la que se hizo referencia, sino toda una línea de caché que contiene la palabra, está apostando a que las palabras que siguen también vayan a necesitarse pronto.

La prebúsqueda también puede controlarse explícitamente. Cuando un compilador se da cuenta de que va a necesitar ciertos datos, puede incluir una instrucción explícita para obtenerlos, y colocar esa instrucción en una posición suficientemente adelantada como para que los datos lleguen a tiempo. Esta estrategia requiere que el compilador tenga un conocimiento total de la máquina subyacente y de su temporización, así como control sobre dónde se colocan todos los datos. Tales instrucciones LOAD especulativas funcionan óptimamente cuando se sabe con seguridad que los datos se van a necesitar. Causar un fallo de página por un LOAD que está en una rama que después no se toma resulta muy costoso.

Una tercera técnica que puede ocultar la latencia es el **multienlace** (*multithreading*). Casi todos los sistemas modernos manejan el concepto de multiprogramación, en el que varios

procesos pueden ejecutarse simultáneamente (o en seudoparalelo por tiempo compartido). Si puede lograrse que la conmutación entre procesos sea lo bastante rápida (por ejemplo, si se proporciona a cada proceso su propio mapa de memoria y registros en hardware), entonces cuando un proceso se bloquea mientras espera la llegada de datos remotos el hardware puede conmutar rápidamente a otro proceso que sí pueda continuar. En el caso limitante, la CPU ejecuta la primera instrucción del enlace 1, la segunda instrucción del enlace 2, etc. De este modo, la CPU puede mantenerse ocupada, aunque los enlaces individuales tengan latencias de memoria largas.

De hecho, algunas máquinas conmutan automáticamente entre los procesos de forma cíclica, después de cada instrucción, a fin de ocultar las latencias largas. Una de las primeras supercomputadoras, la CDC 6600, llevó esta idea a tal extremo que en sus anuncios se aseguraba que tenía 10 unidades de procesamiento periféricas que podían operar en paralelo. En realidad sólo había un procesador periférico que simulaba 10 procesadores ejecutando instrucciones de cada uno por turno circular, primero una instrucción del procesador periférico 1, luego una instrucción del procesador periférico 2, y así.

Una cuarta técnica para ocultar la latencia es el uso de escrituras que no se bloquean. Normalmente, cuando se ejecuta una instrucción STORE, la CPU espera hasta que se finaliza la instrucción antes de continuar. Con escrituras que no se bloquean, se inicia la operación de memoria, pero el programa de todos modos continúa. Continuar después de un LOAD es más difícil, pero con ejecución en desorden hasta eso es posible.

8.1.4 Software

Aunque este capítulo trata primordialmente las arquitecturas de computadora paralelas, es apropiado hablar un poco acerca del software también. Después de todo, sin software paralelo el hardware paralelo no sirve de mucho, y es por ello que los buenos diseñadores de hardware toman en cuenta las necesidades del software al diseñar el hardware. Si desea un tratamiento del software para computadoras paralelas, vea (Wilkinson y Allen, 1999).

Existen cuatro estrategias generales para producir software para computadoras paralelas. En un extremo está la adición de bibliotecas numéricas especiales a lenguajes secuenciales por lo demás normales. Por ejemplo, se podría invocar desde un programa secuencial un procedimiento de biblioteca que invierta una matriz grande o resuelva un conjunto de ecuaciones diferenciales parciales en un procesador paralelo sin que el procesador siquiera se dé cuenta de la existencia de paralelismo. El problema con este enfoque es que el paralelismo sólo puede usarse en unos cuantos procedimientos y el grueso del código seguirá siendo secuencial.

Una segunda estrategia es la adición de bibliotecas especiales que contienen primitivas de comunicación y control. Aquí el programador tiene la responsabilidad de crear y controlar el paralelismo dentro de un lenguaje de programación convencional, empleando estas primitivas adicionales.

El siguiente paso es añadir unas cuantas construcciones especiales a lenguajes de programación existentes, como la capacidad de bifurcar fácilmente nuevos procesos paralelos, ejecutar las iteraciones de un ciclo en paralelo, o efectuar operaciones aritméticas con

todos los elementos de un vector al mismo tiempo. Este enfoque se ha adoptado ampliamente y se han modificado muchos lenguajes de programación para que incluyan algo de paralelismo.

La estrategia más extrema sería inventar un lenguaje totalmente nuevo especialmente para el procesamiento en paralelo. La ventaja obvia de inventar un nuevo lenguaje es que éste sería idóneo para la programación en paralelo, pero la desventaja igualmente obvia es que los programadores tendrían que aprender un nuevo lenguaje. Muchos de ellos tienen otras cosas que hacer con su tiempo. Casi todos los lenguajes paralelos nuevos son imperativos (con instrucciones para modificar variables de estado), pero unos cuantos son funcionales, basados en lógica u orientados a objetos.

Ha habido tantas bibliotecas, extensiones y lenguajes inventados para la programación en paralelo, y abarcan una gama tan amplia de posibilidades, que es imposible clasificarlos todos de alguna forma razonable. En lugar de intentarlo, nos concentraremos en cinco cuestiones básicas que constituyen el corazón de todo el software para computadoras paralelas.

1. Modelos de control.
2. Granularidad del paralelismo.
3. Paradigmas computacionales.
4. Métodos de comunicación.
5. Primitivas de sincronización.

Cada una de estas cuestiones se estudiará en su oportunidad.

Modelos de control

Tal vez la decisión fundamental que el software debe tomar es si habrá un solo enlace de control o varios. En el primer modelo hay un programa y un contador de programa, pero varios conjuntos de datos. Cuando se emite una instrucción, se ejecuta con todos los conjuntos de datos simultáneamente, con diferentes elementos de procesamiento.

Por ejemplo, imagine un programa meteorológico que cada hora recibe mediciones de temperatura de miles de sensores remotos que tienen que promediar para cada sensor. Cuando el programa emite la instrucción

CARGAR LA TEMPERATURA PARA LA 1 A.M. EN EL REGISTRO R1

cada procesador la ejecuta utilizando sus propios datos y su propio R1. Después, cuando el programa emite la instrucción

SUMAR LA TEMPERATURA PARA LAS 2 A.M. AL REGISTRO R1

una vez más, cada procesador lo hace utilizando sus propios datos. Al final del cálculo, cada procesador habrá calculado la temperatura media para un sensor distinto.

Las implicaciones de este modelo de programación para el hardware son enormes. De hecho, lo que dice es que cada elemento de procesamiento es básicamente una ALU y una memoria, sin lógica propia para decodificar instrucciones. En vez de ello, una sola unidad centralizada trae instrucciones y le dice a las ALU lo que tienen que hacer a continuación.

En el modelo alternativo hay varios enlaces de control, cada uno con su propio contador de programa, registros y variables locales. Cada enlace de control ejecuta su propio programa con sus propios datos, posiblemente comunicándose o sincronizándose con otros enlaces de control cada cierto tiempo. Existen muchas variaciones de esta idea básica, y juntas constituyen el modelo dominante del procesamiento en paralelo. Por esta razón, ahora nos concentraremos primordialmente en la computación en paralelo con múltiples enlaces de control.

Granularidad del paralelismo

Se puede introducir paralelismo de control en diversos niveles. En el nivel más bajo, las instrucciones de máquina individuales pueden contener paralelismo (por ejemplo, el IA-64). En este nivel, los programadores normalmente no tienen conocimiento del paralelismo; está bajo el control del compilador o del hardware.

Un nivel más arriba está el **paralelismo en el nivel de bloques**, que permite a los programadores controlar explícitamente cuáles enunciados deben ejecutarse secuencialmente y cuáles deben ejecutarse en paralelo. Una de las formas más elegantes de expresar este tipo de paralelismo se debe a Algol 68, en el cual se usaba

begin Enunciado-1; Enunciado-2; Enunciado-3 **end**

para crear un bloque con (por ejemplo) tres enunciados arbitrarios que se debían ejecutar en sucesión. En contraste,

begin Enunciado-1, Enunciado-2, Enunciado-3 **end**

servía para ejecutar los mismos tres enunciados en paralelo. Mediante una colocación cuidadosa de signos de punto y coma, comas, paréntesis y delimitadores **begin/end**, se podían expresar combinaciones arbitrarias de ejecución secuencial y paralela.

Se presenta un paralelismo con grano un poco más grueso cuando es posible llamar a un procedimiento y no obligar al invocador a esperar a que termine antes de continuar. No esperar implica que el invocador y el invocado se ejecutarán en paralelo. Si el invocador está en un ciclo que llama a un procedimiento en cada iteración y no espera a que termine ninguno de ellos, es posible iniciar un gran número de procedimientos en paralelo a la vez.

Otra forma de paralelismo es tener un método para que un proceso cree o bifurque múltiples **enlaces** o **procesos ligeros**, todos los cuales se ejecutan dentro del espacio de direcciones del proceso. Cada enlace tiene su propio contador de programa, registros y pila, pero fuera de eso comparte el resto del espacio de direcciones (y todas las variables globales) con todos los demás enlaces. (En contraste con los enlaces, diferentes procesos no comparten un mismo espacio de direcciones.) Los enlaces se ejecutan con independencia unos de otros, posiblemente en diferentes CPU. En algunos sistemas, el sistema operativo tiene conoci-

miento de todos los enlaces y se encarga de planificarlos; en otros, cada proceso de usuario se encarga de su propia planificación y gestión de enlaces, sin que el sistema operativo siquiera sepa que existen enlaces.

Por último, la forma más burda de paralelismo es hacer que varios procesos independientes colaboren para resolver un problema. A diferencia de los enlaces, que comparten un solo espacio de direcciones, cada proceso independiente tiene su espacio propio, por lo que los procesos deben cooperar a cierta distancia. Esto implica que el problema tiene que dividirse en fragmentos relativamente grandes, uno para cada proceso. Sin embargo, los procesos independientes ofrecen la mayor oportunidad para aprovechar el paralelismo a gran escala, sobre todo en multicomputadoras.

Paradigmas computacionales

Casi todos los programas paralelos, sobre todo los que implican grandes números de enlaces o procesos independientes, usan algún paradigma subyacente para estructurar su trabajo. Existen muchos de esos paradigmas. En esta sección sólo mencionaremos algunos de los más utilizados.

Una generalización de la idea de tener un programa paralelo que consta de un enlace de control y múltiples unidades de ejecución es el **paradigma de programa único, múltiples datos** (SPMD, *Single Program Multiple Data*). La idea aquí es que aunque el sistema consiste en varios procesos independientes, todos ejecutan el mismo programa pero con diferentes conjuntos de datos. Sólo que ahora, a diferencia del ejemplo de las temperaturas, no están en coordinación, sincronizados hasta la última instrucción. En vez de ello, cada uno realiza el mismo cálculo, pero a su propio ritmo.

Un segundo paradigma es la **fila de procesamiento** (*pipeline*), que se ilustra en la figura 8-11(a) con tres procesos. Los datos se alimentan al primer proceso, que los transforma y los alimenta al segundo proceso, etc. Si el flujo de datos es largo (por ejemplo, una presentación en video), todos los procesadores podrían estar ocupados al mismo tiempo. Las filas de procesamiento UNIX funcionan así y se pueden ejecutar como procesos individuales en paralelo en una multicomputadora o un multiprocesador.

Otro paradigma es el cálculo de la figura 8-11(b), en el que el trabajo se divide en fases, por ejemplo, iteraciones de un ciclo. Durante cada fase, varios procesos trabajan en paralelo, pero cuando cada uno termina espera hasta que los demás han terminado también antes de iniciar la siguiente fase. Un cuarto paradigma es el de **divide y vencerás**, que se ilustra en la figura 8-11(c). En este paradigma un proceso inicia y luego se bifurca en otros procesos a los que puede delegar parte del trabajo. Una analogía en el mundo de la construcción es un contratista general que recibe un pedido, y luego subcontrata gran parte del trabajo a subcontratistas de albañilería, electricidad, plomería, pintura y otras especialidades. Éstos, a su vez, podrían subcontratar parte de su trabajo a otros subcontratistas especializados.

Nuestro último modelo es el **paradigma del trabajador repetido**, a veces llamado **granja de tareas**, el cual se ilustra en la figura 8-11(d). Aquí se tiene una cola de trabajo centralizada y los trabajadores sacan tareas de la cola y las llevan a cabo. Si una tarea genera nuevas tareas, éstas se añaden a la cola central. Cada vez que un trabajador termina su tarea actual, acude a la cola de tareas para obtener una nueva. Este paradigma también puede implementarse con un administrador activo en la parte superior que reparte el trabajo, en lugar de obligar a los trabajadores que lo consigan por su cuenta.

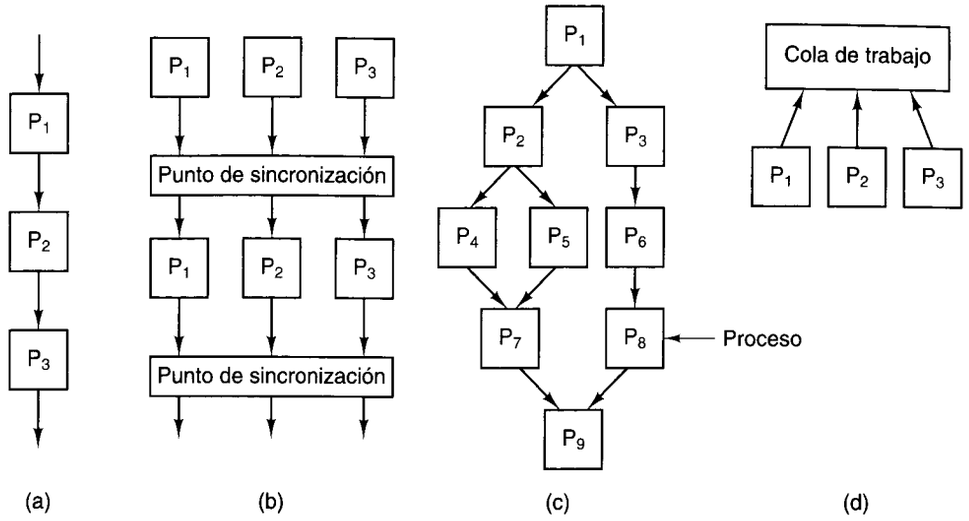


Figura 8-11. Paradigmas computacionales. (a) Fila de procesamiento. (b) Cómputo en fases. (c) Divide y vencerás. (d) Trabajador repetido.

Métodos de comunicación

Cuando un programa se divide en fragmentos (digamos, procesos) que se ejecutan en paralelo, los fragmentos (procesos) a menudo necesitan comunicarse entre sí. Esta comunicación puede efectuarse de una de dos maneras: variables compartidas o transferencia explícita de mensajes. En el primer método, todos los procesos tienen acceso a la memoria lógica común y pueden comunicarse leyéndola y escribiendo en ella. Por ejemplo, un proceso puede ajustar una variable y otro proceso puede leerla.

En un multiprocesador, las variables pueden compartirse entre múltiples procesos mapeando la misma página al espacio de direcciones de cada proceso. Entonces se pueden leer y escribir variables compartidas empleando instrucciones de máquina LOAD y STORE. En una multicomputadora, aunque no tiene memoria física compartida, también es posible compartir variables lógicamente, aunque es un poco más complicado. Como vimos antes, Linda maneja el concepto de espacio de tuplas compartido, aun en una multicomputadora, y Orca maneja el concepto de objetos compartidos cruzando fronteras de máquinas. Otra posibilidad es compartir un solo espacio de direcciones en una multicomputadora y paginar a través de la red de interconexión. En síntesis, es posible permitir que los procesos se comuniquen a través de variables compartidas tanto en multiprocesadores como en multicomputadoras.

La alternativa a la comunicación a través de memoria compartida es la comunicación por transferencia explícita de mensajes. En este modelo, los procesos usan primitivas como send y receive para comunicarse. Un proceso emite un send, nombrando a otro proceso como destino. Tan pronto como el segundo emite un receive, el mensaje se copia en el espacio de direcciones del receptor.

La transferencia de mensajes tiene muchas variaciones, pero todas se reducen a tener dos primitivas send y receive que sirven para transmitir mensajes. Éstas generalmente se implementan como llamadas al sistema. Veremos algunos de los detalles más adelante en este capítulo.

Otro aspecto importante de la transferencia de mensajes es el número de receptores. El caso más simple es el de un transmisor y un receptor, llamado **transferencia de mensajes punto a punto**. Sin embargo, a veces es útil entregar un mensaje a todos los procesos, en lo que se conoce como **difusión** (*broadcasting*), o a un subconjunto específico de los procesos, en lo que se conoce como **multidifusión** (*multicasting*).

Cabe señalar que la transferencia de mensajes es fácil de implementar en un multiprocesador con sólo copiar del transmisor al receptor. Por tanto, las cuestiones de memoria física compartida (multiprocesador *versus* multicomputadora) y memoria lógica compartida (comunicación a través de variables compartidas *versus* transferencia explícita de mensajes) son independientes. Las cuatro combinaciones tienen sentido y pueden implementarse; se enumeran en la figura 8-12.

Físico (hardware)	Lógico (software)	Ejemplos
Multiprocesador	Variables compartidas	Procesamiento de imágenes como en la figura 8-1
Multiprocesador	Transf. de mensajes	Transf. de mensajes simulada con buffers en la memoria
Multicomputadora	Variables compartidas	DSM, Linda, Orca, etc. en un SP/2 o una red de PC
Multicomputadora	Transf. de mensajes	PVM o MPI en un SP/2 o una red de PC

Figura 8-12. Combinaciones de compartimiento físico y lógico.

Primitivas de sincronización

Los procesos paralelos no sólo necesitan comunicarse; con frecuencia también necesitan sincronizar sus acciones. Un ejemplo de sincronización es cuando los procesos comparten variables lógicamente y tienen que asegurarse de que mientras un proceso está escribiendo en una estructura de datos compartida ningún otro proceso esté tratando de leerla. En otras palabras, se requiere alguna forma de **exclusión mutua** para evitar que varios procesos usen los mismos datos al mismo tiempo.

Existen diversas primitivas de software que pueden servir para asegurar la exclusión mutua. Éstas incluyen semáforos, candados, mutexes y secciones críticas. La característica común de todas estas primitivas es que permiten a un proceso solicitar el uso exclusivo de algún recurso (variable compartida, dispositivo de E/S, etc.). Si se otorga el permiso, el proceso puede usar el recurso. Si un segundo proceso pide permiso mientras el primero todavía está usando el recurso, se le negará el permiso o se bloqueará hasta que el primer proceso haya liberado el recurso.

Un segundo tipo de primitiva de sincronización que se necesita en muchos programas paralelos es alguna forma de permitir que todos los procesos se bloqueen hasta que finalice cierta fase del trabajo, como se muestra en la figura 8-11(b). Una primitiva común aquí es la **barrera**. Cuando un proceso llega a una barrera, se bloquea hasta que todos los procesos llegan a ella. Cuando llega el último proceso, todos los procesos se liberan simultáneamente y pueden continuar.

8.1.5 Taxonomía de computadoras paralelas

Aunque podríamos hablar mucho más acerca del software para computadoras paralelas, las limitaciones de espacio nos obligan a regresar al tema principal de este capítulo, la arquitectura de las computadoras paralelas. Se han propuesto y construido muchos tipos de computadoras paralelas, por lo que es natural preguntarse si hay alguna forma de clasificarlas dentro de alguna taxonomía. Muchos investigadores lo han intentado, con regular éxito (Flynn, 1972; Gajski y Pier, 1985; Treleaven, 1985). Lamentablemente, todavía no ha surgido el Carlos Linneo† de la computación en paralelo. El único esquema que se usa mucho es el de Flynn, pero hasta el de él es, en el mejor de los casos, una aproximación muy burda (figura 8-13).

Flujos de instrucciones	Flujos de datos	Nombre	Ejemplos
1	1	SISD	Máquina clásica de von Neumann
1	Varios	SIMD	Supercomputadora vectorial, procesador de arreglos
Varios	1	MISD	Tal vez ninguno
Varios	Varios	MIMD	Multiprocesador, multicomputadora

Figura 8-13. Taxonomía de Flynn para computadoras paralelas.

La clasificación de Flynn se basa en dos conceptos: flujos de instrucciones y flujos de datos. Un flujo de instrucciones corresponde a un contador de programa. Un sistema que tiene n CPU tiene n contadores de programa, y por tanto n flujos de instrucciones.

Un flujo de datos consiste en un conjunto de operandos. El ejemplo de cálculo de temperaturas que se dio antes tiene varios flujos de datos, uno para cada sensor.

Los flujos de instrucciones y de datos son, hasta cierto punto, independientes, por lo que existen cuatro combinaciones, las cuales se enumeran en la figura 8-13. SISD es la computadora secuencial clásica de von Neumann; tiene un flujo de instrucciones, un flujo de datos y

† Carlos Linneo (1707-1778) fue el biólogo sueco que ideó el sistema que ahora se usa para clasificar todas las plantas y animales según su reino, phylum, clase, orden, familia, género y especie.

hace una cosa a la vez. Las máquinas SIMD tienen una sola unidad de control que emite una instrucción a la vez, pero tienen múltiples ALU para ejecutarla con varios conjuntos de datos simultáneamente. La ILLIAC IV (figura 2-8) es el prototipo de las máquinas SIMD. Existen máquinas SIMD modernas y se usan para cálculos científicos.

Las máquinas MISD son una categoría un tanto extraña, en la que varias instrucciones operan con un mismo dato. No se sabe a ciencia cierta si existen máquinas de este tipo, aunque algunas personas clasifican a las máquinas con filas de procesamiento como MISD.

Por último, tenemos las MIMD, que no son más que múltiples CPU independientes que operan como parte de un sistema mayor. Casi todos los procesadores paralelos pertenecen a esta categoría. Tanto los multiprocesadores como las multicomputadoras son máquinas MIMD.

La taxonomía de Flynn termina aquí, pero la hemos extendido en la figura 8-14. Hemos dividido a SIMD en dos subgrupos. El primero es el de las supercomputadoras numéricas y otras máquinas que operan con vectores, realizando la misma operación con cada elemento del vector. El segundo es para las máquinas tipo paralelo, como la ILLIAC IV, en las que una unidad de control maestra, difunde instrucciones a muchas ALU independientes.

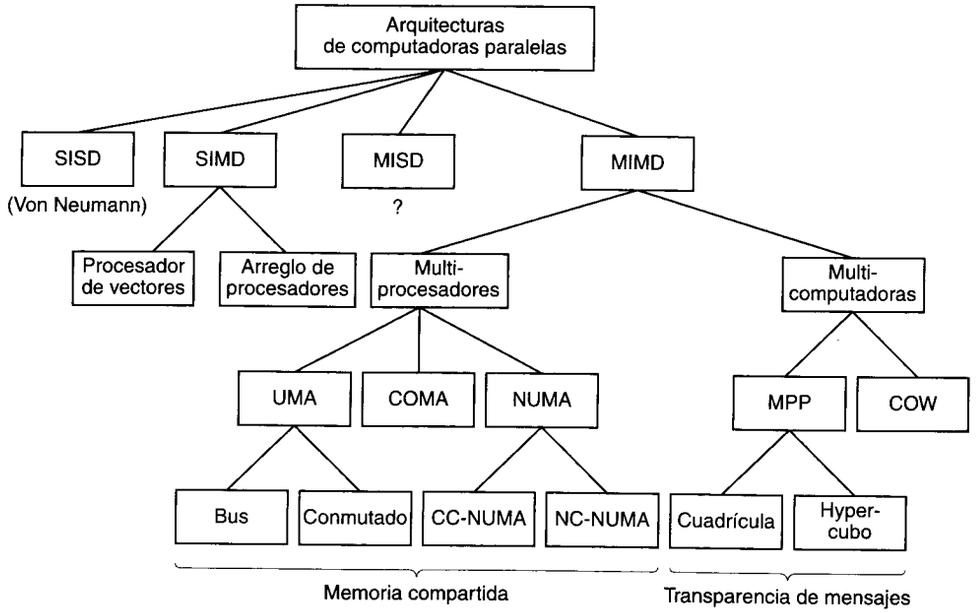


Figura 8-14. Una taxonomía de computadoras paralelas.

En nuestra taxonomía, la categoría MIMD se ha dividido en multiprocesadores (máquinas con memoria compartida) y multicomputadoras (máquinas que transfieren mensajes). Existen tres clases de multiprocesadores, que se distinguen por la forma en que se implementa la memoria compartida: **acceso uniforme a la memoria** (UMA, *Uniform Memory Access*),

acceso no uniforme a la memoria (NUMA, *NonUniform Memory Access*) y **sólo acceso a memoria caché** (COMA, *Cache Only Memory Access*). Estas categorías existen porque en los multiprocesadores grandes la memoria normalmente se divide en varios módulos. Las máquinas UMA tienen la propiedad de que cada CPU tiene el mismo tiempo de acceso a todos los módulos de memoria. En otras palabras, cada palabra de memoria se puede leer con la misma rapidez que cualquier otra palabra de memoria. Si esto es técnicamente imposible, las referencias más rápidas se frenan de modo que tarden lo mismo que las más lentas y los programadores no noten las diferencias. Éste es el significado de "uniforme" aquí. Esta uniformidad hace que el desempeño sea predecible, lo cual es un factor importante para escribir código eficiente.

En contraste, un multiprocesador NUMA no posee esta propiedad. Es común que haya un módulo de memoria cercano a cada CPU y que el acceso a ese módulo sea mucho más rápido que el acceso a módulos distantes. El resultado es que, por razones de desempeño, es importante dónde se coloquen el código y los datos. Las máquinas COMA tampoco son uniformes, pero de diferente manera. Estudiaremos cada uno de estos tipos y sus subcategorías con detalle más adelante.

La otra categoría principal de máquinas MIMD comprende las multicomputadoras que, a diferencia de los multiprocesadores, no tienen una memoria primaria compartida en el nivel arquitectónico. En otras palabras, el sistema operativo de una CPU de multicomputadora no puede acceder a memoria conectada a una CPU distinta con sólo ejecutar una instrucción LOAD; tiene que enviar un mensaje explícito y esperar una respuesta. La capacidad del sistema operativo para leer una palabra distante con sólo emitir una LOAD es lo que distingue a los multiprocesadores de las multicomputadoras. Como ya dijimos, aun en una multicomputadora los programas de usuario podrían tener la capacidad para acceder a memoria remota usando instrucciones LOAD y STORE, pero esta ilusión es mantenida por el sistema operativo, no por el hardware. La diferencia es sutil, pero muy importante. Dado que las multicomputadoras no tienen acceso directo a memoria remota, suele describírseles como máquinas **sin acceso a memoria remota**, (NORMA, *NO Remote Memory Access*).

Las multicomputadoras se pueden dividir a grandes rasgos en dos categorías. La primera categoría contiene los **procesadores masivamente paralelos** (MPP, *Massively Parallel Processors*), que son supercomputadoras caras que consisten en muchas CPU acopladas estrechamente por una red de interconexión patentada de alta velocidad. La Cray T3E y la SP/2 de IBM son ejemplos muy conocidos.

La otra categoría consiste en PC o estaciones de trabajo normales, tal vez montadas en anaqueles, y conectadas mediante tecnología comercial ordinaria. En el aspecto lógico no hay mucha diferencia, pero las enormes supercomputadoras que cuestan muchos millones de dólares se usan de diferente manera que las redes de PC armadas por los usuarios a una fracción del precio de un MPP. Estas máquinas "hechas en casa" reciben diversos nombres, como **redes de estaciones de trabajo** (NOW, *Network of Workstations*) y **cúmulos de estaciones de trabajo** (COW, *Cluster of Workstations*).

En las secciones que siguen examinaremos con cierto detalle las principales categorías: SIMD, MIMD/multiprocesadores y MIMD/multicomputadoras. La meta es presentar un panorama general de cada tipo, sus subcategorías y sus principios de diseño fundamentales. Se usarán varios ejemplos como ilustración.

8.2 COMPUTADORAS SIMD

Las computadoras SIMD (un solo flujo de instrucciones, múltiples flujos de datos; *Single Instruction stream Multiple Data stream*) se usan para resolver problemas científicos y de ingeniería que son computacionalmente intensivos e implican estructuras de datos regulares como vectores y arreglos. Estas máquinas se caracterizan por tener una sola unidad de control que ejecuta instrucciones una por una, pero cada instrucción opera con varios datos. Las dos especies principales de computadoras SIMD son los arrreglos de procesadores y los procesadores de vectores, que examinaremos a continuación.

8.2.1 Arreglos de procesadores

La idea de un arreglo de procesadores surgió hace más de 40 años (Unger, 1958), pero tuvieron que pasar 10 años para que se construyera el primero, ILLIAC IV, y entrara en servicio (para la NASA). Desde entonces, varias compañías han construido arreglos de procesadores comerciales, como el CM-2 de Thinking Machines y el Maspar MP-2, pero ninguno de ellos ha tenido un éxito comercial espectacular.

La idea en que se basa el **arreglo de procesadores** es que una sola unidad de control proporciona las señales que controlan muchos elementos de procesamiento, como se muestra en la figura 2-7. Cada elemento de procesamiento consiste en una CPU o ALU ampliada y por lo regular un poco de memoria local. Puesto que una sola unidad de control está alimentando a todos los elementos de procesamiento, éstos operan de forma sincronizada.

Aunque todos los arreglos de procesadores siguen este modelo general, hay varios aspectos de diseño en los que pueden diferir. El primero es la estructura del elemento de procesamiento. Dichos elementos pueden ir desde los extraordinariamente simples hasta otros más complejos. En el extremo inferior, los elementos de procesamiento pueden ser poco más que una ALU de un bit, como en el caso del CM-2. En esa máquina, cada ALU tomaba dos operandos de un bit de su memoria local, más un bit de su palabra de situación del programa (digamos, el bit de acarreo). El resultado de la operación era un bit de datos y algunos bits indicadores. Para realizar una suma entera de 32 bits, la unidad de control tenía que difundir una instrucción de suma de un bit 32 veces. A 600 ns por instrucción, se requerían 19.2 1μs para efectuar una suma entera, más tiempo que en la IBM PC original. Sin embargo, si trabajaban todos los 65,536 elementos de procesamiento, era posible realizar 3000 millones de sumas por segundo, lo que daba un tiempo de suma efectivo de 300 picosegundos.

Otras opciones para el elemento de procesamiento incluyen una ALU de 8 bits, una ALU de 32 bits o una unidad más potente con capacidad de punto flotante. Hasta cierto punto, la decisión de qué incluir en el elemento de procesamiento depende de los objetivos de la máquina. Para triturar números, la capacidad de punto flotante es razonable (aunque su costo seguramente reducirá mucho el número de elementos de procesamiento), pero para recuperación de información tal vez no se necesite.

Un segundo aspecto de diseño es la forma de interconectar los elementos de procesamiento. En principio, casi todas las topologías que se ilustran en la figura 8-4 son posibles candidatos. Las cuadrículas rectangulares son una opción común porque se ajustan bien a

muchos problemas bidimensionales que implican matrices e imágenes y su escala se puede aumentar con facilidad ya que la adición de más procesadores automáticamente aumenta el ancho de banda.

Una tercera cuestión de diseño es el grado de autonomía local que los elementos de procesamiento tienen. En el tipo de diseño que estamos describiendo, la unidad de control especifica la instrucción que debe ejecutarse, pero en muchos procesadores de arreglos cada elemento de procesamiento puede optar por ejecutar o no ejecutar una instrucción, con base en datos locales como bits de código de condición. Esta característica añade mucha flexibilidad. Puesto que los procesadores de arreglos tienen un futuro incierto, no profundizaremos más en ellos.

8.2.2 Procesadores vectoriales

El otro tipo de máquina SIMD es el **procesador vectorial**, que ha tenido un éxito comercial mucho mayor. Un linaje de máquinas diseñadas por Seymour Cray para Cray Research (ahora parte de Silicon Graphics), comenzando con la Cray-1 en 1976 y continuando hasta la C90 y la T90, ha dominado la computación científica durante décadas. En esta sección examinaremos los principios fundamentales que se aplican en estas computadoras de alto desempeño.

Una aplicación de trituración de números típica contiene muchos enunciados como

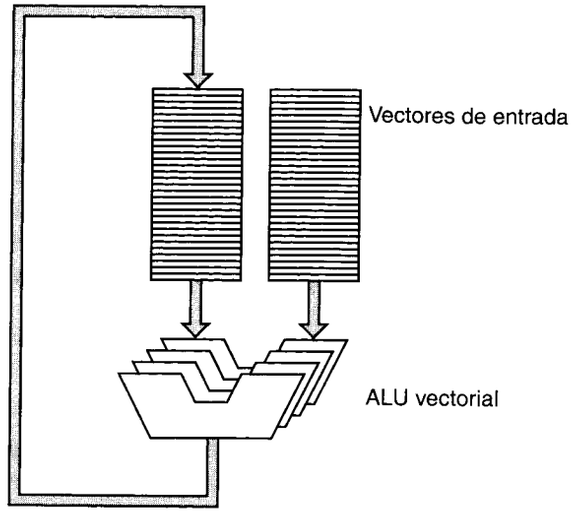
```
for (i = 0; i < n; i++) a[i] = b[i] + c[i];
```

donde *a*, *b* y *c* son **vectores**, es decir, arreglos de números, casi siempre en punto flotante. El ciclo ordena a la computadora sumar los *i*-ésimos elementos de *b* y *c* y almacenar el resultado en el *i*-ésimo elemento de *a*. Técnicamente, el programa especifica que los elementos deben sumarse en sucesión, pero normalmente el orden no importa.

En la figura 8-15 se muestra una posible arquitectura SIMD apropiada para este tipo de procesamiento vectorial. La máquina recibe dos vectores de *n* elementos como entradas y opera con los elementos correspondientes en paralelo utilizando una ALU vectorial que puede operar con los *n* elementos simultáneamente y produce como resultado un vector. Los vectores de entrada y de salida se pueden almacenar en la memoria o en registros vectoriales especiales.

Las computadoras vectoriales también necesitan efectuar operaciones escalares (no vectoriales) y operaciones mixtas escalares-vectoriales. Los tipos básicos de operaciones vectoriales se enumeran en la figura 8-16. La primera, f_1, efectúa alguna operación, como coseno o raíz cuadrada, con cada elemento de un solo vector. La segunda, f_2, toma un vector como entrada y produce un escalar como salida. Un ejemplo típico es sumar todos los elementos. La tercera, f_3, realiza una operación diádica con dos vectores, como sumar los elementos correspondientes. Por último, f_4 combina un operando escalar con uno vectorial. Un ejemplo típico es multiplicar cada elemento de un vector por una constante. A veces es más rápido convertir el escalar en un vector, cuyos elementos son todos iguales al escalar, y luego realizar la operación con dos vectores.

Todas las operaciones comunes con vectores pueden construirse utilizando estas formas. Por ejemplo, el producto interior (producto punto) de dos vectores consiste en multiplicar primero los elementos correspondientes (f_3) y obtener después la sumatoria del resultado (f_2).



Figur a8-15. ALU vectorial.

Operación	Ejemplos
$A_i = f_1(B_i)$	f_1 = coseno, raíz cuadrada
Escalar = $f_2(A)$	f_2 = sumatoria, mínimo
$A_i = f_3(Bi, C_i)$	f_3 = sumar, restar
$A_i = f_4$(escalar, B_i)	f_4 = multiplicar B_i por una constante

Figura 8-16. Diversas combinaciones de operaciones vectoriales y escalares.

En la práctica, pocas supercomputadoras se construyen en realidad como lo que se muestra en la figura 8-15. La razón no es técnica —ciertamente es factible construir tales máquinas— sino económica. Un diseño con, digamos, 64 ALUs de muy alta velocidad sería demasiado costoso, incluso para una supercomputadora.

El método que se emplea normalmente es combinar el procesamiento de vectores con el uso de filas de procesamiento. Las operaciones de punto flotante son muy complejas, requieren varios pasos, y cualquier operación de múltiples pasos es candidata para una implementación con filas de procesamiento. Si no está familiarizado con la aritmética de punto flotante, lea el Apéndice B.

Como ejemplo de uso de filas de procesamiento para operaciones de punto flotante, considere la figura 8-17. En este ejemplo, un número normalizado tiene una mantisa mayor o igual que 1, pero menor que 10. De lo que se trata aquí es de restar 9.212×10^{11} a 1.082×10^{12}.

Para restar dos números de punto flotante primero es necesario ajustarlos de modo que sus exponentes tengan el mismo valor. En este ejemplo podemos convertir el sustraendo (el

Paso	Nombre	Valores
1	Traer operandos	$1.082 \times 10^{12} - 9.212 \times 10^{11}$
2	Ajustar exponente	$1.082 \times 10^{12} - 0.9212 \times 10^{12}$
3	Ejecutar resta	0.1608×10^{12}
4	Normalizar resultado	1.608×10^{11}

Figura 8-17. Pasos de una resta de punto flotante.

número que se resta) a 0.9212×10^{12} o bien convertir el minuendo (el número del cual se resta) a 10.82×10^{11}. En general, ambas conversiones implican riesgos. Elevar un exponente podría causar un subdesbordamiento de la mantisa, y bajar un exponente podría causar un desbordamiento de la mantisa. El subdesbordamiento es el menos grave de los dos, porque el número se puede aproximar con 0, así que seguiremos la primera ruta. Después de ajustar ambos exponentes a 12, tenemos los valores que se muestran en el paso 2 de la figura 8-17. A continuación realizamos la resta, seguida de la normalización del resultado.

El uso de filas de procesamiento puede aplicarse al ciclo for loop que se presentó al principio de esta sección. En la figura 8-18 vemos un sumador de punto flotante con filas de procesamiento de cuatro etapas. En cada ciclo, la primera etapa obtiene un par de operandos. Luego, la segunda etapa ajusta el exponente del operando más pequeño de modo que coincida con el del más grande. La tercera etapa efectúa la operación, y la cuarta normaliza el resultado. De este modo, un resultado sale de la fila de procesamiento en cada ciclo.

	Ciclo						
Paso	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Buscar operandos	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6	B_7, C_7
Ajustar exponente		B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6
Ejecutar operación			B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5
Normalizar resultado				B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4

Figura 8-18. Sumador de punto flotante con filas de procesamiento.

Una diferencia importante entre usar filas de procesamiento para operaciones con vectores y usarla para la ejecución de instrucciones de propósito general es la ausencia de saltos al operar con vectores. Cada ciclo se aprovecha al máximo, y no se desperdician ranuras.

La supercomputadora vectorial Cray-1

Las supercomputadoras por lo regular tienen múltiples ALUs, cada una especializada en alguna operación específica, y todas capaces de operar en paralelo. Por ejemplo, consideremos la Cray-1, una de las primeras supercomputadoras. Aunque ya no se usa, tiene una arquitectura simple tipo RISC que la convierte en un buen objeto de estudio, y su arquitectura perdura en muchas supercomputadoras vectoriales modernas.

Como la generalidad de las máquinas RISC, la Cray-1 está orientada hacia los registros, con instrucciones en su mayoría de 16 bits que consisten en un código de operación de siete bits y tres números de registro de tres bits para los tres operandos. Hay cinco clases de registros, como se muestra en la figura 8-19. Los ocho registros A de 24 bits sirven para direccionar la memoria. Los 64 registros B de 24 bits sirven para guardar registros A cuando no se necesitan, en lugar de escribirlos de vuelta en la memoria. Los ocho registros S de 64 bits sirven para contener cantidades escalares (enteras y de punto flotante). Los valores de estos registros pueden utilizarse como operandos para operaciones tanto enteras como de punto flotante. Los 64 registros T de 64 bits son almacenamiento adicional para los registros S, otra vez con objeto de reducir el número de instrucciones LOAD y STORE.

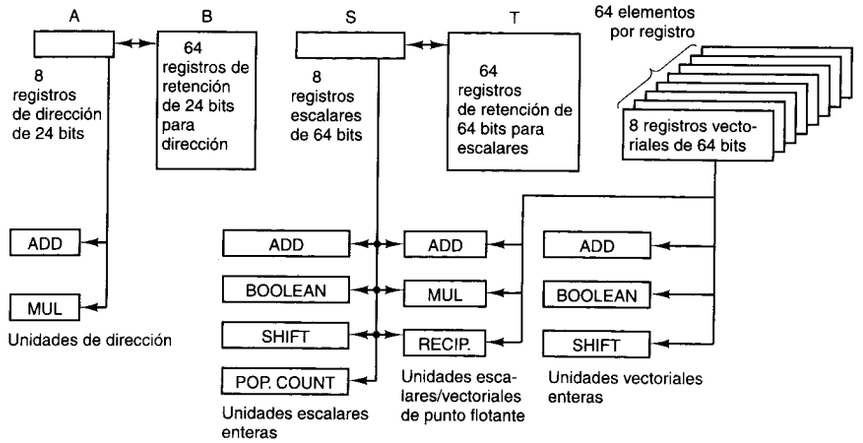


Figura 8-19. Registros y unidades funcionales de la Cray-1.

La parte más interesante del conjunto de registros de la Cray-1 es el grupo de ocho registros vectoriales. Cada registro puede contener un vector de punto flotante de 64 elementos. Es posible sumar, restar o multiplicar dos vectores con una sola instrucción de 16 bits. No es posible dividir, pero sí calcular recíprocos. Los registros vectoriales se pueden cargar de la memoria y guardarse en ella, pero tales transferencias son costosas, y lo mejor es minimizarlas. Todas las operaciones con vectores usan operandos en registros.

No todas las supercomputadoras tienen la propiedad estilo RISC de requerir que todos los operandos estén en registros. La CDC Cyber 205, por ejemplo, realizaba operaciones con vectores en la memoria. Este enfoque permitía manejar vectores de longitud arbitraria pero hacía mucho más lenta la máquina porque la memoria era un cuello de botella importante.

La Cray-1 tiene 12 unidades funcionales distintas, como se ilustra en la figura 8-19. Dos son para aritmética con direcciones de 24 bits y cuatro son para operaciones con escalares enteros de 64 bits. Al igual que su antepasada, la CDC 6600, la Cray-1 carece de una unidad para efectuar multiplicación entera (aunque sí tiene una para multiplicación de punto flotante). Las seis unidades restantes trabajan con vectores, y su estructura interna utiliza muchas filas de procesamiento. Las unidades de suma, multiplicación y recíproco también operan con números escalares de punto flotante, además de con vectores.

Al igual que muchas otras computadoras vectoriales, la Cray-1 permite **encadenar operaciones**. Por ejemplo, una forma de calcular

```
R1 = R1 * R2 + R3
```

donde R1, R2 y R3 son registros vectoriales, sería efectuar la multiplicación vectorial, elemento por elemento, guardar el resultado en algún lugar, y luego efectuar la suma vectorial. Con el encadenamiento, tan pronto como se han multiplicado los primeros elementos, el producto puede ir directamente al sumador, junto con el primer elemento de R3. No se requiere un almacenamiento intermedio. Esta técnica mejora considerablemente el desempeño.

Como comentario final acerca de la Cray-1, resulta interesante examinar su desempeño absoluto. El reloj opera a 80 MHz y tiene una memoria principal de 8 MB. En su época (de mediados a fines de los años setenta) era la computadora más potente del mundo. Hoy día ya no es posible comprar una PC con un reloj tan lento ni una memoria tan pequeña; ya nadie las fabrica. Esta observación es un testimonio a la rapidez con que avanza la industria de las computadoras.

8.3 MULTIPROCESADORES CON MEMORIA COMPARTIDA

Como vimos en la figura 8-14, los sistemas MIMD se pueden dividir en multiprocesadores y multicomputadoras. En esta sección examinaremos los multiprocesadores; la siguiente estará dedicada a las multicomputadoras. Un multiprocesador es un sistema de cómputo que tiene varias CPU y un solo espacio de direcciones visible para todas las CPU. La máquina ejecuta una copia del sistema operativo, con un conjunto de tablas que incluyen las tablas en las que se lleva la contabilidad de cuáles páginas de memoria están ocupadas y cuáles están libres. Cuando un proceso se bloquea, su CPU guarda su estado en las tablas del sistema operativo y busca en esas tablas otro proceso que pueda ejecutar. Es esta imagen de sistema único lo que distingue un multiprocesador de una multicomputadora.

Un multiprocesador, como todas las computadoras, debe tener dispositivos de E/S, como discos, adaptadores de red y otros equipos. En algunos sistemas de multiprocesador sólo ciertas CPU tienen acceso a los dispositivos de E/S, y por tanto tienen una función de E/S especial. En otros, cada CPU tiene el mismo acceso a todos los dispositivos de E/S. Si toda CPU tiene igual acceso a todos los módulos de memoria y todos los dispositivos de E/S, y el sistema operativo la trata como intercambiable con las demás, el sistema se describe como **multiprocesador simétrico** (SMP, *Symmetric MultiProcessor*). La explicación que sigue está orientada hacia este tipo de sistemas.

8.3.1 Semántica de la memoria

Aunque todos los multiprocesadores presentan a las CPU la imagen de un solo espacio de direcciones compartido, a menudo hay muchos módulos de memoria presentes, y cada uno contiene alguna porción de la memoria física. Las CPU y las memorias a menudo están conectadas por una red de interconexión compleja, como se explicó en la sección 8.1.2. Va-

rias CPU pueden estar intentando leer una palabra de memoria al mismo tiempo que varias otras CPU están tratando de escribir la misma palabra, y algunos de los mensajes de solicitud podrían rebasar a otros en el camino y entregarse en un orden distinto de aquel en que fueron emitidos. Si a este problema sumamos la existencia de múltiples copias de algunos bloques de memoria (por ejemplo, en cachés), el resultado puede convertirse fácilmente en un caos si no se toman medidas estrictas para evitarlo. En esta sección veremos lo que significa realmente la memoria compartida y cómo pueden responder razonablemente las memorias en estas condiciones.

Una perspectiva de la semántica de memoria es verla como un contrato entre el software y el hardware de memoria (Adve y Hill, 1990). Si el software acepta cumplir con ciertas reglas, la memoria conviene en entregar ciertos resultados. La discusión se centra entonces en la naturaleza de las reglas. Estas reglas se denominan **modelos de consistencia** y se han propuesto e implementado varios de ellos.

Para tener una idea de la naturaleza del problema, supongamos que la CPU 0 escribe el valor 1 en alguna palabra de memoria y poco después la CPU 1 escribe el valor 2 en la misma palabra. Ahora la CPU 2 lee la palabra y obtiene el valor 1. ¿El dueño de la computadora debe llevar la máquina al taller para que la reparen? Todo depende de lo que la memoria prometió (su contrato).

Consistencia estricta

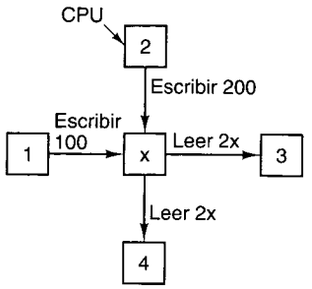
El modelo más sencillo es el de **consistencia estricta**. Con este modelo, cualquier lectura de una posición x siempre devuelve el valor de la escritura más reciente en x. Los programadores aman a este modelo, pero en la práctica es imposible de implementar de otra manera que no sea tener un solo módulo de memoria que atiende todas las solicitudes bajo el régimen de primero que llega, primero que se atiende, sin uso de cachés ni repetición de datos. Una implementación así convertiría a la memoria en un enorme cuello de botella y por ello no es un candidato serio, lo cual es lamentable.

Consistencia secuencial

El siguiente mejor modelo se llama **consistencia secuencial** (Lamport, 1979). La idea aquí es que, en presencia de múltiples solicitudes de lectura y escritura, el hardware escoge (de forma no determinista) cierta intercalación de todas las solicitudes, pero todas las CPU perciben el mismo orden.

Para entender qué significa esto, consideremos un ejemplo. Suponga que la CPU 1 escribe el valor 100 en la palabra x, y 1 ns después la CPU 2 escribe el valor 200 en la palabra x. Ahora suponga que 1 ns después de emitirse la segunda instrucción de escritura (que no necesariamente ha terminado de ejecutarse) otras dos CPU, la 3 y la 4, leen la palabra x dos veces en rápida sucesión, como se muestra en la figura 8-20(a). En la figura 8-20(b)-(d) se muestran tres posibles ordenamientos de los seis sucesos (dos escrituras y cuatro lecturas). En la figura 8-20(b), la CPU 3 obtiene (200, 200) y la CPU 4 obtiene (200, 200). En la figura 8-20(c), las CPU obtienen (100, 200) y (200, 200), respectivamente. En la figura 8-20(d) las

CPU obtienen (100, 100) y (200, 200), respectivamente. Todas estas posibilidades son válidas, además de otras que no se muestran.



(a)

(b)	(c)	(d)
E100	E100	E200
E200	L3 = 100	L4 = 200
L3 = 200	E200	E100
L3 = 200	L4 = 200	L3 = 100
L4 = 200	L3 = 200	L4 = 100
L4 = 200	L4 = 200	L3 = 100

Figura 8-20. (a) Dos CPU escriben y dos CPU leen una palabra de memoria en común. (b) – (d) Tres posibles intercalados en el tiempo de las dos escrituras y cuatro lecturas.

Sin embargo —y ésta es la esencia de la consistencia secuencial— pase lo que pase, una memoria secuencialmente consistente nunca permitirá que la CPU 3 obtenga (100, 200) mientras la CPU 4 obtiene (200, 100). Si ocurriera esto, implicaría que, según la CPU 4, la escritura de 100 por la CPU 1 terminó antes que la escritura de 200 por la CPU 2. No hay problema con eso, pero también implicaría que, según la CPU 3, la escritura de 200 por la CPU 2 terminó antes que la escritura de 100 por la CPU 1. Por sí solo, este resultado también es posible. El problema es que la consistencia secuencial garantiza que hay un solo ordenamiento global de todas las escrituras que todas las CPU ven. Si la CPU 3 observa que 100 se escribió primero, entonces la CPU 4 también deberá ver este orden.

Aunque la consistencia secuencial no es una regla tan potente como la consistencia estricta, de todos modos es muy útil. En realidad, lo que está diciendo es que, cuando suceden varias cosas al mismo tiempo, existe un orden verdadero en el que ocurren, tal vez determinado por los tiempos y la casualidad, pero todos los procesadores observan este mismo orden. Aunque esta afirmación puede parecer obvia, a continuación analizaremos modelos de consistencia que no garantizan ni siquiera esto.

Consistencia del procesador

Un modelo de consistencia menos rígido, pero que es más fácil de implementar en multiprocesadores grandes, es la **consistencia del procesador** (Goodman, 1989). Este modelo tiene dos propiedades:

1. Todas las CPU ven las escrituras de cualquier CPU en el orden en que se emitieron.
2. Para cada palabra de la memoria, todas las CPU ven todas las escrituras en ella en el mismo orden.

Ambos puntos son importantes. El primero dice que si la CPU 1 emite escrituras con los valores 1A, 1B y 1C a alguna localidad de memoria en ese orden, entonces todos los demás procesadores las verán en ese orden también. En otras palabras, cualquier otro procesador que esté en un ciclo corto observando 1A, 1B y 1C leyendo las palabras escritas nunca verá el valor escrito por 1B y luego el valor escrito por 1A, etc. El segundo punto es necesario para asegurar que cada localidad de memoria tendrá un valor inequívoco después de que varias CPU escriben en ella y después se detienen. Todo mundo debe estar de acuerdo en quién fue el último.

Incluso con estas restricciones, el diseñador tiene mucha flexibilidad. Considere lo que sucede si la CPU 2 emite las escrituras 2A, 2B y 2C al mismo tiempo que la CPU 1 emite sus tres escrituras. Otras CPU que estén leyendo activamente la memoria observarán cierta intercalación de las seis escrituras, digamos 1A, 1B, 2A, 2B, 1C, 2C o 2A, 1A, 2B, 2C, 1B, 1C o muchas otras. La consistencia de procesador *no* garantiza que toda CPU verá el mismo ordenamiento (a diferencia de la consistencia secuencial, que sí ofrece esta garantía). Por tanto, está permitido que el hardware se comporte de tal manera que algunas CPU vean el primero de los dos ordenamientos que pusimos de ejemplo, que algunas vean el segundo y que algunas vean otros ordenamientos. Lo que *sí* se garantiza es que ninguna CPU verá una secuencia en la que 1B ocurre antes que 1A, etc. El orden en que cada CPU realiza sus escrituras se observa en todos lados.

Consistencia débil

Nuestro siguiente modelo, **consistencia débil**, ni siquiera garantiza que las escrituras de una sola CPU se verán en orden (Dubois *et al.*, 1986). En una memoria con consistencia débil, una CPU podría ver 1A antes que 1B y otra CPU podría ver 1A después de 1B. Sin embargo, a fin de poner cierto orden en el caos, las memorias con consistencia débil tienen variables de sincronización o una operación de sincronización. Cuando se ejecuta una sincronización, todas las escrituras pendientes se terminan y no se inicia ninguna nueva hasta que todas las viejas se llevan a cabo y la sincronización misma termina. Lo que una sincronización hace realmente es "vaciar la fila de procesamiento" y poner la memoria en un estado estable sin operaciones pendientes. Las operaciones de sincronización son en sí secuencialmente consistentes; es decir, cuando varias CPU las emiten, se escoge algún orden, pero todas las CPU ven el mismo orden.

En la consistencia débil, el tiempo se divide en épocas bien definidas delimitadas por las sincronizaciones (secuencialmente consistentes), como se ilustra en la figura 8-21. No se garantiza un orden relativo para 1A y 1B, y diferentes CPU podrían ver las dos escrituras en diferente orden; es decir, una CPU podría ver 1A y luego 1B, mientras que otra CPU podría ver 1B y después 1A. Esta situación se permite. Sin embargo, todas las CPU ven 1B antes que 1C porque la primera operación de sincronización obliga a que 1A, 1B y 2A terminen antes que se permita el inicio de 1C, 2B, 3A o 3B. Así pues, al realizar operaciones de sincronización, el software puede imponer cierto orden sobre la secuencia de sucesos, aunque no sin costo porque el vaciado de la fila de procesamiento de memoria toma cierto tiempo.

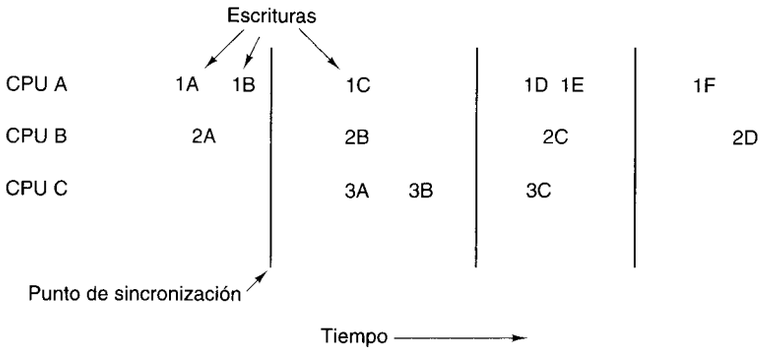


Figura 8-21. Una memoria con consistencia débil usa operaciones de sincronización para dividir el tiempo en épocas secuenciales.

Consistencia de liberación

La consistencia débil tiene el problema de que es muy ineficiente porque debe terminar todas las operaciones de memoria pendiente y detener las nuevas hasta que terminen las actuales. La **consistencia de liberación** mejora la situación al adoptar un modelo parecido al de secciones críticas (Gharachorloo *et al.*, 1990). La idea en que se basa este modelo es que, cuando un proceso sale de una sección crítica no es necesario hacer que todas las escrituras se lleven a cabo inmediatamente; sólo es necesario asegurarse de que terminen antes de que cualquier proceso ingrese otra vez en esa sección crítica.

En este modelo, la operación de sincronización que ofrece la consistencia débil se divide en dos operaciones distintas. Si quiere leer o escribir una variable de datos compartida, una CPU (es decir, su software) debe efectuar primero una operación acquire (adquirir) sobre la variable de sincronización para obtener acceso exclusivo a los datos compartidos. Luego la CPU puede usarlos como desee, leyéndolos y escribiéndolos a voluntad. Cuando termina, la CPU efectúa una operación release (liberar) sobre la variable de sincronización para indicar que ya terminó. La release no hace que las escrituras pendientes finalicen, pero ella misma no termina sino hasta que todas las escrituras emitidas previamente se llevan a cabo. Además, no se impide que nuevas operaciones de memoria se inicien de inmediato.

Cuando se emite la siguiente acquire, se verifica si ya se finalizaron todas las operaciones release previas. Si no, la acquire se detiene hasta que todas terminan (y por tanto terminan también todas las escrituras realizadas antes). De este modo, si la siguiente acquire ocurre suficiente tiempo después de la release más reciente, no tiene que esperar antes de iniciar y se puede ingresar en la sección crítica sin retraso. Si ocurre demasiado pronto después de una release, la acquire (y todas las instrucciones que le siguen) se retrasarán hasta que todas las release pendientes terminen, lo que garantiza que las variables dentro de la sección crítica se han actualizado. Este esquema es un poco más complicado que la consistencia débil, pero tiene la importante ventaja de que no retrasa instrucciones con tanta frecuencia a fin de mantener la consistencia.

8.3.2 Arquitecturas SMP basadas en el bus UMA

Los procesadores más sencillos se basan en un solo bus, como se ilustra en la figura 8-22(a). Dos o más CPU y uno o más módulos de memoria usan el mismo bus para comunicarse. Cuando una CPU quiere leer una palabra de memoria, primero verifica si el bus está ocupado. Si el bus está ocioso, la CPU coloca en él la dirección de la palabra que desea, habilita unas cuantas señales de control, y espera hasta que la memoria coloca la palabra deseada en el bus.

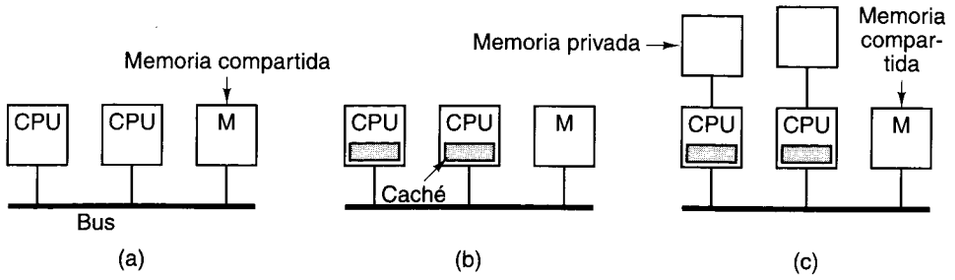


Figura 8-22. Tres multiprocesadores basados en bus. (a) Sin cachés. (b) Con cachés. (c) Con cachés y memorias privadas.

Si el bus está ocupado cuando una CPU quiere leer o escribir en la memoria, la CPU esperará hasta que el bus se desocupe. Aquí radica el problema de este diseño. Con dos o tres CPU, la contención por el bus será manejable; con 32 o 64 será intolerable. El sistema estará limitado totalmente por el ancho de banda del bus, y casi todas las CPU estarán ociosas la mayor parte del tiempo.

La solución a este problema es añadir una caché a cada CPU, como se muestra en la figura 8-22(b). La caché puede estar dentro del chip de la CPU, junto al chip, en la tarjeta del procesador, o en alguna combinación de las tres posibilidades. Puesto que ahora muchas lecturas se pueden atender con la caché local, habrá mucho menos tráfico de bus, y el sistema podrá manejar más CPU.

Otra posibilidad más es el diseño de la figura 8-22(c), en el que cada CPU tiene no sólo una caché, sino también una memoria privada local a la que accede a través de un bus dedicado (privado). Para usar esta configuración de manera óptima, el compilador debe colocar todo el texto del programa, cadenas, constantes y demás datos sólo de lectura, pilas y variables locales en las memorias privadas. Entonces, la memoria compartida se usará sólo para variables compartidas escribibles. En casi todos los casos, esta colocación cuidadosa reduce considerablemente el tráfico de bus, pero requiere la cooperación activa del compilador.

Caché espía

Aunque los argumentos de desempeño antes mencionados son ciertamente válidos, hemos pasado por alto un problema fundamental. Supongamos que la memoria es secuencialmente consistente. ¿Qué sucede si la CPU 1 tiene una línea en su caché, y luego la CPU 2 trata de

leer una palabra de la misma línea de caché? Si no se fijan reglas especiales, ella también traerá una copia a su caché. En principio, tener la misma línea en dos cachés es aceptable. Supongamos ahora que la CPU 1 modifica la línea e inmediatamente después la CPU 2 lee su copia de la línea de su caché. La CPU 2 obtendrá **datos obsoletos**, y esto viola el contrato entre el software y la memoria. Al programa que se ejecuta en la CPU 2 no le hará gracia.

Este problema, llamado en la literatura problema de **coherencia de caché** o **consistencia de caché**, es muy grave. Si no se soluciona, no es posible usar cachés, y los multiprocesadores orientados a bus estarán limitados a dos o tres CPU. En vista de la importancia de este problema, se han propuesto muchas soluciones (por ejemplo, Goodman, 1983; Papamarcos y Patel, 1984). Aunque todos estos algoritmos de uso de cachés, llamados **protocolos de coherencia de cachés**, difieren en los detalles, todos ellos evitan que aparezcan diferentes versiones de la misma línea de caché simultáneamente en dos o más cachés.

En todas las soluciones, el controlador de caché se diseña especialmente de modo que pueda vigilar el bus, observando todas las solicitudes de bus de otras CPU y cachés y emprendiendo acciones en algunos casos. Estos dispositivos se llaman **cachés espía** (*snooping caches* o *snoopy caches*) porque "espían" el bus. El conjunto de reglas implementadas por los cachés, las CPU y la memoria para evitar que diferentes versiones de los datos aparezcan en varias cachés constituyen el protocolo de coherencia de cachés. La unidad de transferencia y almacenamiento de una caché se denomina **línea de caché** y suele ser de 32 o 64 bytes.

El protocolo de coherencia de caché más sencillo es el de **escritura a través**. La mejor manera de explicarlo es distinguir los cuatro casos que se muestran en la figura 8-23. Cuando una CPU trata de leer una palabra que no está en su caché (es decir, cuando hay un fallo de lectura), su controlador de caché carga en la caché la línea que contiene la palabra. La memoria proporciona la línea, y en este protocolo siempre está actualizada. Las lecturas subsecuentes (es decir, aciertos de caché) se pueden satisfacer con la caché.

Acción	Solicitud local	Solicitud remota
Fallo de lectura	Buscar datos en memoria	
Acierto de lectura	Usar datos de caché local	
Fallo de escritura	Actualizar datos en memoria	
Acierto de escritura	Actualizar caché y memoria	Invalidar entrada de caché

Figura 8-23. Protocolo de coherencia de caché de escritura a través. Las celdas vacías indican que no se hace nada.

Cuando hay un fallo de escritura, la palabra que se modificó se escribe en la memoria principal. La línea que contiene la palabra a la que se hizo referencia *no* se carga en la caché. Cuando hay un acierto de escritura, la caché se actualiza y la palabra se escribe también a través en la memoria principal. La esencia de este protocolo es que el resultado de todas las operaciones de escritura sea que la palabra escrita se escribe siempre en la memoria a fin de mantener a ésta actualizada en todo momento.

Examinemos otra vez todas estas acciones, pero esta vez desde el punto de vista del espía, que se muestra en la columna de la derecha en la figura 8-23. Llamemos a la caché que realiza las acciones caché 1, y al espía, caché 2. Cuando la caché 1 tiene un fallo de lectura, hace una solicitud de bus para traer una línea de la memoria. La caché 2 ve esto pero no hace nada. Cuando la caché 1 tiene un acierto de lectura, la solicitud se satisface localmente, y no ocurre ninguna solicitud de bus, por lo que la caché 2 no tiene conocimiento de los aciertos de lectura de la caché 1.

Las escrituras son más interesantes. Si la CPU 1 efectúa una escritura, la caché 1 emitirá una solicitud de escritura al bus, tanto en los fallos como en los aciertos. En todas las escrituras, la caché 2 verifica si tiene la palabra que se está escribiendo. Si no la tiene, desde su punto de vista se trata de una solicitud remota/fallo de escritura y no hace nada. (A fin de aclarar un punto sutil, observe que en la figura 8-23 un fallo remoto implica que la palabra no está presente en la caché del espía; no importa si estaba en la caché del originador o no. Así, una misma solicitud puede ser un acierto localmente y un fallo en el espía, o viceversa.)

Suponga ahora que la caché 1 escribe una palabra que *sí está* presente en la caché 2 (solicitud remota/acierto de escritura). Si la caché 2 no hace nada, tendrá datos obsoletos, por lo que marca como no válida la entrada de caché que contiene la palabra recién modificada. De hecho, lo que hace es sacar los datos de la caché. Puesto que todos las cachés espían todas las solicitudes de bus, cada vez que se escribe una palabra el efecto neto es actualizarla en la caché del originador, actualizarla en la memoria, y sacarla de todos los demás cachés. De este modo, se evita que haya versiones inconsistentes.

Desde luego, la CPU de la caché 2 está en libertad de leer la misma palabra ya en el siguiente ciclo. En ese caso, la caché 2 leerá la palabra de la memoria, que está actualizada. En ese punto, la caché 1, la caché 2 y la memoria tendrán copias idénticas de la palabra. Si cualquiera de las CPU efectúa una escritura entonces, se depurará la caché de la otra, y la memoria se actualizará.

Puede haber muchas variaciones de este protocolo básico. Por ejemplo, cuando hay un acierto de escritura, la caché espía normalmente invalida su entrada que contiene la palabra que se está escribiendo. Como alternativa, esa caché podría aceptar el valor nuevo y actualizar su caché en lugar de marcarlo como no válido. En lo conceptual, actualizar la caché es lo mismo que marcarla como no válido e inmediatamente leer la palabra de la memoria. En todos los protocolos de caché se debe escoger entre una **estrategia de actualizar** y una **estrategia de invalidar**. Estos protocolos tienen diferente desempeño bajo diferentes cargas. Los mensajes de actualización llevan una carga útil y por tanto son más grandes que los de invalidación, pero podrían evitar fallos de caché futuros.

Otra variante es cargar la caché espía cuando hay fallos de escritura. La corrección del algoritmo no cambia por esta carga, sólo su desempeño. La cuestión es: "¿Qué probabilidad hay de que una palabra recién escrita se escriba otra vez pronto?" Si tal probabilidad es alta, puede ser mejor cargar la caché cuando hay fallos de escritura, lo que se denomina **política de escribir y asignar**. Si la probabilidad es baja, es mejor no actualizar cuando hay fallos de escritura. Si la palabra se *lee* pronto, se cargará por el fallo de lectura de todos modos; poco se ganará cargándola después del fallo de escritura.

Al igual que con muchas soluciones sencillas, ésta es ineficiente. Cada operación de escritura accede a la memoria por el bus, de modo que aunque el número de CPU no sea muy

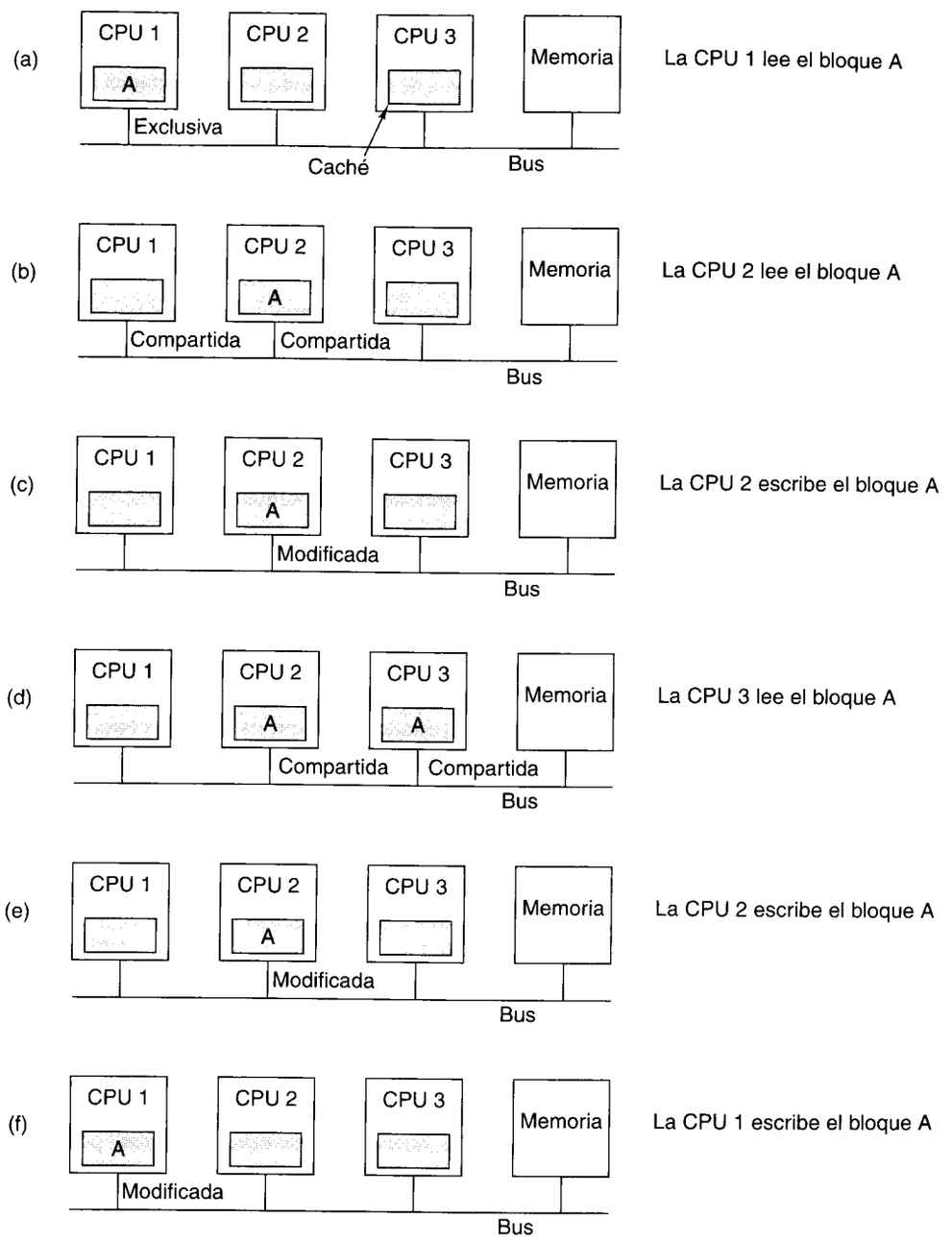


Figura 8-24. Protocolo de coherencia de caché MESI.

grande el bus de todos modos será un cuello de botella. A fin de restringir el tráfico de bus, se han inventado otros protocolos de caché. Todos ellos tienen la propiedad de que no todas las escrituras pasan directamente a la memoria. En vez de ello, cuando se modifica una línea de caché, se enciende un bit dentro de la caché que indica que la línea de caché es correcta pero la memoria no. Tarde o temprano, será preciso escribir esa línea sucia en la memoria, pero posiblemente después de que se haya escrito en ella muchas veces. Este tipo de protocolo se conoce como **protocolo de reescritura**.

El protocolo de coherencia de caché MESI

Un protocolo de coherencia de caché de escritura de vuelta muy utilizado es **MESI**, cuyo nombre proviene de las iniciales en inglés de los cuatro estados que usa (Papamarcos y Patel, 1984). Se basa en el **protocolo de escritura única** (Goodman, 1983) anterior. El protocolo MESI se usa en el Pentium II y muchas otras CPU para espiar el bus. Cada entrada de caché puede estar en uno de los cuatro estados siguientes:

1. No válida – La entrada de caché no contiene datos válidos.
2. Compartida – Varios cachés podrían contener la línea; la memoria está actualizada.
3. Exclusiva – Ninguna otra caché contiene la línea; la memoria está actualizada.
4. Modificada – La entrada es válida; la memoria no es válida; no existen copias.

Cuando la CPU arranca, todas las entradas de caché se marcan como no válidas. La primera vez que se lee la memoria, la línea a la que se hace referencia se trae al caché de la CPU que leyó la memoria y se marca como exclusiva (estado E), ya que es la única copia que hay en una caché, como se ilustra en la figura 8-24(a) para el caso en que la CPU 1 lee la línea *A*. Lecturas subsecuentes efectuadas por esa CPU usan la entrada en la caché y no pasan por el bus. Otra CPU podría traer la misma línea y colocarla en su caché, pero como la caché que originalmente la tenía (CPU 1) espía, la CPU ve que ya no está sola y anuncia por el bus que también tiene una copia. Ambas copias se marcan como compartidas (estado S, *shared*), como se muestra en la figura 8-24(b). Así, el estado S implica que la línea está en uno o más cachés para lectura y la memoria está actualizada. Lecturas subsecuentes de una CPU de una línea que tiene en caché en el estado S no usan el bus y no hacen que el estado cambie.

Considere ahora qué sucede si la CPU 2 escribe en la línea de caché que tiene en el estado S. Esa CPU coloca una señal de invalidar en el bus para indicar a todas las demás CPU que desechen sus copias. La copia que queda en caché pasa al estado M (modificado), como se muestra en la figura 8-24(c). La línea no se escribe en la memoria. Vale la pena señalar que si una línea está en el estado E cuando se escribe, no es necesario enviar una señal por el bus para invalidar otros cachés porque se sabe que no existen otras copias.

Consideremos ahora lo que sucede si la CPU 3 lee la línea. La CPU 2, que ahora posee la línea, sabe que la copia que está en la memoria no es válida, por lo que aserta una señal en el bus para pedir a la CPU 3 que espere mientras escribe su línea de vuelta en la memoria. Cuando termina, la CPU 3 trae una copia, y la línea se marca como compartida en ambos cachés, como se muestra en la figura 8-24(d). Más adelante, la CPU 2 escribe la línea otra vez, lo que invalida la copia que está en la caché de la CPU 3, como se muestra en la figura 8-24(e).

Por último, la CPU 1 escribe una palabra en la línea. La CPU 2 ve que se ha intentado una escritura y aserta una señal de bus para pedir a la CPU 1 que espere mientras escribe su línea de vuelta en la memoria. Una vez que termina, la CPU 2 marca su propia copia como no válida, ya que sabe que otra CPU está a punto de modificarla. En este punto tenemos la situación en la que una CPU está escribiendo una línea que no está en ninguna caché. Si se está usando la política de escribir y asignar, la línea se cargará en la caché y se marcará como modificada, como se muestra en la figura 8-24(f). Si no se está usando la política de escribir y asignar, la escritura se efectuará directamente en la memoria y la línea no estará en ninguna caché.

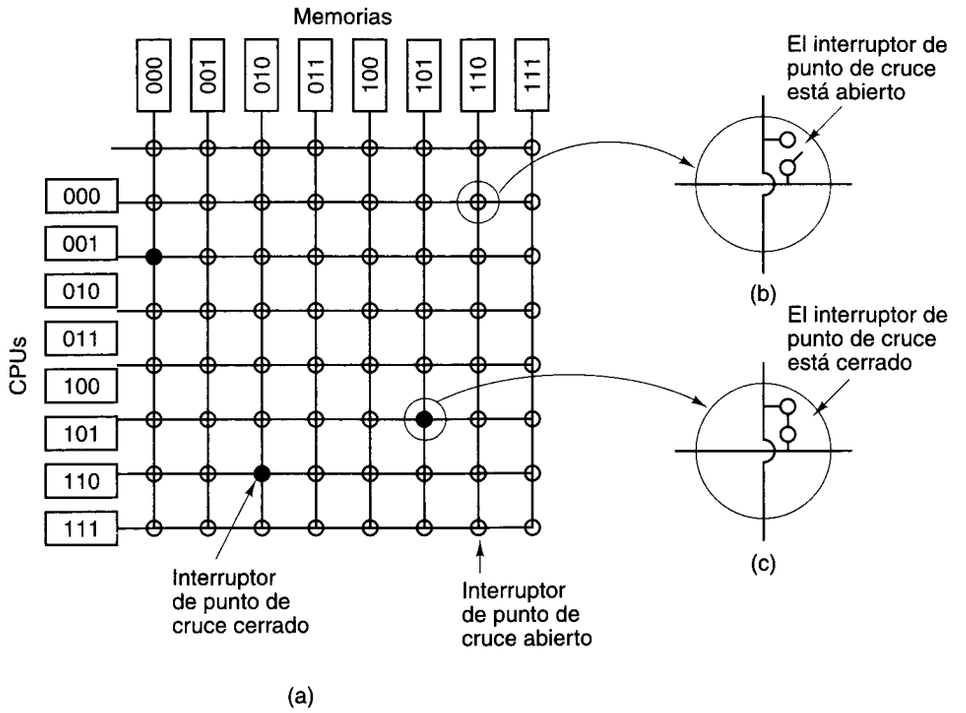


Figura 8-25. (a) Conmutador de barra cruzada de 8×8. (b) Punto de cruce abierto. (c) Punto de cruce cerrado.

8.3.3 Multiprocesadores UMA que usan conmutadores de barras cruzadas

Aun con todas las posibles optimaciones, el uso de un solo bus limita el tamaño de un multiprocesador UMA a unas 16 o 32 CPU. Si queremos rebasar ese límite, necesitamos un tipo de red de interconexión distinto. El circuito más sencillo para conectar n CPU a k memorias es el **conmutador de barras cruzadas** que se muestra en la figura 8-25. Este tipo de conmutadores se ha usado desde hace décadas en las centrales de conmutación telefónica para conectar un grupo de líneas entrantes a un conjunto de líneas salientes de forma arbitraria.

En cada intersección de una línea horizontal (entrante) y una vertical (saliente) hay un **punto de cruce**. Un punto de cruce es un conmutador pequeño que se puede abrir o cerrar eléctricamente, dependiendo de si se quiere conectar o no las líneas horizontal y vertical. En la figura 8-25(a) vemos tres puntos de cruce cerrados simultáneamente, lo que permite conexiones entre los pares (CPU, memoria) (001, 000), (101, 101) y (110, 010) al mismo tiempo. Hay muchas otras combinaciones posibles. De hecho, el número de combinaciones es igual al número de formas distintas en que cuatro torres se pueden colocar sin peligro en un tablero de ajedrez.

Una de las propiedades más útiles del conmutador de barras cruzadas es que es una **red no bloqueadora**, lo que significa que a ninguna CPU se le niega la conexión que necesita porque algún punto de cruce o línea está ocupado (suponiendo que el módulo de memoria en sí está disponible). Además, no es necesario planear con anticipación. Aunque ya se hayan establecido siete conexiones arbitrarias, siempre será posible conectar la última CPU con la última memoria. Más adelante veremos esquemas de interconexión que no poseen estas propiedades.

Una de las peores propiedades del conmutador de barras cruzadas es el hecho de que el número de puntos de cruce aumenta en proporción a n^2. Con 1000 CPU y 1000 módulos de memoria necesitamos un millón de puntos de cruce. Un conmutador de barras cruzadas de ese tamaño no es factible. No obstante, para sistemas de tamaño mediano, un diseño de barras cruzadas resulta práctico.

El Sun Enterprise 10000

Como ejemplo de multiprocesador UMA basado en un conmutador de barras cruzadas, consideremos el Sun Enterprise 10000 (Charlesworth, 1998; Charlesworth *et al.*, 1998). Este sistema consiste en un solo mueble que tiene hasta 64 CPU. El conmutador de barras cruzadas, llamado **Gigaplane-XB**, está empacado en una tarjeta de circuitos que tiene ocho ranuras de inserción en cada lado. Cada ranura puede aceptar una tarjeta de procesadores enorme (40×50 cm) que contiene cuatro CPU UltraSPARC de 333 MHz y 4 GB de RAM. Debido a los requisitos de temporización tan estrictos y la baja latencia de las barras cruzadas, accesar a memoria en otra tarjeta no es más lento que accesar a memoria en la misma tarjeta.

Tener un solo bus para todo el tráfico entre procesadores y la memoria no sería factible, por lo que en el Enterprise 10000 se usa otra estrategia. Primero, hay un conmutador de barras cruzadas 16×16 para transferir datos entre la memoria y los cachés. Las líneas de caché son de 64 bytes y el conmutador tiene 16 bytes de anchura, por lo que se requieren cuatro ciclos para transferir una línea de caché. El conmutador opera punto a punto, por lo que no puede usarse para mantener la consistencia de las cachés.

Por ello, además del conmutador de barras cruzadas, hay cuatro buses de direcciones que sirven para espiar, como se muestra en la figura 8-26. Cada bus se usa para una cuarta parte del espacio de direcciones físico, de modo que se usan dos bits de dirección para seleccionar el bus que se espiará. Cada vez que una CPU tiene un fallo de lectura y necesita leer la memoria, la dirección se coloca en el bus de dirección apropiado para ver si tal vez está en alguna caché remota. Las 16 tarjetas espían por los cuatro buses de direcciones simultáneamente, de modo que si no hay respuesta quiere decir que la línea no está en caché y se debe traer de la memoria.

La búsqueda de información en la memoria se efectúa de punto a punto por el conmutador de barras cruzadas, de 16 en 16 bytes. El ciclo de bus es de 12 ns (83.3 MHz) y cada bus de direcciones se puede espiar en un ciclo sí y otro no, para un total de 167 millones de espionajes/s. Cada espionaje podría requerir la transferencia de una línea de caché de 64 bytes, de modo que el conmutador podría tener que transferir 9.93 GB/s (recuerde que 1 GB = 1.0737×10^9 bytes/s, no 10^9 bytes/s). De hecho, se puede hacer pasar una línea de caché de 64 bytes a través del conmutador en cuatro ciclos de bus (48 ns), lo que da un ancho de banda de 1.24 GB/s por transferencia. Puesto que el conmutador puede manejar 16 transferencias simultáneas, su ancho de banda instantáneo máximo es de 19.87 GB/s, que es suficiente para mantenerse a la par de la tasa de espionaje, aun tomando en cuenta la contención por el conmutador, que reduce el ancho de banda práctico a cerca del 60% del valor teórico.

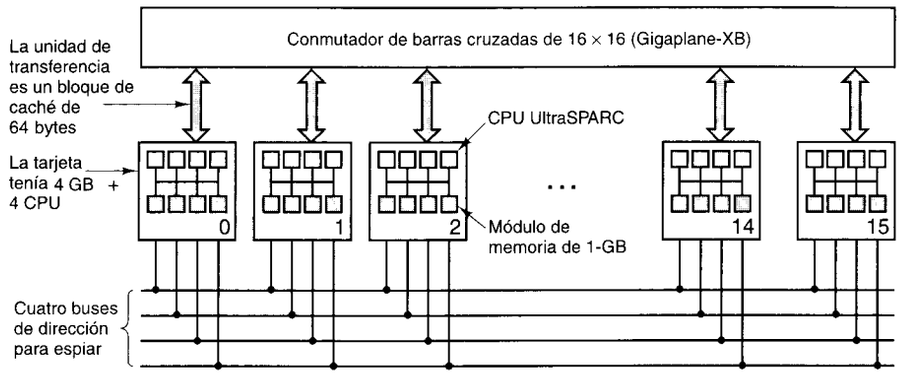


Figura 8-26. Multiprocesador simétrico Sun Enterprise 10000.

Es evidente que el Enterprise 10000 está en el límite; usa cuatro buses para espiar en paralelo y un conmutador de barras cruzadas muy ancho para transferir datos. Si queremos ir mucho más allá de 64 CPU, vamos a necesitar algo muy distinto.

8.3.4 Multiprocesadores UMA que usan redes de conmutación multietapas

Ese "algo muy distinto" puede basarse en el humilde conmutador de 2×2 que se muestra en la figura 8-27(a). Este conmutador tiene dos entradas y dos salidas. Los mensajes que llegan por cualquiera de las líneas de entrada se pueden conmutar a cualquiera de las líneas de salida. Para nuestros fines, los mensajes contendrán hasta cuatro partes, como se muestra en la figura 8-27(b). El campo *Módulo* indica cuál memoria debe usarse. *Dirección* especifica una dirección dentro de un módulo. *Cód-op* da la operación, digamos READ o WRITE. Por último, el campo *Valor* opcional podría contener un operando, digamos una palabra de 32 bits que se escribirá con la instrucción WRITE. El conmutador inspecciona el campo *Módulo* y lo usa para determinar si el mensaje se debe enviar a *X* o a *Y*.

Nuestros conmutadores 2×2 se pueden acomodar de muchas maneras para construir **redes de conmutación multietapas** más grandes (Adams *et al.*, 1987; Bhuyan *et al.*, 1989;

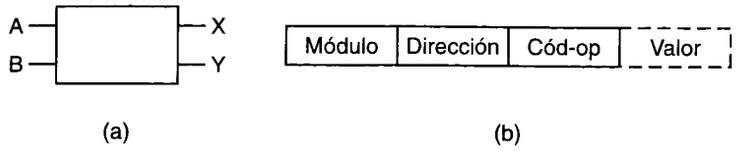


Figura 8-27. (a) Conmutador 2 × 2. (b) Un formato de mensaje.

Kumar y Reddy, 1987). Una posibilidad es la austera **red omega** que se ilustra en la figura 8-28. Aquí hemos conectado ocho CPU con ocho memorias empleando 12 conmutadores. De forma más general, para n CPU y n memorias necesitaríamos $\log_2 n$ etapas, con $n/2$ conmutadores por etapa, para un total de $(n/2)\log_2 n$ conmutadores, que es mucho mejor que n^2 puntos de cruce, sobre todo para valores grandes de n.

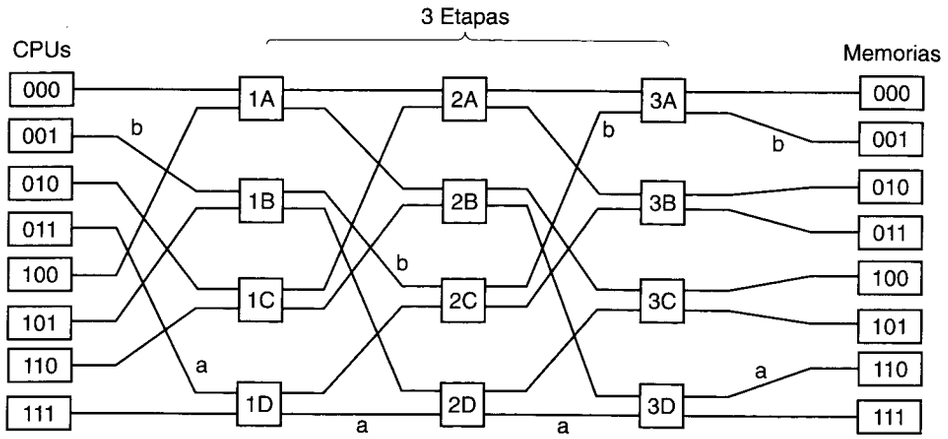


Figura 8-28. Red de conmutación omega.

El patrón de cableado para la red omega se conoce como **barajado perfecto**, ya que el mezclado de las señales en cada etapa se parece a un mazo de naipes que se divide a la mitad y luego se mezcla naipe por naipe. Para ver cómo funciona la red omega, suponga que la CPU 011 quiere leer una palabra del módulo de memoria 110. La CPU envía un mensaje READ al conmutador 1D que contiene 110 en el campo *Módulo*. El conmutador toma el primer bit (es decir, el de la extrema izquierda) de 110 y lo usa para enrutamiento. Un 0 dirige a la salida superior y un 1 dirige a la salida inferior. Puesto que este bit es 1, el mensaje se encamina por la salida inferior hacia 2D.

Todos los conmutadores de segunda etapa, incluido 2D, usan el segundo bit para enrutamiento. Éste también es 1, de modo que el mensaje ahora se reenvía por la salida inferior hacia 3D. Aquí se prueba el tercer bit y se ve que es 0. Por tanto, el mensaje sale por la salida superior y llega a la memoria 110, como se deseaba. El camino que este mensaje sigue se ha marcado con la letra *a* en la figura 8-28.

A medida que el mensaje avanza por la red de conmutación, los bits del extremo izquierdo del número de módulo dejan de necesitarse. Se pueden aprovechar para registrar el número de línea entrante, a fin de que la respuesta pueda encontrar el camino de regreso. En el caso del camino *a*, las líneas entrantes son 0 (entrada superior de 1D), 1 (entrada inferior de 2D) y 1 (entrada inferior de 3D), respectivamente. La respuesta se encamina de regreso usando 011, sólo que esta vez se lee de derecha a izquierda.

Al mismo tiempo que está pasando todo esto, la CPU 001 quiere escribir una palabra en el módulo de memoria 001. Aquí ocurre un proceso análogo, y el mensaje se encamina por las salidas superior, superior e inferior, respectivamente, camino que se ha marcado con la letra *b*. Cuando el mensaje llega, su campo *Módulo* es 001, lo que representa el camino que tomó. Puesto que estas dos solicitudes no usan los mismos conmutadores, líneas ni módulos de memoria, pueden ocurrir en paralelo.

Considere ahora qué sucedería si simultáneamente la CPU 000 quisiera acceder al módulo de memoria 000. Su solicitud entraría en conflicto con la solicitud de la CPU 001 en el conmutador 3A. Una de ellas tendría que esperar. A diferencia del conmutador de barras cruzadas, la red omega es una **red bloqueadora**. No todos los conjuntos de solicitudes se pueden procesar simultáneamente. Puede haber conflictos por el uso de un alambre o por el uso de un conmutador, y también entre solicitudes *a* la memoria y respuestas *de* la memoria.

Obviamente es deseable repartir las referencias a la memoria de manera uniforme entre los módulos. Una técnica común consiste en usar los bits de orden bajo como número de módulo. Considere, por ejemplo, un espacio de direcciones orientado a bytes para una computadora que por lo regular accede a palabras de 32 bits. Los dos bits de orden bajo normalmente son 00, pero los siguientes 3 bits tienen una distribución uniforme. Si se usan estos 3 bits como número de módulo, las palabras con direcciones consecutivas estarán en módulos consecutivos. Un sistema de memoria en el que palabras consecutivas están en diferentes módulos se describe como **intercalado**. Las memorias intercaladas maximizan el paralelismo porque casi todas las referencias a la memoria son a palabras consecutivas. También es posible diseñar redes de conmutación que sean no bloqueadoras y que ofrezcan varios caminos de cada CPU a cada módulo de memoria, a fin de distribuir el tráfico mejor.

8.3.5 Multiprocesadores NUMA

A estas alturas debe ser evidente que los multiprocesadores UMA de un solo bus generalmente están limitados a unas cuantas docenas de CPU, y los multiprocesadores de barras cruzadas o conmutados necesitan mucho hardware (costoso) y no son mucho más grandes. Si se quiere llegar a más de 100 CPU, algo tiene que ceder. Por lo regular, lo que cede es la idea de que todos los módulos de memoria tienen el mismo tiempo de acceso. Esta concesión da lugar a la idea de **multiprocesadores con acceso no uniforme a la memoria** (NUMA, *NonUniform Memory Access*). Al igual que sus parientes UMA, éstos ofrecen un solo espacio de direcciones para todas las CPU, pero a diferencia de las máquinas UMA, el acceso a los módulos de memoria local es más rápido que a los módulos remotos. Así pues, todos los programas UMA se pueden ejecutar sin cambios en las máquinas NUMA, pero el desempeño es más bajo que en una máquina UMA si la velocidad de reloj es la misma.

Las máquinas NUMA tienen tres características clave comunes a todas ellas que juntas las distinguen de otros multiprocesadores:

1. Hay un solo espacio de direcciones visible para todas las CPU.
2. El acceso a la memoria remota se efectúa con instrucciones LOAD y STORE.
3. El acceso a la memoria remota es más lento que el acceso a la memoria local.

Cuando el tiempo de acceso a la memoria remota no se oculta (porque no hay uso de cachés), el sistema se llama **NC-NUMA**. Si hay cachés coherentes, el sistema se llama **CC-NUMA** (al menos la gente de hardware lo llama así). La gente de software a menudo lo llama **DSM en hardware** porque es básicamente igual a la memoria compartida distribuida (DSM) pero implementado por el hardware utilizando páginas pequeñas.

Una de las primeras máquinas NC-NUMA (aunque todavía no se acuñaba ese nombre) fue la Cm* de Carnegie-Mellon, que se ilustra en forma simplificada en la figura 8-29 (Swan *et al.*, 1977). La Cm* consistía en una colección de CPU LSI-11, cada una con un poco de memoria que se direccionaba por un bus local. (El LSI-11 fue una versión en un solo chip del DEC PDP-11, una minicomputadora popular en los años setenta.) Además, los sistemas LSI-11 estaban conectados por un bus del sistema. Cuando una solicitud de memoria llegaba a la MMU (que tenía modificaciones especiales), se verificaba si la palabra requerida estaba en la memoria local. Si así era, se enviaba una solicitud por el bus local para obtener la palabra; si no, la solicitud se enviaba por el bus del sistema al sistema que contenía la palabra, el cual entonces respondía. Desde luego, lo segundo tardaba mucho más que lo primero. Si bien un programa podía ejecutarse sin problema usando memoria remota, tardaba 10 veces más en ejecutarse que el mismo programa usando memoria local.

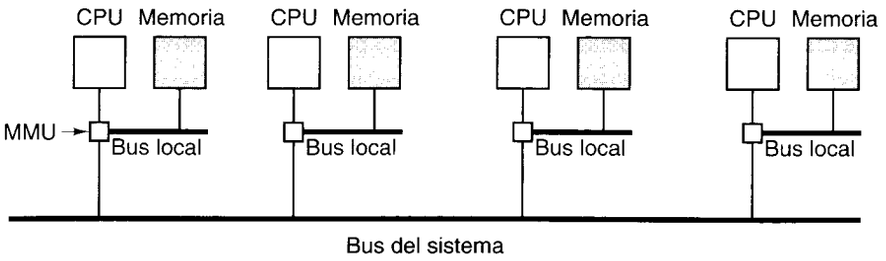


Figura 8-29. Máquina NUMA basada en dos niveles de buses. El Cm* fue el primer multiprocesador que usó este diseño.

La coherencia de memoria está garantizada en una máquina NC-NUMA porque no se usan cachés. Cada palabra de memoria reside en una y sólo una localidad, por lo que no hay peligro de que una copia tenga datos obsoletos: no hay copias. Desde luego, ahora es muy importante cuál página está en cuál memoria porque el menoscabo en el desempeño debido a que esté en el lugar equivocado es muy alto. Por ello, las máquinas NC-NUMA utilizan software complejo para colocar las páginas de modo que se maximice el desempeño.

Por lo regular existe un proceso demonio llamado **explorador de páginas** que se ejecuta cada cierto número de segundos. Su trabajo consiste en examinar las estadísticas de utilización y cambiar de lugar las páginas en un intento por mejorar el desempeño. Si una página parece estar en el lugar equivocado, el explorador de páginas anula su correspondencia con la memoria de modo que la siguiente referencia a ella cause un fallo de página. Cuando ocurra el fallo, se tomará una decisión acerca de dónde debe colocarse la página, posiblemente en una memoria distinta de aquella en la que estaba antes. Para evitar la hiperpaginación, generalmente hay una regla que dice que, una vez que una página se coloca, no podrá cambiarse de lugar durante un tiempo ΔT. Se han estudiado diversos algoritmos, pero la conclusión es de que ningún algoritmo tiene un desempeño óptimo en todas las circunstancias (LaRowe y Ellis, 1991).

8.3.6 Multiprocesadores NUMA con coherencia de caché

Los diseños de multiprocesador como el de la figura 8-29 no se prestan a cambios de escala porque no usan cachés. Tener que acudir a la memoria remota cada vez que se accesa a una palabra de memoria no local es un freno importante para el desempeño. Sin embargo, si se añaden cachés, también hay que añadir coherencia de cachés. Una forma de proporcionar coherencia de cachés es espiar el bus del sistema. Técnicamente, no es difícil hacerlo, pero como vimos con el Enterprise 10000, incluso con cuatro buses para espiar y un conmutador de barras cruzadas de alta velocidad con 16 bytes de anchura para el tráfico de datos, 64 CPU es casi el límite de tamaño. Si queremos construir multiprocesadores en verdad grandes, se requiere un enfoque fundamentalmente distinto.

El enfoque más popular para construir multiprocesadores **NUMA con coherencia de caché** (CC-NUMA, *Cache Coherent NUMA*) es el **multiprocesador basado en directorios**. La idea es mantener una base de datos que dice dónde está cada línea de caché y cuál es su situación. Cuando se hace referencia a una línea de caché, se consulta la base de datos para averiguar dónde está y si está limpia o sucia (modificada). Puesto que esta base de datos se debe consultar en cada instrucción que hace referencia a la memoria, se tiene que mantener en hardware especial extremadamente rápido capaz de responder en una fracción de ciclo de bus.

A fin de hacer un poco más concreta la idea de un multiprocesador basado en directorios, consideremos un ejemplo (hipotético) sencillo, un sistema con 256 nodos, cada uno de los cuales consiste en una CPU y 16 MB de RAM conectada a la CPU por un bus local. La memoria total es de 2^{32} bytes, dividida en 2^{26} líneas de caché de 64 bytes cada una. La memoria se reparte estáticamente entre los nodos, con 0-16M en el nodo 0, 16M-32M en el nodo 1, y así. Los nodos se conectan con una red de interconexión, como se muestra en la figura 8-30(a). La red podría ser una cuadrícula, un hipercubo u otra topología. Cada nodo contiene también las entradas de directorio para las 2^{18} líneas de caché de 64 bytes que constituyen su memoria de 2^{24} bytes. Por el momento, supondremos que una línea puede estar contenida en cuando más una caché.

Para ver cómo funciona el directorio, sigamos la pista a una instrucción LOAD emitida por la CPU 20 que hace referencia a una línea que está en caché. Primero la CPU que emite la instrucción la presenta a su MMU, la cual la traduce en una dirección física, digamos 0x24000108. La MMU divide esta dirección en las tres partes que se muestran en la figura

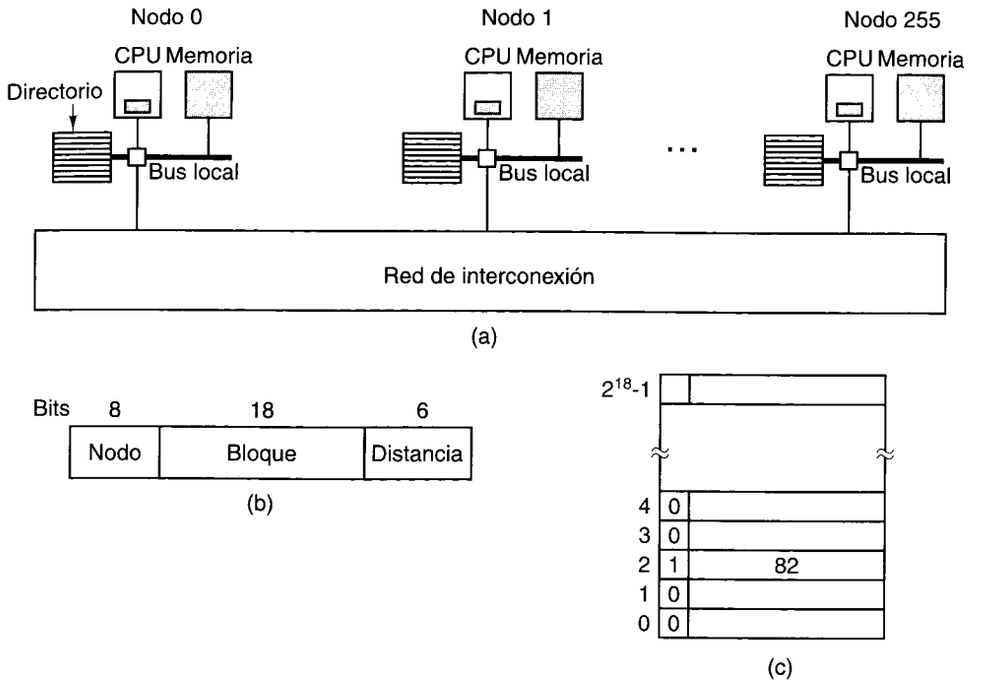


Figura 8-30. (a) Multiprocesador de 256 nodos basado en directorios. (b) División de una dirección de memoria de 32 bits en campos. (c) El directorio del nodo 36.

8-30(b). En decimal, las tres partes son el nodo 36, línea 4 y distancia 8. La MMU ve que la palabra de memoria a la que se hizo referencia es del nodo 36, no del nodo 20, por lo que envía un mensaje de solicitud por la red de interconexión al nodo base de la línea, el 36, preguntándole si su línea 4 está en caché y, si así es, dónde está.

Cuando la solicitud llega al nodo 36 por la red de interconexión, se le encamina al hardware de directorio. El hardware busca la entrada 4 de su tabla de 2^{18} entradas, una para cada una de sus líneas de caché, y la extrae. En la figura 8-30(c) vemos que la línea no está en la caché, por lo que el hardware trae la línea 4 de la RAM local, la devuelve al nodo 20 y actualiza la entrada 4 de su directorio de modo que indique que la línea ahora está en caché en el nodo 20.

Consideremos ahora una segunda solicitud, que ahora pide la línea 2 del nodo 36. En la figura 8-30(c) vemos que esta línea está en caché en el nodo 82. En este punto el hardware podría actualizar la entrada 2 del directorio de modo que diga que la línea ahora está en el nodo 20 y luego enviar al nodo 82 un mensaje pidiéndole que pase la línea al nodo 20 e invalide su caché. Observe que incluso en un supuesto "multiprocesador de memoria compartida" hay una gran cantidad de transferencia de mensajes oculta.

Como digresión rápida, calculemos qué tanta memoria están ocupando los directorios. Cada nodo tiene 16 MB de RAM y 2^{18} entradas de 9 bits para mantenerse al tanto de esa RAM. Así, el gasto extra del directorio es de 9×2^{18} bits dividido entre 16 MB, o sea cerca del

1.76%, que generalmente es aceptable (aunque tiene que ser memoria de alta velocidad, lo que aumenta su costo). Aun con líneas de caché de 32 bytes el gasto extra sería de sólo un 4%. Con líneas de caché de 128 bytes, el gasto extra sería de menos del 1%.

Una limitación obvia de este diseño es que una línea se puede colocar en caché sólo en un nodo. Para poder colocar líneas en caché en varios nodos necesitaríamos alguna forma de localizarlas a todas, por ejemplo para invalidarlas o actualizarlas cuando se efectúa una escritura. Hay varias opciones que permiten colocar en caché en varios nodos al mismo tiempo.

Una posibilidad es dar a cada entrada de directorio k campos para especificar otros nodos, lo que permite a cada línea colocarse en caché en hasta k nodos. Una segunda posibilidad es sustituir el número de nodo de nuestro sencillo diseño por un mapa de bits, con un bit por nodo. Con esta opción no hay límite para el número de posibles copias, pero el gasto extra aumenta sustancialmente. Tener un directorio con 256 bits por cada línea de caché de 64 bytes (512 bits) implica un gasto extra de más del 50%. Una tercera posibilidad es mantener un campo de 8 bits en cada entrada de directorio y usarlo como cabeza de una lista enlazada que "ensarta" todas las copias de la línea de caché. Esta estrategia requiere almacenamiento extra en cada nodo para los apuntadores de la lista enlazada, y también requiere seguir una lista enlazada para encontrar todas las copias cuando esto es necesario. Cada posibilidad tiene sus ventajas y desventajas, y las tres se han usado en sistemas reales.

Otra mejora del diseño de directorios consiste en asentar si la línea de caché está limpia (la memoria base está actualizada) o sucia (la memoria base no está actualizada). Si llega una solicitud de lectura para una línea de caché limpia, el nodo base puede satisfacer la solicitud desde la memoria, sin tener que reenviarla a una caché. En cambio, una solicitud de lectura de una línea de caché sucia debe reenviarse al nodo que contiene la línea de caché porque sólo ella tiene una copia válida. Si sólo se permite una copia en caché, como en la figura 8-30, no resulta realmente ventajoso mantenerse al tanto de si está limpia o sucia, porque cualquier solicitud nueva requerirá el envío de un mensaje a la copia existente para invalidarla.

Desde luego, mantenerse al tanto de si cada línea de caché está limpia o sucia implica que cuando una línea de caché se modifica hay que informar de ello al nodo base, aunque sólo exista una copia en caché. Si hay varias copias, la modificación de una de ellas requiere invalidar todas las demás, por lo que se necesita algún protocolo para evitar condiciones de competencia. Por ejemplo, para modificar una línea de caché compartida, uno de los que la tienen podría tener que solicitar acceso exclusivo *antes* de modificarla. Semejante solicitud haría que todas las demás copias se invaliden antes de poder dar el permiso. En (Stenstrom *et al.*, 1997) se describen optimaciones del desempeño adicionales para máquinas CC-NUMA.

El multiprocesador Stanford DASH

El primer multiprocesador CC-NUMA basado en directorios, **arquitectura de directorios para memoria compartida** (DASH, *Directory Architecture for SHared memory*), se construyó en la Stanford University (Lenoski *et al.*, 1992) como proyecto de investigación. Su diseño es fácil de explicar y ha influido mucho en varios productos comerciales, como el SGI Origin 2000, de

modo que vale la pena echarle un vistazo. Nos concentraremos en el prototipo de 64 CPU que realmente se construyó, pero es fácil cambiar la escala del diseño a máquinas más grandes.

En la figura 8-31(a) se muestra un diagrama ligeramente simplificado del prototipo DASH. Consiste en 16 cúmulos, cada uno de los cuales contiene un bus, cuatro CPU MIPS R3000, 16 MB de la memoria global, y algo de equipo de E/S (discos, etc.), que no se muestran. Cada CPU espía en su bus local, pero no en ningún otro bus. La coherencia local se mantiene espiando; la coherencia global requiere un mecanismo distinto porque no hay espionaje global.

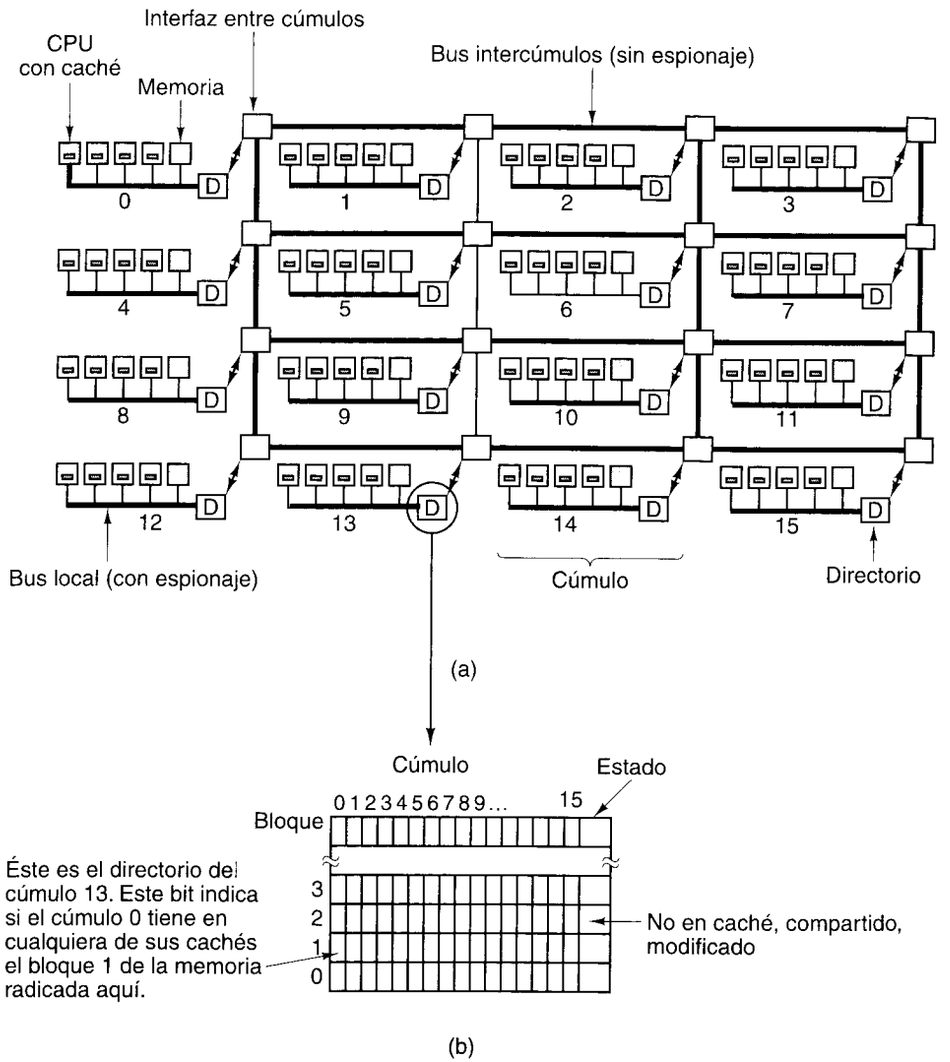


Figura 8-31. (a) La arquitectura DASH. (b) Un directorio DASH.

El espacio de direcciones total disponible en el prototipo es de 256 MB, dividido en 16 regiones de 16 MB cada una. La memoria global del cúmulo 0 contiene las direcciones 0 a 16M. La memoria global del cúmulo 1 contiene las direcciones 16M a 32M, etc. La memoria se coloca en caché y se transfiere en unidades de 16 bytes (líneas), de modo que cada cúmulo tiene 1M líneas de memoria dentro de su espacio de direcciones.

Cada cúmulo tiene un directorio que se mantiene al tanto de cuáles cúmulos tienen actualmente copias de sus líneas. Puesto que cada cúmulo posee 1M líneas de memoria, tiene 1M entradas en su directorio, una por línea. Cada entrada contiene un mapa de bits con un bit por cúmulo que indica si ese cúmulo tiene o no actualmente en caché la línea. La entrada también tiene un campo de dos bits que indica el estado de la línea.

Tener 1M entradas de 18 bits cada una implica que el tamaño total de cada directorio es de más de 2 MB. Con 16 cúmulos, la memoria de directorio total es de poco más de 36 MB, o sea cerca del 14% de los 256 MB. Si se incrementa el número de CPU por cúmulo, la cantidad de memoria de directorio no cambia. Así, tener más CPU por cúmulo permite amortizar el costo de la memoria de directorio, y también del controlador de bus, entre un número mayor de CPU, lo que reduce el costo por CPU. Es por esto que cada cúmulo tiene varias CPU.

Cada cúmulo de DASH está conectado a una interfaz que permite al cúmulo comunicarse con otros. Las interfaces están conectadas por enlaces intercúmulo (buses primitivos) en una cuadrícula rectangular, como se muestra en la figura 8-31(a). A medida que se añaden más cúmulos al sistema, también se añaden más enlaces entre cúmulos, por lo que el ancho de banda aumenta y el sistema es escalable. El sistema de enlaces intercúmulos usa enrutamiento por túnel, de modo que la primera parte de un paquete se reenvía antes de recibirse todo el paquete, con lo que se reduce el retraso en cada brinco. Aunque no se muestra en la figura, en realidad hay dos conjuntos de enlaces intercúmulos, uno para paquetes de solicitud y uno para paquetes de respuesta. Los enlaces intercúmulos no pueden espiarse.

Cada línea de caché puede estar en uno de los tres estados siguientes:

1. NO EN CACHÉ – La única copia de la línea está en esta memoria.
2. COMPARTIDA – La memoria está actualizada; la línea puede estar en varias cachés.
3. MODIFICADA – La memoria es incorrecta; sólo una caché contiene la línea.

El estado de cada línea de caché se guarda en el campo *Estado* de su entrada de directorio, como se muestra en la figura 8-31(b).

Los protocolos DASH se basan en propiedad e invalidación. En cada instante, cada línea de caché tiene un dueño único. En el caso de líneas NO EN CACHÉ o COMPARTIDAS, el cúmulo base de la línea es el dueño. En el caso de líneas MODIFICADAS, el cúmulo que contiene la única copia es el dueño. Para escribir en una línea COMPARTIDA primero es necesario encontrar e invalidar todas las copias existentes. Es aquí donde entran los directorios.

Para ver cómo funciona este mecanismo, consideremos primero la forma en que una CPU lee una palabra de memoria. Primero examina su propio caché. Si la caché no tiene la palabra, se emite una solicitud por el bus de cúmulo local para ver si otra CPU del cúmulo

tiene la línea que la contiene. Si así es, se ejecuta una transferencia de caché a caché para colocar la línea en la caché de la CPU solicitante. Si la línea es COMPARTIDA, se hace una copia; si es MODIFICADA, se informa al directorio base que la línea ahora es COMPARTIDA. De cualquier manera, un acierto de uno de los cachés satisface la instrucción pero no afecta el mapa de bits de ningún directorio (porque el directorio tiene un bit por cúmulo, no un bit por CPU).

Si la línea no está presente en ninguno de los cachés del cúmulo, se envía un paquete de solicitud al cúmulo base de la línea, el cual puede determinarse examinando los cuatro bits altos de la dirección de memoria. El cúmulo base bien podría ser el cúmulo del solicitante, en cuyo caso el mensaje no se envía físicamente. El hardware de gestión de directorio del cúmulo base examina sus tablas para ver en qué estado está la línea. Si está NO EN CACHÉ o COMPARTIDA, el hardware trae la línea de su memoria global y la devuelve al cúmulo solicitante; luego actualiza su directorio para indicar que la línea está en caché en el cúmulo del solicitante.

En cambio, si la línea requerida está MODIFICADA, el hardware de directorio busca la identidad del cúmulo que tiene la línea y reenvía ahí la solicitud. El cúmulo que tiene la línea la envía entonces al cúmulo solicitante y marca su propia copia como COMPARTIDA porque ahora hay varias copias; también envía una copia de vuelta al cúmulo base para que esa memoria pueda actualizarse y el estado de la línea se cambie a COMPARTIDA.

Las escrituras funcionan de otra manera. Antes de poder efectuar una escritura, la CPU que quiere escribir debe estar segura de que es la dueña de la única copia de la línea de caché en el sistema. Si ya tiene la línea en su caché y la línea está MODIFICADA, la escritura puede efectuarse de inmediato. Si tiene la línea pero está COMPARTIDA, primero se envía un paquete al cúmulo base para solicitar que se busquen todas las demás copias y se invaliden.

Si la CPU solicitante no tiene la línea de caché, emite una solicitud por el bus local para ver si cualquiera de sus vecinas la tiene. Si así es, se efectúa una transferencia de caché a caché (o de memoria a caché). Si la línea es COMPARTIDA, el cúmulo base deberá invalidar todas las demás copias, si existen.

Si la difusión local no logra encontrar una copia y la línea tiene su base en otro lado, se envía un paquete al cúmulo base. Aquí podemos distinguir tres casos. Si la línea está NO EN CACHÉ, se marca como MODIFICADA y se envía al solicitante. Si la línea está COMPARTIDA, todas las copias se invalidan y luego se sigue el procedimiento para NO EN CACHÉ. Si la línea está MODIFICADA, la solicitud se remite al cúmulo remoto que actualmente es dueño de la línea (si es necesario). Este cúmulo satisface la solicitud y luego invalida su propia copia.

Mantener la consistencia de la memoria en DASH es relativamente complejo y lento. Un solo acceso a la memoria podría requerir el envío de un número considerable de paquetes. Además, a fin de mantener la consistencia de la memoria, el acceso por lo regular no puede llevarse a cabo antes de que se haya reconocido la recepción de todos los paquetes, lo que puede tener un efecto notable sobre el desempeño. Para superar estos problemas, DASH utiliza diversas técnicas especiales, como dos conjuntos de enlaces intercúmulos, escrituras por filas de procesamiento y el uso de consistencia de liberación en lugar de consistencia secuencial como modelo de memoria.

El multiprocesador Sequent NUMA-Q

Aunque la máquina DASH tuvo una influencia notable, nunca fue un producto comercial. En vista de la importancia que están adquiriendo los multiprocesadores CC-NUMA, vale la pena examinar también un producto comercial. Como ejemplo usaremos la máquina Sequent NUMA-Q 2000 porque utiliza un protocolo de coherencia de cachés interesante e importante llamado **interfaz coherente escalable** (SCI, *Scalable Coherent Interface*). Este protocolo se estandarizó como IEEE 1596 y se usa también en varias otras máquinas CC-NUMA.

El NUMA-Q se basa en la tarjeta **quad estándar** vendida por Intel que contiene cuatro chips de CPU Pentium Pro y hasta 4 GB de RAM. Cada CPU tiene una caché de nivel 1 y una caché de nivel 2. Todas estas cachés se mantienen coherentes espiando en el bus local de 534 MB/s de la tarjeta quad con el protocolo MESI. Las líneas de caché son de 64 bytes. El NUMA-Q se ilustra en la figura 8-32.

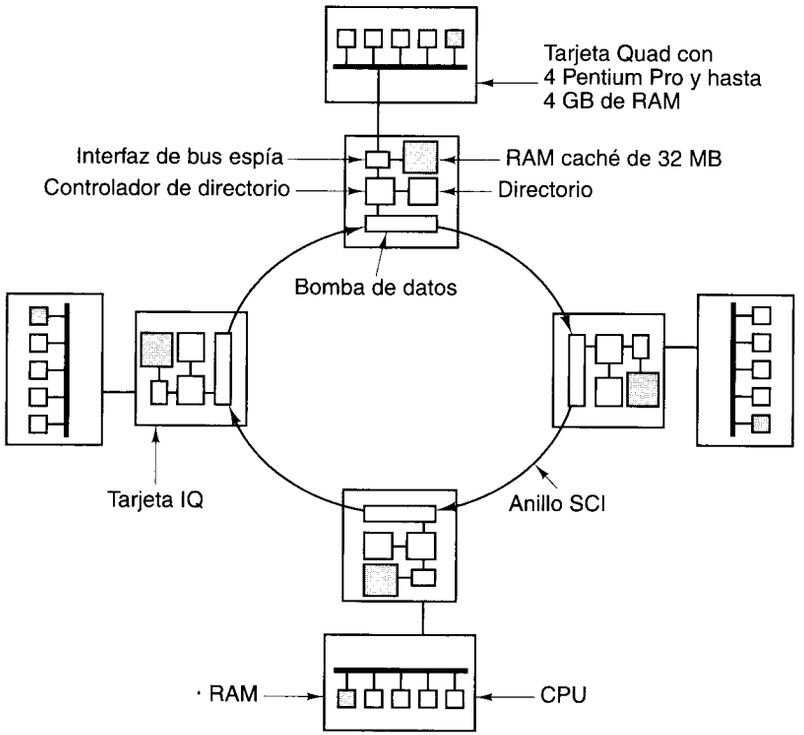


Figura 8-32. El multiprocesador NUMA-Q.

El tamaño del sistema se expande más allá de cuatro CPU insertando una tarjeta de controlador de red en una ranura de la tarjeta quad diseñada para controladores de red. Dicha tarjeta, llamada **tarjeta IQ-Link**, es el adhesivo que convierte todas las tarjetas quad en un solo multiprocesador. Su labor primaria es implementar el protocolo SCI. Cada tarjeta IQ-Link

contiene 32 MB de caché, un directorio que se mantiene al tanto de lo que está en la caché, una interfaz de espionaje con el bus local de la tarjeta quad, y un chip a la medida llamado **bomba de datos** que conecta la tarjeta IQ-Link con otras tarjetas IQ-Link. Básicamente, el chip "bombea" datos del lado de la entrada al lado de la salida, quedándose con los datos que van dirigidos a su nodo y dejando pasar los demás datos sin modificación.

Juntas, todas las tarjetas IQ-Link forman un anillo, como se muestra en la figura 8-32. El protocolo SCI mantiene la coherencia de todas las cachés de tarjeta IQ-Link usando el anillo; no tiene conocimiento del protocolo MESI que se usa para mantener la coherencia entre las cuatro CPU y la caché de 32 MB en cada nodo, así que este diseño tiene dos niveles de protocolo de coherencia, igual que DASH.

Sequent optó por usar SCI como interconexión de las tarjetas quad porque SCI se diseñó para sustituir al bus en sistemas multiprocesador y multicomputadora grandes (como el NUMA-Q). SCI apoya la coherencia de caché que se requiere en los multiprocesadores, pero también la transferencia rápida de bloques que se necesita en las multicomputadoras. SCI puede manejar hasta 64K nodos, cada uno con un espacio de direcciones de hasta 2^{48} bytes. El sistema NUMA-Q más grande consiste en 63 tarjetas quad, que contienen 252 CPU y casi 2^{38} bytes de memoria física, muy por debajo de los límites de SCI.

El anillo que conecta las tarjetas IQ-Link cumple con el protocolo SCI. En realidad, no se trata realmente de un anillo, sino de cables individuales punto a punto. Su anchura es de 18 bits que incluyen un bit de reloj, un bit indicador y 16 bits de datos, todos enviados en paralelo. Los enlaces operan a 500 MHz, lo que da una tasa de datos de 1 GB/s. El tráfico por los enlaces consiste en paquetes, cada uno de los cuales tiene una cabecera de 14 bytes, 0, 16, 64 o 256 bytes de datos, y una suma de verificación de dos bytes. El tráfico de paquetes consiste en solicitudes y respuestas.

La memoria física del NUMA-Q 2000 se divide entre los nodos, de modo que cada página de memoria tiene una máquina base. Cada tarjeta quad puede contener hasta 4 GB de RAM y las líneas de caché son de 64 bytes, de modo que cada tarjeta quad guarda 2^{26} líneas de caché. Cuando una línea de caché no se está usando, se encuentra en sólo un lugar, la memoria base.

Sin embargo, las líneas de caché a menudo se encuentran en una o más cachés remotas, por lo que cada nodo necesita una **tabla de memoria local** de 2^{26} entradas para poder localizar todas sus líneas de caché, dondequiera que se encuentren. Una posibilidad habría sido tener un mapa de bits por cada entrada, que indique cuáles tarjetas IQ-Link tienen la línea. DASH opera de este modo. Sin embargo, SCI evita este mapa de bits porque no funciona bien cuando cambia la escala. (Recuerde que SCI puede manejar 64K nodos; tener 2^{26} entradas de directorio, cada una de 64K bits, sería demasiado costoso.)

En lugar de usar un mapa de bits, todas las copias de una línea de caché se "ensartan" en una lista doblemente enlazada. La entrada de la tabla de memoria local del nodo base indica cuál nodo tiene la cabeza de la lista. En la NUMA-Q 2000 un número de 6 bits es suficiente, porque hay cuando más 63 nodos. Para el sistema SCI máximo, un número de 16 bits es lo bastante grande. De cualquier modo, este esquema funciona mucho mejor en sistemas grandes que un mapa de bits al estilo DASH con 63 o 64K bits. Esta propiedad es lo que hace que SCI sea mucho más escalable que un diseño estilo DASH.

Además de la tabla de memoria local, cada tarjeta IQ-Link tiene un directorio con una entrada por cada línea de caché que tiene actualmente. Puesto que la caché tiene una capacidad de 32 MB y las líneas de caché son de 64 bytes, cada tarjeta IQ-Link puede contener hasta 2^{19} líneas de caché. Así pues, cada directorio tiene 2^{19} entradas, una para cada línea de caché.

Si una línea está en una sola caché, la tabla de memoria local del nodo base apunta al nodo en el que se encuentra la línea. Si después la línea se coloca en caché en un segundo nodo, el protocolo SCI hará que el directorio base apunte a la nueva entrada, que a su vez apunta a la entrada vieja. De este modo se forma una lista de dos elementos. A medida que nuevos nodos comparten la misma línea, se agregan, por turno, a la cabeza de la lista. Todos los nodos que contienen la línea de caché se enlazan así en una cadena arbitrariamente larga. Este encadenamiento se ilustra en la figura 8-33 para el caso de una línea de caché que se encuentra en los nodos 4, 9 y 22.

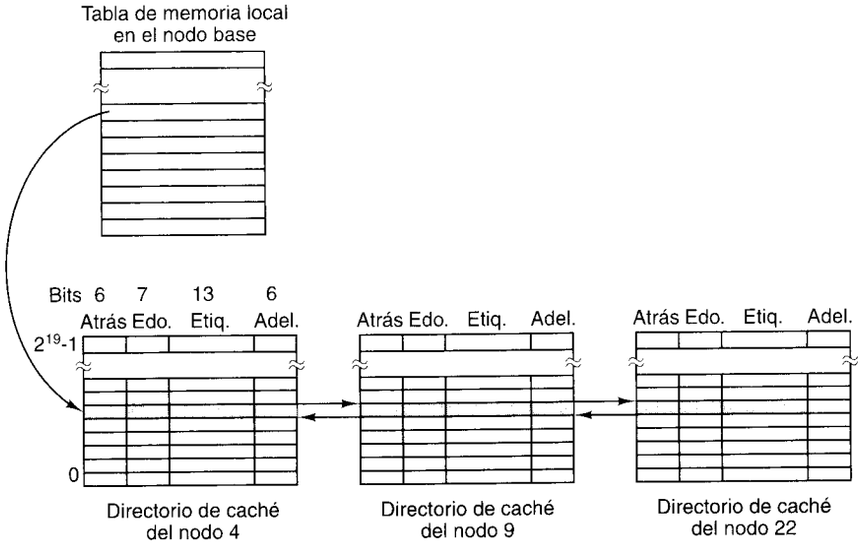


Figura 8-33. SCI encadena todos los contenedores de una línea de caché dada formando una lista doblemente enlazada. En este ejemplo se muestra una línea en caché en tres nodos.

Cada entrada de directorio consta de 36 bits. Seis de ellos apuntan al nodo que tiene la línea anterior de la cadena. Así mismo, seis bits apuntan al nodo que tiene la siguiente línea de la cadena. Un cero indica el fin de la cadena, y es por ello que el sistema máximo es de 63 nodos, no de 64. Se necesitan siete bits para registrar el estado de la línea.

Por último, se requieren 13 bits para el campo de etiqueta que identifica la línea. Recuerde que el sistema NUMA-Q 2000 más grande contiene $63 \times 2^{32} \approx 2^{38}$ bytes de RAM, por lo que hay casi 2^{32} líneas de caché. Los cachés son de correspondencia directa, así que con 2^{32} líneas de caché que corresponden a 2^{19} entradas de caché hay 2^{13} líneas que corresponden a cada entrada. Por tanto, se necesita una etiqueta de 13 bits para saber cuál es la que está ahí.

Toda línea de caché tiene una localidad fija en una y sólo una memoria que es su base. Estas líneas pueden estar en uno de tres estados: NO EN CACHÉ, COMPARTIDA y MODIFICADA. La terminología SCI para estos estados es HOME, FRESH y GONE, respectivamente, pero a fin de evitar complicaciones innecesarias seguiremos con la terminología anterior, pues ya hemos estado usándola. El estado NO EN CACHÉ implica que la línea no está en la caché de ninguna tarjeta IQ-Link, aunque podría estar en una caché local de la misma tarjeta quad. El estado COMPARTIDA implica que la línea está en al menos una caché de tarjeta IQ-Link, posiblemente en más, y que la memoria está actualizada. Por último, el estado MODIFICADA implica que la línea está en caché en alguna tarjeta IQ-Link y podría haberse modificado allí con lo que la memoria no estaría al día.

El la tarjeta IQ-Link los bloques pueden estar en cualquiera de 29 estados estables o en cualquiera de un gran número de estados transitorios. De hecho, se necesitan 7 bits para registrar el estado de cada línea de caché, como se muestra en la figura 8-33. Cada estado estable se compone de dos campos. El primero indica si la línea es la cabeza de su lista, el final de su lista, un miembro intermedio de su lista o la única entrada de la lista. El segundo contiene información acerca de si la línea está limpia, es exclusiva, etc.

El protocolo SCI define tres operaciones de gestión de listas: añadir un nodo a una lista, eliminar un nodo de una lista y eliminar todos los nodos excepto uno. La última operación se necesita cuando una línea compartida se modifica y por tanto se vuelve exclusiva.

El protocolo SCI tiene tres opciones de complejidad. El protocolo mínimo sólo permite una copia de cada línea en una caché, como en la figura 8-30. El protocolo típico permite poner en caché cada línea en un número ilimitado de nodos. El protocolo completo contiene varias características extra que mejoran el desempeño. La NUMA-Q implementa el protocolo típico, así que ése es el que explicaremos a continuación.

Consideremos el manejo de una instrucción READ. Si la CPU que ejecuta la instrucción no puede encontrar la línea de caché en su propia tarjeta, la tarjeta IQ-Link captura la solicitud en su nodo. La tarjeta envía un paquete de solicitud a la tarjeta IQ-Link base, que entonces busca en tablas el estado de la línea. Si el estado es NO EN CACHÉ, el estado se vuelve COMPARTIDO y la línea se devuelve de la memoria. Luego se actualiza la tabla de memoria local en el nodo base para que contenga una lista de un elemento que apunta al nodo en cuyo caché la línea está ahora.

Suponga ahora que el estado es COMPARTIDO. También se devuelve la línea de la memoria y el nuevo nodo se convierte en la cabeza de la lista, es decir, su número se coloca en la entrada de directorio en el nodo base. La entrada de directorio en el nodo solicitante se ajusta de modo que apunte al antiguo nodo cabeza. Este procedimiento alarga la lista en un elemento, y la nueva caché es la primera entrada.

Por último, imagine que la línea solicitada está en el estado MODIFICADO. El directorio base no puede devolver la línea de la memoria porque no tiene una copia actualizada. En vez de ello, le dice a la tarjeta IQ-Link solicitante en qué caché está la línea y le ordena traerla de ahí. También se modifica la tabla de memoria local de modo que apunte a la nueva ubicación.

Desde luego, el procesamiento de una instrucción WRITE es un poco diferente. Si el nodo solicitante ya está en la lista, deberá eliminar todas las demás entradas para convertirse en la única; si no está en la lista, deberá eliminar todas las entradas y luego insertarse en la

lista como única entrada. En todos los casos, el nodo termina siendo la única entrada válida de la lista, y es al que el directorio base apunta. Los pasos reales del manejo de instrucciones tanto READ como WRITE son muy complicados porque el protocolo debe funcionar correctamente aun cuando varias máquinas están realizando operaciones incompatibles con la misma línea. La necesidad de que el protocolo sea correcto aun durante operaciones concurrentes es la que da lugar a todos los estados transitorios. Lamentablemente, no tenemos espacio en este libro para una exposición completa del protocolo. Si desea conocer toda la historia, vea IEEE 1596.

8.3.7 Multiprocesadores COMA

Las máquinas NUMA y CC-NUMA tienen la desventaja de que las referencias a una memoria remota son mucho más lentas que las referencias a la memoria local. En CC-NUMA, el uso de cachés oculta hasta cierto punto esta diferencia en el desempeño. No obstante, si la cantidad de datos remotos requeridos excede considerablemente la capacidad de caché, ocurrirán constantemente fallos de caché y el desempeño será deficiente.

Así, tenemos la situación de que las máquinas UMA, como la Sun Enterprise 10000, tienen excelente desempeño pero su tamaño es limitado y su costo es considerable. Las máquinas NUMA se pueden escalar a tamaños un poco mayores pero requieren colocación de páginas manual o semiautomática, a menudo con resultados mixtos. El problema es que es difícil predecir cuáles páginas se necesitarán dónde y, en cualquier caso, las páginas suelen ser demasiado grandes como para estarlas moviendo de aquí para allá. Las máquinas CC-NUMA, como la Sequent NUMA-Q, podrían tener un desempeño pobre si muchas CPU necesitan muchos datos remotos. En síntesis, todos estos diseños tienen limitaciones graves.

Un tipo distinto de multiprocesador trata de superar todos estos problemas utilizando la memoria principal de cada CPU como caché. En este diseño, llamado **acceso sólo a memoria caché** (COMA, *Cache Only Memory Access*), las páginas no tienen máquinas base fijas, como sucede en las máquinas NUMA y CC-NUMA. De hecho, las páginas no son importantes.

En vez de ello, el espacio de direcciones físico se divide en líneas de caché, que migran dentro del sistema según se les necesita. Los bloques no tienen máquinas base. Al igual que los nómadas de algunos países del Tercer Mundo, el hogar es el lugar donde están en ese momento. Una memoria que simplemente atrae líneas conforme las necesita se denomina **memoria de atracción**. Utilizar la RAM principal como caché de gran capacidad eleva considerablemente la tasa de aciertos, y por ende el desempeño.

Claro que, como ya sabemos, nada es gratis. Los sistemas COMA introducen dos problemas nuevos:

1. ¿Cómo se localizan las líneas de caché?
2. Cuando una línea se elimina de la memoria, ¿qué sucede si es la última copia?

El primer problema tiene que ver con el hecho de que, una vez que la MMU ha traducido una dirección virtual a una dirección física, si la línea no está en la verdadera caché de hardware, no es fácil determinar si está o no en la memoria principal. El hardware de paginación no

ayuda aquí porque cada página se compone de muchas líneas de caché individuales que divagan por la máquina individualmente. Además, aunque se sepa que una línea no está en la memoria principal, ¿dónde está entonces? No es posible preguntarle a la máquina base, porque no hay una máquina base.

Se han propuesto algunas soluciones al problema de la localización. Para ver si una línea de caché está o no en la memoria principal, podría agregarse hardware nuevo que siga la pista a la etiqueta de cada línea que está en caché. Luego, la MMU podría comparar la etiqueta de la línea requerida con las etiquetas de todas las líneas de caché en la memoria en busca de un acierto. Esta solución requiere hardware adicional.

Una solución un poco distinta sería mapear páginas enteras pero sin exigir que estén presentes todas las líneas de caché. En esta solución, el hardware necesitaría un mapa de bits por cada página, con un bit por cada línea de caché para indicar la presencia o ausencia de esa línea. En este diseño, llamado **COMA simple**, si una línea de caché está presente, deberá estar en la posición correcta de su página, pero si no está presente, cualquier intento de usarla causará una trampa que permitirá al software ir a buscarla y traerla a la memoria.

Esto nos lleva a encontrar líneas que son realmente remotas. Una solución es asignar a cada página una máquina base en términos de dónde está su entrada de directorio, pero no dónde están los datos. Así, puede enviarse un mensaje a la máquina base para al menos localizar la línea de caché. Otros esquemas implican organizar la memoria en forma de árbol y buscar hacia arriba hasta encontrar la línea.

El segundo problema de la lista anterior tiene que ver con no eliminar la última copia. Como en CC-NUMA, una línea de caché podría estar en varios nodos al mismo tiempo. Cuando ocurre un fallo de caché, se debe traer una línea, lo que normalmente implica que hay que desalojar otra línea. ¿Qué sucede si la línea escogida para el desalojo es la última copia? En tal caso, no podemos eliminarla.

Una solución es regresar al directorio y verificar si hay otras copias. Si las hay, la línea puede desalojarse sin peligro; si no, hay que pasarla a otro lugar. Otra solución consiste en marcar una copia de cada línea de caché como copia maestra y nunca desalojarla. Esta solución evita tener que consultar el directorio. Tomando todo en cuenta, COMA promete alcanzar un mejor desempeño que CC-NUMA, pero no se han construido muchas máquinas COMA, así que se necesita más experiencia. Las dos máquinas COMA que se han construido hasta ahora son la KSR-1 (Burkhardt *et al.*, 1992) y la Data Diffusion Machine (Hagersten *et al.*, 1992). Se puede encontrar más información acerca de máquinas COMA en (Falsafi y Wood, 1997; Joe y Hennessy, 1994; Morin *et al.*, 1996; y Saulsbury *et al.*, 1995).

8.4 MULTICOMPUTADORAS DE TRANSFERENCIA DE MENSAJES

Como vimos en la figura 8-14, los dos tipos de procesadores paralelos MIMD son los multiprocesadores y las multicomputadoras. En la sección anterior estudiamos los multiprocesadores. Vimos que, desde la perspectiva del sistema operativo, los multiprocesadores parecen tener una memoria compartida a la que se accede con instrucciones LOAD y STORE

ordinarias. Como vimos, esta memoria compartida se puede implementar de varias maneras, que incluyen espiar buses, conmutadores de barras cruzadas, redes de conmutación multietapas, y diversos esquemas basados en directorios. No obstante, los programas escritos para un multiprocesador pueden accesar a cualquier localidad de memoria sin saber nada acerca de la topología interna ni del esquema de implementación. Esta ilusión es lo que hace a los multiprocesadores tan atractivos.

Por otra parte, los multiprocesadores también tienen sus limitaciones, y ésta es la razón por la que las multicomputadoras también son importantes. En primer lugar, los multiprocesadores no pueden escalarse a tamaños muy grandes. Ya vimos la enorme cantidad de hardware que Sun tuvo que usar para escalar el Enterprise 10000 a 64 CPU. La Sequent NUMA-Q llega a 256 CPU, pero el precio que se paga es un tiempo de acceso a memoria no uniforme. En contraste, ahora estudiaremos dos multicomputadoras que tienen 2048 y 9152 CPU, respectivamente. Pasarán años antes de que alguien construya un multiprocesador comercial con 9000 nodos, y para entonces ya se estarán usando multicomputadoras de 100,000 nodos.

Además, la contención por la memoria en un multiprocesador puede afectar mucho el desempeño. Si 100 CPU están tratando de leer y escribir las mismas variables constantemente, la lucha por las distintas memorias, buses y directorios puede perjudicar enormemente el desempeño.

Como consecuencia de éstos y otros factores, ha surgido un gran interés por construir y usar computadoras paralelas en las que cada CPU tiene su propia memoria privada, no accesible directamente para ninguna otra CPU. Éstas son las multicomputadoras. Los programas en CPU de multicomputadoras interactúan empleando primitivas como send y receive para transferir explícitamente mensajes, porque no pueden acceder directamente a la memoria de otra CPU con instrucciones LOAD y STORE. Esta diferencia cambia totalmente el modelo de programación.

Cada nodo de una multicomputadora consiste en una o unas cuantas CPU, algo de RAM, (tal vez compartida entre las CPU de ese nodo únicamente), un disco u otros dispositivos de E/S, y un procesador de comunicaciones. Los procesadores de comunicaciones están conectados por una red de interconexión de alta velocidad de los tipos que vimos en la sección 8.1.2. Se usan muchas topologías, esquemas de conmutación y algoritmos de enrutamiento distintos. Lo que todas las multicomputadoras tienen en común es que cuando un programa de aplicación ejecuta la primitiva send, se avisa al procesador de comunicaciones y éste transmite un bloque de datos de usuario a la máquina de destino (tal vez después de pedir y obtener permiso). En la figura 8-34 se muestra una multicomputadora genérica.

Hay multicomputadoras de todas las formas y tamaños, por lo que es difícil dar una taxonomía nítida. No obstante, destacan dos "estilos" generales: los MPP y los COW. A continuación estudiaremos cada uno de estos tipos.

8.4.1 Procesadores masivamente paralelos (MPP)

La primera categoría consiste en los **procesadores masivamente paralelos** (MPP, *Massively Parallel Processors*), que son enormes supercomputadoras que cuestan muchos millones de dólares. Éstos se usan en la ciencia, ingeniería y la industria para cálculos de gran envergadu-

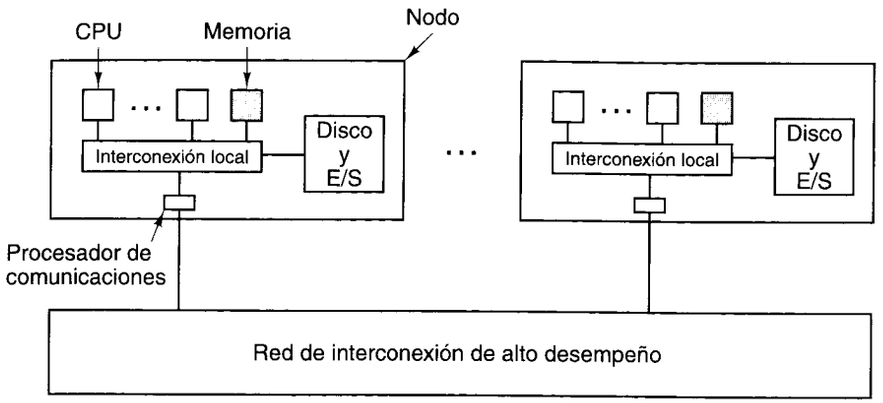


Figura 8-34. Multicomputadora genérica.

ra, para manejar números enormes de transacciones por segundo, o como bodega de datos (almacenamiento y gestión de bases de datos inmensas). En un principio los MPP se usaron primordialmente como supercomputadoras científicas, pero ahora casi todos se usan en entornos comerciales. En cierto sentido, estas máquinas son las sucesoras de las poderosas *mainframes* de los años sesenta (pero la conexión es tenue, algo así como que un paleontólogo diga que un gorrión es el sucesor de *Tyrannosaurus Rex*). En gran medida, los MPP han desplazado a las máquinas SIMD, las supercomputadoras vectoriales y los procesadores de arreglos en la cima de la cadena alimenticia digital.

Casi todas estas máquinas usan CPU estándar como procesadores. Las preferidas son la línea Pentium de Intel, el UltraSPARC de Sun, el IBM RS/6000 y el DEC Alpha. Lo que distingue a los MPP es su uso de una red de interconexión propia de alto desempeño diseñada para transferir mensajes con baja latencia y gran ancho de banda. Ambas cosas son importantes porque casi todos los mensajes son pequeños (mucho menos de 256 bytes) pero casi todo el tráfico total se debe a mensajes grandes (más de 8 KB). Los MPP también usan una gran cantidad de software y bibliotecas propios.

Otro aspecto que caracteriza a los MPP es su enorme capacidad de E/S. Los problemas lo bastante grandes como para justificar el uso de MPP casi siempre tienen cantidades enormes de datos que procesar, a veces del orden de terabytes. Estos datos deben distribuirse entre muchos discos y tienen que desplazarse dentro de la máquina a gran velocidad.

Por último, otro aspecto característico de los MPP es su atención a la tolerancia ante fallos. Con miles de CPU, son inevitables varios fallos por semana. Abortar una serie de procesamiento de 18 horas porque una CPU se cayó es inaceptable, sobre todo cuando cabe esperar un fallo de esos cada semana. Por ello, los MPP grandes siempre cuentan con hardware y software especial para monitorear el sistema, detectar fallos y recuperarse de ellos sin tener que parar.

Aunque sería agradable pasar ahora al estudio de los principios generales del diseño de los MPP, la verdad es que no hay muchos principios. En lo esencial, un MPP es una colección

de nodos de computación más o menos estándar unidos por una interconexión muy rápida de los tipos que ya estudiamos. Lo que haremos, entonces, es examinar dos ejemplos de MPP: el Cray T3E y el Intel/Sandia Option Red.

El Cray T3E

La familia Cray T3E, sucesora de la T3D, incluye las más modernas supercomputadoras de una línea que se remonta a la innovadora 6600 de Seymour Cray a mediados de los años sesenta. Los diversos modelos, el T3E, T3E-900 y T3E-1200, tienen idéntica arquitectura, y sólo difieren en su precio y su desempeño (por ejemplo, 600, 900 o 1200 megaFLOPs por CPU). Un megaFLOP es un millón de operaciones de punto flotante por segundo. A diferencia de la 6600 y la Cray-1, que tenían poco paralelismo, estas máquinas son masivamente paralelas, con hasta 2048 CPU por sistema. Usaremos el término "T3E" para referirnos a toda la familia en un sentido genérico, pero las cifras de desempeño que citemos serán para la T3E-1200. Estas máquinas son vendidas por Cray Research, una división de Silicon Graphics, y se usan para modelado del clima, diseño de fármacos, exploración petrolera y muchas otras aplicaciones.

La CPU que utiliza el T3E es la DEC Alpha 21164, que es un procesador RISC superescalar capaz de emitir cuatro instrucciones por ciclo de reloj. Esta CPU trabaja a 300, 450 o 600 MHz, dependiendo del modelo. De hecho, la velocidad del reloj es la diferencia primordial entre los distintos modelos de T3E. El Alpha es una máquina de 64 bits, con registros de 64 bits. No obstante, las direcciones virtuales están limitadas a 43 bits, y las físicas, a 40 bits, lo que permite accesar hasta 1 TB de memoria física.

Cada Alpha tiene dos niveles de cachés en el chip. El de primer nivel tiene 8 KB para instrucciones y 8 KB para datos. El de segundo nivel es una caché asociativa por conjuntos de tres vías, unificado, de 96 KB, tanto para instrucciones como para datos. No existe una caché en el nivel de tarjeta. Las cachés sólo contienen instrucciones y datos de la RAM local, que puede ser de hasta 2 GB por CPU. Con un máximo de 2048 CPU, la memoria total del sistema puede alcanzar los 4 TB.

Cada Alpha está encapsulado por circuitos especiales que reciben el nombre de **concha**, como se muestra en la figura 8-35. La concha contiene la memoria, el procesador de comunicaciones y 512 registros especiales llamados **registros E**. Estos registros se pueden cargar con direcciones de memoria remota para leer o escribir palabras (o bloques de 8 palabras) de memoria remota. Por tanto, la T3E sí tiene acceso a la memoria remota, pero no a través de las instrucciones LOAD y STORE normales. Esto hace que sea una especie de híbrido entre una máquina NC-NUMA y un MPP, pero tiene más rasgos de MPP, ya que el sistema operativo ciertamente sabe que no puede leer o escribir en la memoria remota como si fuera local. La coherencia de la memoria está garantizada porque las palabras que se leen de una memoria remota no se colocan en caché; la memoria remota es la única copia.

Los nodos T3E se conectan de dos formas distintas, como se muestra en la figura 8-35. La interconexión principal es un toroide 3D dúplex. Por ejemplo, un sistema con 512 nodos podría configurarse como un cubo de $8 \times 8 \times 8$. Cada nodo del toroide 3D tiene seis enlaces

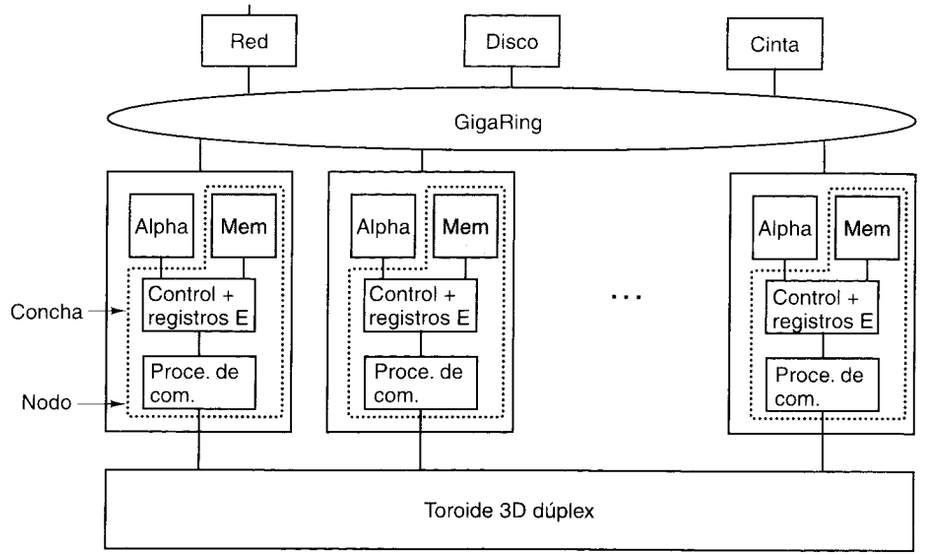


Figura 8-35. El T3E de Cray Research.

con otros nodos: sus vecinos al norte, oriente, sur, poniente, arriba y abajo. La tasa de transferencia por estos enlaces es de 480 MB/s en cada dirección.

Los nodos también están conectados por uno o más **GigaRings**, el subsistema de E/S de gran ancho de banda conmutado por paquetes. Los nodos lo usan para comunicarse entre sí y con las redes, discos y otros periféricos enviando paquetes de hasta 256 bytes por él. Cada GigaRing consiste en un par de anillos contrarrotatorios con una anchura de 32 bits que conectan los nodos de CPU con nodos de E/S especiales. Estos últimos contienen ranuras para insertar tarjetas de red (por ejemplo, HIPPI, Ethernet, ATM, FDDI o canal de fibra), discos, cintas y otros dispositivos.

Como puede haber hasta 2048 nodos en un sistema T3E, ocurren fallos con cierta regularidad. Para resolver este problema, por cada 128 nodos de usuario, el sistema contiene un nodo de repuesto. Los nodos muertos pueden sustituirse por nodos de repuesto mientras está funcionando el sistema, sin necesidad de un rearranque. Además de los nodos de usuario y los nodos de repuesto, algunos nodos se dedican a ejecutar servidores de sistema operativo porque los nodos de usuario no ejecutan el sistema operativo completo, sino sólo un núcleo básico. El sistema operativo que se usa es una versión de UNIX.

El Intel/Sandia Option Red

Las computadoras del extremo superior y las fuerzas armadas han estado entrelazadas en Estados Unidos desde 1943, cuando el ejército contrató a John Mauchly para construir ENIAC, la primera computadora electrónica moderna. La relación entre las fuerzas armadas estadounidenses y la computación de alta velocidad continúa. A mediados de los años noventa, los departamentos de la Defensa y de Energía de Estados Unidos lanzaron un ambicioso programa para efectuar avances en supercomputación diseñando cinco MPP que operarían a velocidades de 1, 3, 10, 30 y 100 teraflops/s, respectivamente. Para hacer una comparación con una computadora más conocida, 100 teraflops (10^{14} operaciones de punto flotante/s) es 500,000 veces la potencia de un Pentium Pro de 200 MHz.

A diferencia de la Cray T3E, que es un producto comercial normal que se compra en la tienda (aunque hay que llevar mucho dinero), las máquinas de teraflops son sistemas únicos otorgados mediante licitación competitiva por el Departamento de Energía, que opera los laboratorios nacionales de Estados Unidos. Intel ganó el primer contrato; IBM ganó los dos siguientes. Si planea licitar por contratos futuros, 80 millones de dólares sería una buena cifra para tenerla en mente. Dado que estas máquinas son para uso militar (primordialmente para simular explosiones nucleares), alguna persona imaginativa del Pentágono pensó en nombres patrióticos para las tres primeras: rojo, blanco y azul (los colores de la bandera estadounidense); lo malo es que se equivocó en el orden: la máquina de 1 TFLOPS/s (instalada en el Sandia National Laboratory en diciembre de 1996) se llama **Option Red** (opción roja), y las dos siguientes se llaman **Option Blue** (1999) y **Option White** (1900), respectivamente. En el resto de esta sección nos concentraremos en la primera de las máquinas de teraflops, la Option Red.

La máquina Option Red consiste en 4608 nodos dispuestos en una malla tridimensional. Las CPU están instaladas en dos tipos de tarjetas. Las tarjetas **kestrel** se usan como nodos de cómputo, y las tarjetas **eagle** (águila) se usan como nodos de servicio, disco, red y arranque. Hay 4536 nodos de cómputo, 32 nodos de servicio, 32 nodos de disco, 6 nodos de red y dos nodos de arranque.

La tarjeta kestrel, que se ilustra en la figura 8-36(a), contiene dos nodos lógicos, cada uno con dos CPU Pentium Pro de 200 MHz y 64 MB de RAM compartida. Cada nodo kestrel tiene su propio bus local de 64 bits y su propio **chip de interfaz con red** (NIC, *Network Interface Chip*). Los dos NIC están interconectados, por lo que sólo uno de ellos está vinculado realmente con la red de interconexión, a fin de hacer más compacto el paquete (aun así, la máquina ocupa un área de piso de 160 m^2). Las tarjetas eagle también tienen CPU Pentium Pro, pero sólo dos por tarjeta. Estas tarjetas tienen también capacidad de E/S adicional.

Las tarjetas están interconectadas por una estructura de cuadrícula de $32 \times 38 \times 2$ dispuesta en forma de dos planos 32×38 con todos los nodos correspondientes interconectados (el tamaño de la cuadrícula lo determinaron consideraciones de empaque, por lo que no todos los puntos de la cuadrícula tienen tarjetas). En cada punto de cuadrícula hay un chip enrutador especial con seis enlaces: norte, oriente, sur, poniente, al otro plano y a la tarjeta kestrel o eagle conectada a él. Cada enlace puede transferir 400 MB/s en cada dirección simultáneamente. Los chips enrutadores utilizan enrutamiento por túnel a fin de reducir la latencia.

Se usa enrutamiento dimensional; los paquetes pueden pasar primero al otro plano, luego en dirección oriente-poniente, luego norte-sur y por último al plano correcto, si no están ya ahí. La razón para efectuar dos traslados entre los planos es la tolerancia ante fallos. Supongamos que un paquete simplemente tiene que llegar a su vecino que está un brinco al norte, pero el enlace entre ellos está roto. En vez de ello, el mensaje puede pasar al otro plano, dar un brinco al norte y regresar al mismo plano, pasando por alto el enlace defectuoso.

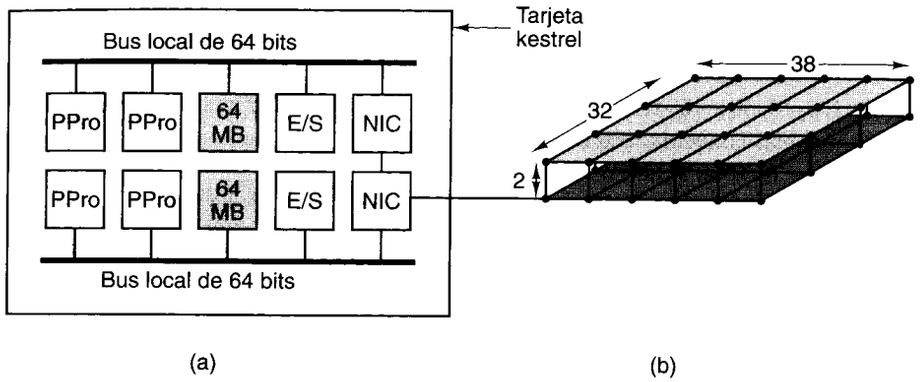


Figura 8-36. El sistema Intel/Sandia Option Red. (a) La tarjeta kestrel. (b) La red de interconexión.

El sistema se divide lógicamente en cuatro particiones: servicio, cómputo, E/S y sistema. Los nodos de servicio son máquinas UNIX de tiempo compartido, de propósito general, que permiten a los programadores escribir y depurar sus programas. Los nodos de cómputo realizan el trabajo pesado (por ejemplo, ejecutan las grandes aplicaciones numéricas para las que se ordenó la máquina). Éstos no ejecutan un sistema UNIX completo, sino un micronúcleo llamado **cougar**. Los nodos de E/S administran los 640 discos con más de 1 TB de datos. Hay dos conjuntos independientes de nodos de E/S, uno para trabajos militares confidenciales y otro para trabajos no confidenciales. Los dos conjuntos se instalan y desinstalan manualmente, de modo que sólo un conjunto está conectado en un momento dado, a fin de evitar la fuga de información de los discos confidenciales a los no confidenciales. Por último, los nodos de sistema sirven para arrancar el sistema.

8.4.2 Cúmulos de estaciones de trabajo (COW)

El otro estilo de multicomputadora es el **cúmulo de estaciones de trabajo** (COW, *Cluster Of Workstations*) o **red de estaciones de trabajo** (NOW, *Network Of Workstations*) (Anderson *et al.*, 1995; Martin *et al.*, 1997). Normalmente, un COW consiste en unos cuantos cientos de PC o estaciones de trabajo conectadas por una tarjeta de red comercial. La diferencia entre un MPP y un COW es análoga a la diferencia entre una *mainframe* y una PC. Ambas tienen una CPU, ambas tienen RAM, ambas tienen discos, ambas tienen un sistema operativo, etc. La *mainframe* simplemente tiene cosas más rápidas (con la posible excepción del sistema operativo). Sin embargo, en el aspecto cualitativo los dos sistemas se sienten diferentes y se usan y controlan de forma distinta. Esta misma diferencia existe entre los MPP y los COW.

La fuerza impulsora de los COW es la tecnología. Las CPU que se usan en los MPP no son más que procesadores comerciales que cualquiera puede comprar. La T3E usa Alphas; la Option Red usa Pentium Pros. No hay nada de especial ahí. Esos sistemas también usan DRAMs ordinarias y ejecutan UNIX. Estas cosas tampoco tienen nada fuera de lo común.

Históricamente, el elemento clave que hizo especiales a los MPP fue su interconexión de alta velocidad, pero la reciente aparición en el mercado de interconexiones de alta velocidad que se venden en tiendas ha comenzado a cerrar la brecha. Por ejemplo, el grupo de investigación del autor ha ensamblado un COW llamado **DAS** que consiste en 128 nodos, cada uno de los cuales contiene un Pentium Pro de 200 MHz y 128 MB de RAM (vea http://www.cs.vu.nl/~bal/das.html). Los nodos están conectados por un toroide bidimensional con enlaces que pueden transferir 160 MB/s en ambas direcciones a la vez. Estas especificaciones no son muy distintas de las de Option Red: los enlaces tienen la mitad de la velocidad, pero la RAM por nodo es dos veces mayor. La única diferencia real es que Sandia tiene un presupuesto mayor; técnicamente, los dos sistemas son prácticamente idénticos.

La ventaja de construir un COW en lugar de un MPP es que el primero se construye en su totalidad con componentes comerciales que se pueden adquirir y ensamblar localmente. Estas piezas tienen series de producción grandes y se benefician de las economías de escala; además, existen en un mercado competitivo, lo que tiende a elevar el desempeño y bajar el precio. En síntesis, es probable que los COW releguen a los MPP a nichos cada vez más diminutos, así como las PC han convertido a las *mainframes* en artículos de especialidad esotéricos.

Aunque existen muchos tipos de COW, dos especies dominan: los centralizados y los descentralizados. Un COW centralizado es un cúmulo de estaciones de trabajo o PC montadas en un anaquel enorme en un solo recinto. A veces se empaquetan de forma mucho más compacta que lo normal a fin de reducir el tamaño físico y la longitud de los cables. Por lo regular, las máquinas son homogéneas y no tienen más periféricos que tarjetas de red y posiblemente discos. Gordon Bell, el diseñador de la PDP-11 y la VAX, ha llamado a tales máquinas **estaciones de trabajo sin cabeza** (porque no tienen dueño). Estuvimos tentados a llamarlas COW sin cabeza, pero temimos que tal término ofendería a demasiadas vacas sagradas, por lo que nos abstuvimos.

Los COW descentralizados consisten en estaciones de trabajo dispersas dentro de un edificio o campus universitario. Casi todas ellas están ociosas muchas horas al día, sobre todo de noche, y por lo regular están conectadas a una LAN. Es común que tales estaciones de trabajo sean heterogéneas y cuenten con un surtido completo de periféricos, aunque tener un COW con 1024 ratones no es realmente mucho mejor que un COW con cero ratones. Lo más importante es que muchas de las estaciones de trabajo tienen dueños con vínculos emocionales con sus máquinas y no les hace mucha gracia que algún astrónomo trate de simular el "big bang" con la suya. Usar estaciones de trabajo ociosas para formar un COW implica tener alguna forma de trasladar trabajos a otra máquina cuando el dueño llega y reclama la suya. Tal migración de trabajos es posible pero hace más complejo el software.

8.4.3 Planificación

Una pregunta obvia es: "¿Qué diferencia hay entre un COW descentralizado y una LAN llena de máquinas de usuario?" La respuesta tiene que ver con el software y la utilización, no con el hardware. En una LAN, los usuarios tienen máquinas personales y las usan para todo su trabajo. En contraste, un COW descentralizado es un recurso compartido al cual los usuarios

pueden presentar trabajos que requieren múltiples CPU. Para atender solicitudes de múltiples usuarios, cada uno de los cuales necesita múltiples CPU, un COW (centralizado o descentralizado) necesita un planificador de trabajos.

En el modelo más sencillo, el planificador requiere que cada trabajo especifique cuántas CPU necesita. Luego los trabajos se ejecutan en orden FIFO estricto, como se muestra en la figura 8-37(a). En este modelo, una vez que se inicia un trabajo, se determina si hay suficientes CPU disponibles para iniciar el siguiente trabajo de la cola de entrada. Si las hay, el trabajo se inicia, etc.; si no, el sistema espera hasta que se desocupan suficientes CPU. Por cierto, aunque hemos sugerido que este COW tiene ocho CPU, bien podría tener 128 CPU que se asignan en unidades de 16 (lo que da ocho grupos de CPU), o alguna otra combinación.

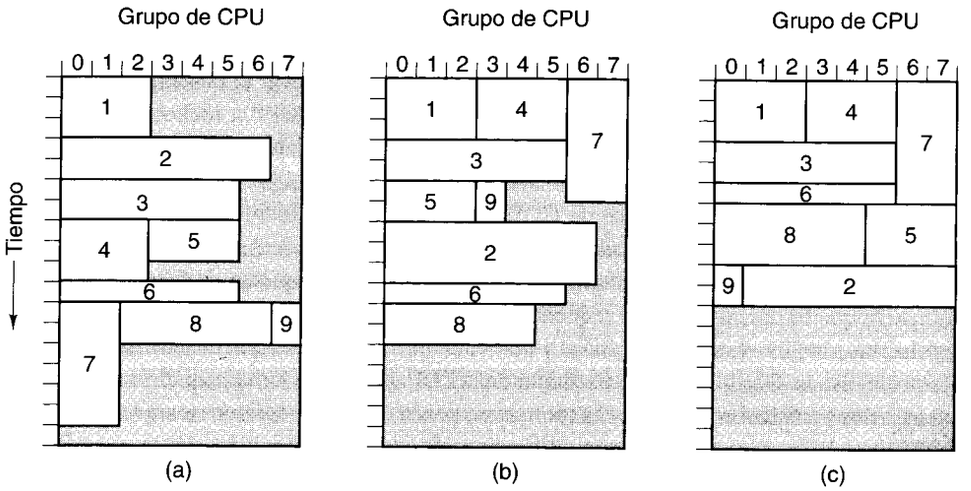


Figura 8-37. Planificación de un COW. (a) FIFO. (b) Sin bloqueo de cabeza de línea. (c) Mosaico. Las áreas sombreadas indican CPU ociosas.

Un mejor algoritmo de planificación evita el bloqueo de cabeza de línea pasando por alto trabajos que no encajan y escogiendo el primero que sí quepa. Cada vez que un trabajo termina, la cola de trabajos restantes se examina en orden FIFO. Este algoritmo da el resultado de la figura 8-37(b).

Un algoritmo de planificación todavía más sofisticado requiere que cada trabajo presentado especifique sus dimensiones, es decir, cuántas CPU durante cuántos minutos. Con esa información, el planificador de trabajos puede tratar de cubrir el rectángulo CPU-tiempo con "mosaicos". Esta técnica es eficaz sobre todo cuando los trabajos se presentan durante el día para ejecutarse durante la noche, de modo que el planificador cuenta con toda la información acerca de todos los trabajos por adelantado y puede ejecutarlos en el orden óptimo, como se ilustra en la figura 8-37(c).

Aunque hemos bosquejado dos extremos en el mundo de las multicomputadoras, enormes MPP patentados y sencillos COW "hechos en casa", existe mucho terreno entre ellos. En

particular, muchos fabricantes de computadoras también entienden que pueden construir COW con componentes que se venden comercialmente y venderlos como sistemas completos. El resultado ha sido una rápida expansión del mercado.

Redes de interconexión comerciales

Puesto que los COW tienen éxito o fracasan dependiendo de la cantidad de las tarjetas de red insertables que pueden obtenerse comercialmente, vale la pena examinar algunas de las tecnologías que están disponibles. Nuestro primer ejemplo es Ethernet, que viene en tres versiones, lenta, de media velocidad y rápida (o, más correctamente, Ethernet clásica, Ethernet rápida y Ethernet de gigabits). Estos tres tipos operan a 10, 100 y 1000 Mbps (1.25, 12.5 y 125 MB/s), respectivamente. Los tres son compatibles en términos de medios, formato de paquetes y protocolos. Sólo el desempeño es diferente.

Cada computadora de una Ethernet tiene un chip de Ethernet, por lo regular en una tarjeta insertable. En su forma más antigua y sencilla, un cable de la tarjeta se insertaba a la fuerza en medio de un cable de cobre grueso, en una disposición conocida como **derivación vampiro** (¿quién, si no un vampiro, mordería un cable coaxial?). Posteriormente se comenzaron a usar cables más delgados y conectores con forma de T. En todos los casos, las tarjetas Ethernet de todas las máquinas están conectadas eléctricamente, como si se hubieran soldado juntas. En la figura 8-38(a) se ilustra la situación de tres máquinas conectadas a una Ethernet.

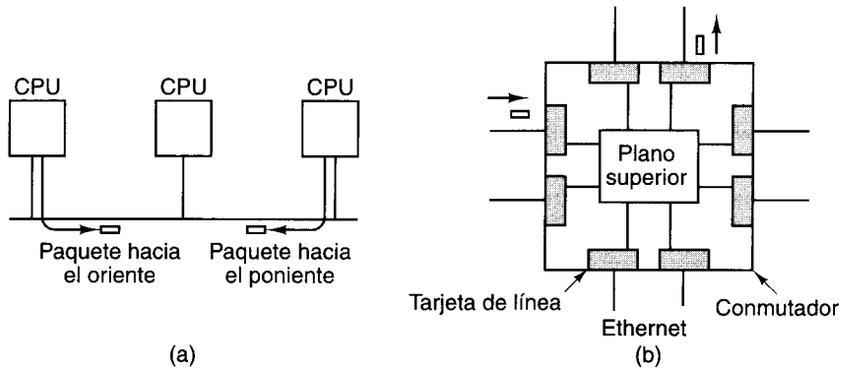


Figura 8-38. (a) Tres computadoras en una Ethernet. (b) Conmutador Ethernet.

En el protocolo Ethernet básico, cuando una máquina quiere enviar un paquete primero sondea el cable para ver si alguien más está transmitiendo actualmente. Si el cable está ocioso, la máquina envía su paquete. Si el cable está ocupado, la máquina espera hasta que la transmisión termina antes de transmitir. Si dos transmisores comienzan a transmitir en el mismo instante, ocurre una colisión. Los dos detectan la colisión, se detienen, esperan cada uno un tiempo aleatorio, y lo vuelven a intentar. Si ocurre otra colisión, se paran y lo vuelven a intentar otra vez, duplicando el tiempo de espera medio después de cada colisión sucesiva.

Un problema con la Ethernet básica de la figura 8.38(a) es que las derivaciones vampiro se rompen fácilmente y detectar un fallo de cable no es tarea fácil. Por esta razón comenzó a

usarse un diseño modificado, en el que un cable de cada máquina llega a un **centro**. Eléctricamente, este diseño es igual al primero, pero el mantenimiento es más fácil porque los cables pueden desconectarse del centro uno por uno hasta que se aísla el cable defectuoso.

Una vez que todas las máquinas tenían un cable tendido hacia el cuarto de máquinas, se hizo posible un tercer diseño: **Ethernet conmutada**, el cual se ilustra en la figura 8-38(b). Aquí el centro es sustituido por un dispositivo que contiene un plano posterior en el que pueden insertarse **tarjetas de línea**. Cada tarjeta de línea acepta una o más Ethernets, y diferentes tarjetas pueden manejar diferentes velocidades, lo que permite conectar entre sí Ethernets clásica, rápida y de Gigabits.

Cuando un paquete llega a una tarjeta de línea, se almacena temporalmente ahí hasta que la tarjeta solicita y le es concedido acceso al plano posterior, el cual funciona de forma parecida a un bus. Una vez que el paquete se ha transferido a la tarjeta de línea a la que está conectada la máquina de destino, se le puede enviar ahí. Si cada tarjeta de línea tiene sólo una Ethernet, y ésta sólo tiene una máquina, ya no ocurren colisiones, aunque sigue siendo posible perder un paquete por desbordamiento de un buffer en una tarjeta de línea. Si se usa como interconexión de multicomputadora, la Ethernet de gigabits conmutada, con una máquina por Ethernet y un plano posterior muy rápido, tiene un desempeño potencial (al menos en términos de ancho de banda) de la cuarta parte del desempeño de los enlaces de una T3E, pero cuesta menos de una cuarta parte del precio.

Si esto suena demasiado bueno para ser verdad, lo es. Si el número de tarjetas de línea es grande, un plano posterior con un precio razonable no podrá manejar la carga, y será necesario colocar varias máquinas en cada Ethernet, con lo que se reintroducen colisiones. Con todo, en términos de potencia por dólar (es decir, precio/desempeño), la Ethernet de gigabits conmutada es un competidor de respeto.

Una segunda tecnología de interconexión es **modo de transferencia asincrónico**, (ATM, *Asynchronous Transfer Mode*). ATM fue inventado por un consorcio de alcance mundial formado por compañías telefónicas y otros organismos interesados, como sustituto totalmente digital del sistema telefónico actual. La idea era que cada teléfono y cada computadora del mundo estuvieran conectados por un tubo de bits digital libre de errores con una velocidad de por lo menos 155 Mbps, y posteriormente 622 Mbps. Para no hacer largo el cuento, fue más fácil proponer esto que hacerlo realidad. No obstante, muchas compañías ahora fabrican tarjetas insertables para PC que operan a 155 Mbps o 622 Mbps. La segunda velocidad, llamada **OC-12**, es probablemente lo bastante buena como para ser la interconexión de una multicomputadora.

El alambre o fibra que sale de una tarjeta ATM llega a un conmutador ATM, un dispositivo con cierto parecido a un conmutador Ethernet en cuanto a que los paquetes llegan y se almacenan temporalmente en tarjetas de línea hasta que pueden pasar a la tarjeta de línea de salida para ser transmitidos a su destino. No obstante, Ethernet y ATM tienen algunas diferencias importantes.

En primer lugar, dado que ATM se diseñó para reemplazar al sistema telefónico, está orientado hacia las conexiones. Antes de enviar un paquete a un destino, la máquina de origen tiene que establecer un circuito virtual desde el origen, a través de uno o más conmutadores ATM, hasta el destino. En la figura 8-39 se muestran dos circuitos virtuales. En contraste, Ethernet no usa circuitos virtuales. Puesto que establecer un circuito virtual toma un tiempo

apreciable, para poder usarse en una interconexión de multicomputadora cada máquina establecería normalmente un circuito virtual con todas las demás máquinas desde el principio y los usaría durante toda la ejecución. Los paquetes que se envían en sucesión por un circuito virtual nunca se entregan en desorden, pero los buffers de las tarjetas de línea pueden desbordarse, como en el caso de la Ethernet conmutada, de modo que la entrega no está garantizada con ninguna de las dos tecnologías.

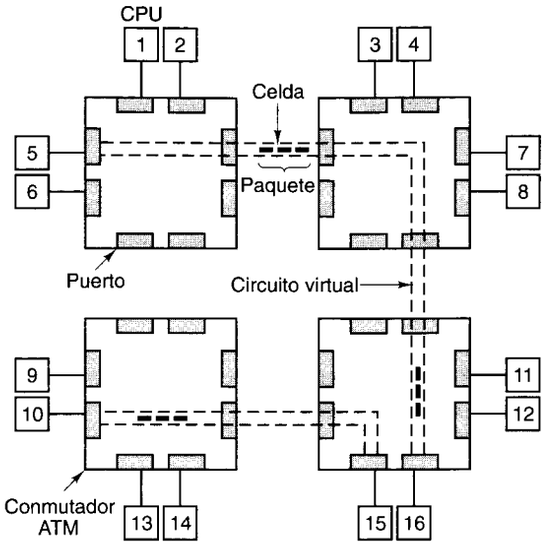


Figura 8-39. Dieciséis CPU conectadas por cuatro conmutadores ATM. Se muestran dos circuitos virtuales.

Otra diferencia entre ATM y Ethernet es que Ethernet puede transmitir paquetes enteros (hasta 1500 bytes de datos) como una sola unidad. En ATM todos los paquetes se dividen en celdas de 53 bytes (ATM fue diseñado por un comité; ¿qué esperaba usted?). Cinco de estos bytes son campos de cabecera que indican a cuál circuito virtual pertenece la celda, qué tipo de celda es, qué prioridad tiene, y varias otras cosas. La porción de carga útil es de 48 bytes. La división de paquetes en celdas y el reensamblado en el otro extremo corre por cuenta de hardware.

Nuestro tercer ejemplo de red de interconexión comercial es Myrinet, una tarjeta insertable producida por una compañía de California que se ha popularizado entre muchos diseñadores de COW (Boden *et al.*, 1995). Myrinet usa el mismo modelo que Ethernet y ATM en cuanto a que cada tarjeta insertable se conecta con un conmutador, y los conmutadores pueden interconectarse en una topología arbitraria. Los enlaces Myrinet son dúplex, de 1.28 Gbps en cada sentido. El tamaño de los paquetes no está limitado por el hardware, y los conmutadores son de barras totalmente cruzadas, lo que les confiere baja latencia y también alto ancho de banda.

Lo que hace a Myrinet interesante para los diseñadores de COW es que las tarjetas insertables contienen una CPU programable y una buena cantidad de RAM. Aunque Myrinet viene con un sistema operativo estándar, una docena de grupos universitarios han escrito el

suyo propio. Muchos de éstos añaden características diseñadas para ampliar la funcionalidad y mejorar el desempeño (por ejemplo, Blumrich *et al.*, 1995; Pakin *et al.*, 1997; Verstoep *et al.*, 1996). Entre las características típicas están protección, control de flujo, difusión y multidifusión confiables, y la capacidad para ejecutar una parte del código de la aplicación en la tarjeta.

8.4.4 Software de comunicación para multicomputadoras

Para programar una multicomputadora se requiere software especial, por lo regular bibliotecas, que se encargue de la comunicación entre procesos y la sincronización. En esta sección hablaremos un poco acerca de ese software. En su mayor parte, los mismos paquetes de software se ejecutan en MPP y COW, así que no es difícil trasladar las aplicaciones de una plataforma a otra.

Los sistemas de transferencia de mensajes tienen dos o más procesos que se ejecutan de forma independiente. Por ejemplo, un proceso podría estar produciendo ciertos datos y otro u otros procesos podrían estarlos consumiendo. No existe garantía de que cuando el transmisor tenga más datos los receptores estarán listos para recibirlos, ya que cada uno ejecuta su propio programa.

Casi todos los sistemas de transferencia de mensajes cuentan con dos primitivas (por lo regular llamadas de biblioteca), send y receive, pero puede haber varios tipos de semánticas distintas. Las tres principales variantes son:

1. Transferencia de mensajes sincrónica.
2. Transferencia de mensajes con buffers.
3. Transferencia de mensajes no bloqueadora.

En la **transferencia de mensajes sincrónica**, si el transmisor ejecuta un send y el receptor todavía no ha ejecutado un receive, el transmisor se bloquea (suspende) hasta que el receptor ejecuta un receive, y en ese momento se copia el mensaje. Cuando el transmisor recupera el control después de la llamada, sabe que el mensaje se envió y se recibió correctamente. Este método tiene la semántica más sencilla y no requiere buffers. Una desventaja importante es que el transmisor permanece bloqueado hasta que el receptor ha obtenido el mensaje y ha confirmado tal recepción.

En la **transferencia de mensajes con buffers**, cuando se envía un mensaje antes de que el receptor esté listo, el mensaje se coloca en un buffer en algún lado, por ejemplo en un buzón, hasta que el receptor lo saca. Así, en la transferencia de mensajes con buffers un transmisor puede continuar después de un send, aunque el receptor esté ocupado con otra cosa. Puesto que el mensaje realmente se envió, el transmisor está en libertad de reutilizar el buffer de mensajes de inmediato. Este esquema reduce el tiempo que el transmisor tiene que esperar. Básicamente, tan pronto como el sistema ha enviado el mensaje el transmisor puede continuar. Sin embargo, el transmisor no tiene ninguna garantía de que el mensaje se recibió correctamente. Incluso si la comunicación es confiable, el receptor podría haberse caído antes de recibir el mensaje.

En la **transferencia de mensajes no bloqueadora**, se permite al transmisor continuar de inmediato después de efectuar la llamada. Lo único que la biblioteca hace es pedir al sistema operativo que lleve a cabo la llamada después, cuando tenga tiempo. El resultado es que el transmisor prácticamente no se bloquea. La desventaja de este método es que cuando el transmisor continúa después del send no puede reutilizar el buffer de mensajes, porque es posible que el mensaje todavía no se haya enviado. El transmisor necesita una forma de averiguar si ya puede reutilizar el buffer. Una idea es hacer que le pregunte repetidamente al sistema. La otra es recibir una interrupción cuando el buffer esté disponible. Ninguna de estas cosas simplifica el software.

En las dos secciones que siguen examinaremos brevemente dos sistemas de transferencia de mensajes que se usan en muchas multicomputadoras: PVM y MPI. Éstos no son los únicos, pero son los más ampliamente utilizados. PVM es más antiguo y fue el estándar *de facto* durante años, pero ahora MPI es el muchacho nuevo en el barrio y está retando al campeón reinante.

PVM — Máquina Virtual Paralela

PVM (*Parallel Virtual Machine*) es un sistema de transferencia de mensajes del dominio público diseñado inicialmente para ejecutarse en COW basadas en UNIX (Geist *et al.*, 1994; Sunderram, 1990). Posteriormente se le ha trasladado a muchas otras plataformas, incluidos casi todos los MPP comerciales. Se trata de un sistema autosuficiente, que incluye gestión de procesos y apoyo de E/S.

PVM consta de dos partes: una biblioteca que el usuario puede invocar y un proceso demonio que se ejecuta todo el tiempo en cada máquina de la multicomputadora. Cuando se inicia PVM, éste determina cuáles máquinas van a formar parte de su multicomputadora virtual leyendo un archivo de configuración. En cada una de esas máquinas se inicia un demonio PVM. Es posible añadir y eliminar máquinas emitiendo comandos en la consola PVM.

PVM permite iniciar *n* procesos paralelos con el comando de consola

```
spawn –count n prog
```

Cada proceso ejecutará *prog*. PVM decide dónde colocará los procesos, pero un usuario puede supeditar esa decisión con argumentos del comando spawn. También se pueden iniciar procesos desde un proceso en ejecución invocando *Pvm_spawn*. Los procesos pueden organizarse en grupos cuya composición cambia dinámicamente durante la ejecución.

La comunicación en PVM se maneja con primitivas de transferencia de mensajes explícitas y de tal manera que se puedan comunicar máquinas que usen diferentes representaciones de los números. Cada proceso PVM tiene un buffer de envío activo y un buffer de recepción activo en cualquier instante. Para enviar un mensaje, un proceso invoca procedimientos de biblioteca para empacar valores en el buffer de envío activo en un formato que se describe a sí mismo, de modo que el receptor pueda reconocerlos y convertirlos a su formato nativo.

Una vez que se ha armado el mensaje, el transmisor invoca el procedimiento de biblioteca *pvm_send*, que es una transmisión bloqueadora. El receptor tiene varias opciones. La más sencilla es *pvm_recv*, que bloquea al receptor hasta que llega un mensaje apropiado. Cuando la llamada regresa el mensaje está en el buffer de recepción activo, y puede desempacarse y convertirse en valores apropiados para su máquina utilizando una serie de procedimientos de

desempacado. Si un receptor está preocupado por la posibilidad de bloquearse indefinidamente, puede invocar *pvm_trecv*, que bloquea al receptor durante cierto tiempo, pero si en ese tiempo no llega un mensaje apropiado se desbloquea e informa del fracaso. Una opción más es *pvm_nrecv*, que regresa inmediatamente, sea con un mensaje o con una indicación de que no ha llegado ningún mensaje. Esta llamada puede efectuarse una y otra vez para ver si han llegado mensajes.

Además de estas primitivas para mensajes de punto a punto, PVM también maneja difusión (*pvm_bcast*) y multidifusión (*pvm_mcast*). La primera transmite a todos los procesos de un grupo; la segunda transmite a una lista específica de procesos.

La sincronización entre procesos se efectúa usando *pvm_barrier*. Cuando un proceso invoca este procedimiento, se bloquea hasta que cierto número de otros procesos han llegado también a la barrera y han efectuado la misma llamada. Hay otras llamadas PVM para gestión de anfitriones, gestión de grupos, gestión de buffers, señalización, verificación de estado y funciones diversas. En síntesis, PVM es un paquete sencillo, directo, fácil de usar, disponible en casi todas las computadoras paralelas, todo lo cual explica su popularidad.

MPI—Interfaz de Transferencia de Mensajes

Un segundo paquete de comunicaciones para programar multicomputadoras es **MPI** (*Message-Passing Interface*). MPI es mucho más rico y complejo que PVM, con muchas más llamadas de biblioteca, muchas más opciones y muchos más parámetros por llamada. La versión original de MPI, ahora llamada MPI-1, fue aumentada por MPI-2 en 1997. A continuación presentaremos una introducción muy superficial a MPI-1, y luego hablaremos un poco acerca de lo que se añadió en MPI-2. Si desea más información acerca de MPI, vea (Gropp *et al.*, 1994; y Snir *et al.*, 1996).

MPI-1 no se ocupa de la creación o gestión de procesos, como hace PVM. Corresponde al usuario crear procesos utilizando llamadas al sistema locales. Una vez creados, los procesos se acomodan en grupos estáticos, inmutables. Es con estos grupos que MPI trabaja.

MPI se basa en cuatro conceptos ortogonales: comunicadores, tipos de datos de mensajes, operaciones de comunicación y topologías virtuales. Un **comunicador** es un grupo de procesos más un contexto. Un contexto es un rótulo que identifica algo, como una fase de ejecución. Cuando se envían y reciben mensajes, el contexto puede servir para evitar que mensajes no relacionados entre sí se interfieran.

Los mensajes se tipifican, y se reconocen muchos tipos de datos, que incluyen caracteres, enteros, cortos, normales y largos, números de punto flotante de precisión sencilla y doble, etc. También es posible construir otros tipos derivados de éstos.

MPI maneja un amplio conjunto de operaciones de comunicación. La más básica sirve para enviar mensajes como sigue:

```
MPI_Send(buffer, cuenta, tipo_datos, destino, etiqueta, comunicador)
```

Esta llamada envía al destino un buffer con *cuenta* de elementos del tipo de datos especificados. El campo *etiqueta* rotula el mensaje para que el receptor pueda decir que sólo quiere recibir un mensaje con esa etiqueta. El último campo indica en cuál grupo de procesos está el destino

(el campo *destino* sólo es un índice dentro de la lista de procesos para el grupo especificado). La llamada correspondiente para recibir un mensaje es

```
MPI_Recv(&buffer, cuenta, tipo_datos, origen, etiqueta, comunicador, &estado)
```

que anuncia que el receptor está buscando un mensaje de cierto tipo proveniente de cierto origen, con cierta etiqueta.

MPI reconoce cuatro modos de comunicación básicos. El modo 1 es sincrónico, en el que el transmisor no puede comenzar a transmitir hasta que el receptor haya invocado a MPI_Recv. El modo 2 usa buffers, por lo que no aplica la restricción anterior. El modo 3 es estándar, que depende de la implementación y puede ser sincrónico o usar buffers. El modo 4 es "listo", en el que el transmisor asegura que el receptor está disponible (como en el sincrónico), pero no se comprueba que sea cierto. Cada una de estas primitivas tiene una versión bloqueadora y una no bloqueadora, así que hay ocho primitivas en total. La recepción sólo tiene dos variantes: bloqueadora y no bloqueadora.

MPI maneja la comunicación colectiva, incluidas la difusión, dispersión/reunión, intercambio total, agregado y barrera. En todas las formas de comunicación colectiva, todos los procesos de un grupo deben emitir la llamada con argumentos compatibles. Si no se hace esto ocurre un error. Una forma típica de comunicación colectiva aplica a procesos organizados en forma de árbol, en el que los valores se propagan hacia arriba, desde las hojas hasta la raíz, sufriendo cierto procesamiento en cada paso; por ejemplo obteniendo la sumatoria o el máximo de los valores.

El cuarto concepto básico en MPI es la **topología virtual**, que permite acomodar los procesos en un árbol, anillo, cuadrícula, toroide u otras topologías. Un acomodo así permite nombrar caminos de comunicación y facilita la comunicación.

MPI-2 añade procesos dinámicos, acceso a memorias remotas, comunicación colectiva no bloqueadora, apoyo para E/S escalable, procesamiento en tiempo real y muchas características nuevas que rebasan el alcance de este libro. En la comunidad científica prevalece actualmente una lucha entre los partidarios de MPI y de PVM. El lado de PVM dice que PVM es más fácil de aprender y más sencillo de usar. El lado de MPI dice que MPI hace más y también señala que es un estándar formal con un comité de estandarización y un documento de definición oficial. El lado de PVM dice que es cierto, y asegura que la falta de una burocracia de estandarización con todas las de la ley no necesariamente es una desventaja.

8.4.5 Memoria compartida en el nivel de aplicaciones

Por nuestros ejemplos, debe ser obvio que es más fácil aumentar la escala de una multicomputadora que de un multiprocesador. El Sun Enterprise 10000, con un máximo de 64 CPU, y el NUMA-Q, con un máximo de 256 CPU, son representativos de los multiprocesadores UMA y NUMA grandes, respectivamente. En contraste, las máquinas T3E y Option Red pueden tener 2048 y 9416 CPU, respectivamente. Podemos debatir eternamente la importancia de estas cifras, pero no podemos escapar la conclusión de que las multicomputadoras pueden ser mucho más grandes que los multiprocesadores. Esto siempre ha sido cierto y seguirá siendo cierto durante muchos años más.

Sin embargo, las multicomputadoras no tienen memoria compartida en el nivel de arquitectura, lo que dio pie al desarrollo de sistemas de transferencia de mensajes como PVM y MPI. A muchos programadores no les gusta este modelo y quisieran tener la ilusión de memoria compartida, aunque no esté realmente ahí. Si se puede lograr este objetivo se tendría lo mejor de dos mundos: hardware grande de bajo costo (al menos por nodo) y facilidad de programación. Éste es el Santo Grial de la computación en paralelo.

Muchos investigadores han llegado a la conclusión de que si bien no es fácil aumentar la escala de la memoria compartida en el nivel de arquitectura, puede haber otras formas de alcanzar la misma meta. En la figura 8-3 vemos que hay otros niveles en el que puede introducirse una memoria compartida. En las secciones que siguen veremos algunas formas de introducir memoria compartida en el modelo de programación de una multicomputadora sin que esté presente en el nivel de hardware.

Memoria compartida distribuida

Una clase de sistema de memoria compartida en el nivel de aplicación es el sistema basado en páginas que se conoce como **memoria compartida distribuida** (DSM, *Distributed Shared Memory*). La idea es sencilla: una colección de CPU en una multicomputadora comparte un espacio de direcciones paginado común. En la versión más simple, cada página se tiene en la RAM de una y sólo una CPU. En la figura 8-40(a) vemos un espacio de direcciones virtual compartido que consiste en 16 páginas, distribuidas entre cuatro CPU.

Cuando una CPU hace referencia a una página en su propia RAM local, la lectura o escritura se efectúa sin retraso. En cambio, cuando una CPU hace referencia a una página en una memoria remota obtiene un fallo de página. La cuestión es que en lugar de traer la página faltante del disco, el sistema de tiempo de ejecución o sistema operativo envía un mensaje al nodo que tiene la página ordenándole que anule la correspondencia de esa página con su memoria y la envíe. Una vez que la página llega, se establece una correspondencia con la memoria local y la instrucción que causó la falla se reinicia, igual que con un fallo de página normal. En la figura 8-40(b) vemos la situación después de que la CPU 0 ha causado un fallo por la página 10: esa página se traslada de la CPU 1 a la CPU 0.

Esta idea básica se implementó por primera vez en IVY (Li y Hudak, 1986, 1989). Se obtiene una memoria totalmente compartida, secuencialmente consistente, en una multicomputadora. Sin embargo, pueden efectuarse muchas optimaciones para mejorar el desempeño. La primera optimación, presente en IVY, es permitir que páginas que están marcadas como sólo de lectura estén presentes en varios nodos al mismo tiempo. Así, cuando ocurre un fallo de página, se envía una copia de la página a la máquina que causó el fallo, pero el original se queda donde estaba porque no hay peligro de conflictos. En la figura 8-40(c) se ilustra la situación de dos CPU que comparten una página sólo de lectura (página 10).

Aun con esta optimación, el desempeño muchas veces es inaceptable, sobre todo cuando un proceso está escribiendo activamente unas cuantas palabras en la parte superior de la página y otro proceso en una CPU diferente está activamente escribiendo más palabras en la parte inferior de la página. Puesto que sólo existe una copia de la página, ésta tiene que rebotarse de un lado al otro constantemente, situación que se conoce como **compartimiento falso**.

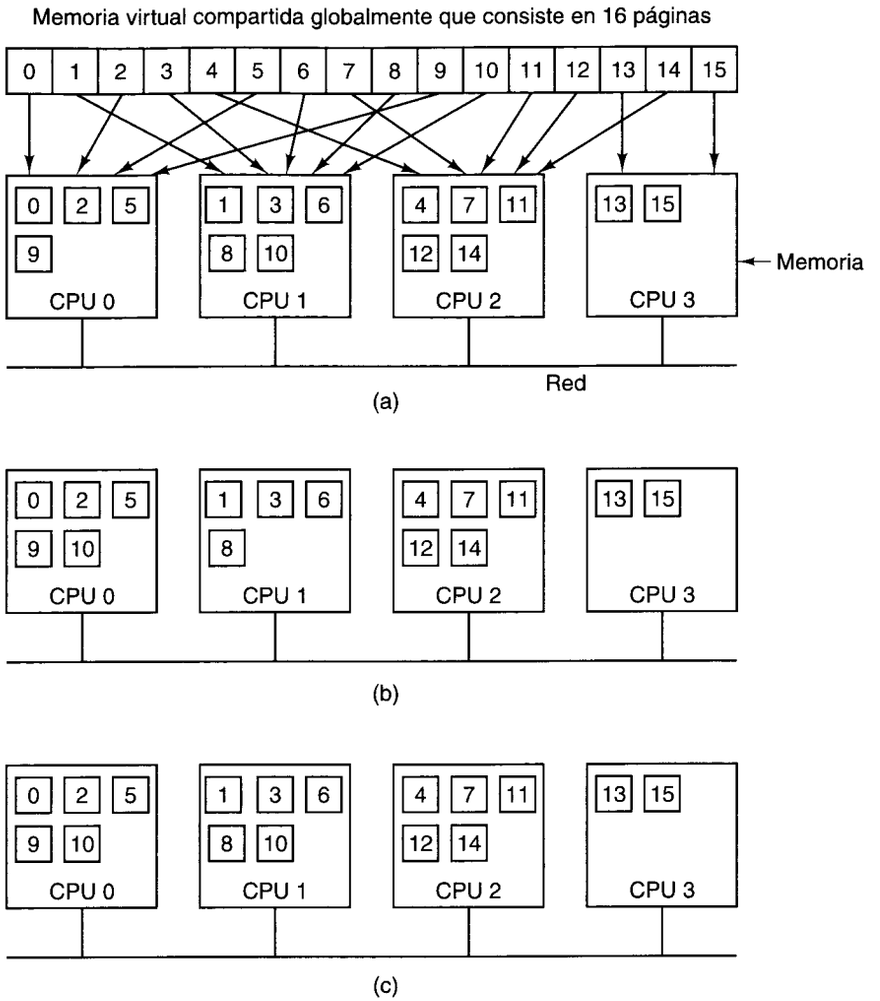


Figura 8-40. Espacio de direcciones virtual que consiste en 16 páginas distribuidas en cuatro nodos de una multicomputadora. (a) La situación inicial. (b) Después de que la CPU 0 hace referencia a la página 10. (c) Después de que la CPU 1 hace referencia a la página 10, que aquí se supone es sólo de lectura.

El problema del compartimiento falso se puede atacar de varias maneras. En el sistema Treadmarks, por ejemplo, se abandona una memoria secuencialmente consistente en favor de la consistencia de liberación (Amza, 1996). Páginas que podrían ser escribibles pueden estar presentes en varios nodos al mismo tiempo, pero antes de efectuar una escritura un proceso debe efectuar una operación acquire para indicar su intención. En ese punto, todas las copias con excepción de la más reciente se invalidan. No puede crearse ninguna otra copia hasta que se ejecute el release correspondiente, y entonces la página puede volverse a compartir.

Una segunda optimación que se efectúa en Treadmarks es establecer inicialmente la correspondencia de cada página escribible en modo de sólo lectura. La primera vez que se escribe en la página, ocurre una falla de protección y el sistema crea una copia de la página, llamada **gemela**. Entonces se establece la correspondencia de la página como de lectura-escritura y las escrituras subsecuentes pueden efectuarse a toda velocidad. Si después ocurre un fallo de página remota y la página tiene que enviarse allá, se efectúa una comparación palabra por palabra entre la página actual y la gemela. Sólo se envían las palabras que se modificaron, lo que reduce el tamaño de los mensajes.

Cuando ocurre un fallo de página, es preciso localizar la página faltante. Puede haber varias soluciones, incluidas las que se usan en máquinas NUMA y COMA, como los directorios (en la base). De hecho, muchas de las soluciones que se usan en DSM también son aplicables a NUMA y COMA porque DSM en realidad sólo es una implementación en software de NUMA o COMA en la que cada página se trata como una línea de caché.

DSM es un área de investigación muy activa. Entre los sistemas interesantes están CASHMERE (Kontothanassis *et al.*, 1997; Stets *et al.*, 1997), CRL (Johnson *et al.*, 1995), Shasta (Scales *et al.*, 1996) y Treadmarks (Amza, 1996; Lu *et al.*, 1997).

Linda

Los sistemas DSM basados en páginas como IVY y Treadmarks usan el hardware de MMU para atrapar los accesos a páginas faltantes. Si bien es útil hacer y enviar diferencias en lugar de páginas enteras, persiste el hecho de que las páginas son una unidad no natural para compartir. Por ello, se han intentado otros enfoques.

Uno de ellos es Linda, que proporciona a procesos de múltiples páginas una memoria compartida distribuida altamente estructurada (Carriero y Gelernter, 1986, 1989). El acceso a esta memoria es a través de un conjunto pequeño de operaciones primitivas que se pueden añadir a lenguajes existentes, como C y FORTRAN, para formar lenguajes paralelos, en este caso C-Linda y FORTRAN-Linda.

El concepto unificador en que se basa Linda es el de un **espacio de tuplas** abstracto, que es global para todo el sistema y accesible para todos sus procesos. El espacio de tuplas es como una memoria compartida global, sólo que con cierta estructura integrada. El espacio de tuplas contiene cierto número de **tuplas**, cada una de las cuales consiste en uno o más campos. En el caso de C-Linda, los tipos de campos incluyen enteros, enteros largos y números de punto flotante, además de tipos compuestos como arreglos (que incluyen cadenas) y estructuras (pero no otras tuplas). En la figura 8-41 se muestran tres tuplas como ejemplos.

Se proporcionan cuatro operaciones con tuplas. La primera, out, coloca una tupla en el espacio de tuplas. Por ejemplo,

```
out("abc", 2, 5);
```

coloca la tupla ("abc", 2, 5) en el espacio de tuplas. Los campos de out normalmente son constantes, variables o expresiones, como en

```
out("matrix–1", i, j, 3.14);
```



```
("abc", 2, 5)
("matrix-1", 1,6, 3.14)
("familia", "es hermana", Carolina, Leonor)
```

Figura 8-41. Tres tuplas Linda.

que produce una tupla con cuatro campos, de los cuales el segundo y el tercero están determinados por los valores actuales de las variables *i* y *j*.

Las tuplas se recuperan del espacio de tuplas con la primitiva in. El direccionamiento es por contenido, más que por nombre o dirección. Los campos de in pueden ser expresiones o parámetros formales. Considere, por ejemplo,

```
in("abc", 2, ? i);
```

Esta operación examina el espacio de tuplas en busca de una tupla que consista en la cadena "abc", el entero 2 y un tercer campo que contenga cualquier entero (suponiendo que *i* es un entero). Si se encuentra, la tupla se saca del espacio de tuplas y se asigna a la variable *i* el valor del tercer campo. La comparación y la eliminación son atómicas, así que si dos procesos ejecutan la misma operación in simultáneamente, sólo uno de ellos tendrá éxito, a menos que estén presentes dos o más tuplas que coincidan. El espacio de tuplas podría incluso contener varias copias de la misma tupla.

El algoritmo de comparación que in usa es sencillo. Los campos de la primitiva in, llamados **plantilla**, se comparan (conceptualmente) con los campos correspondientes de cada tupla del espacio de tuplas. Hay coincidencia si se cumplen las tres condiciones siguientes:

1. La plantilla y la tupla tienen el mismo número de campos.
2. Los tipos de los campos correspondientes son iguales.
3. Cada constante o variable de la plantilla coincide con su campo de tupla.

Los parámetros formales, que se indican con un signo de interrogación seguido de un nombre o tipo de variable, no participan en la comparación (excepto para verificar el tipo), aunque a los que contienen un nombre de variable se les asigna un valor si la comparación tiene éxito.

Si ninguna de las tuplas presentes coincide, el proceso invocador se suspende hasta que otro proceso inserta la tupla requerida; en ese momento el invocador se revive automáticamente y recibe la nueva tupla. El hecho de que los procesos se bloquean y desbloquean automáticamente implica que si un proceso está a punto de producir una tupla y otro está a punto de capturarla, no importa cuál vaya primero.

Además de out e in, Linda tiene una primitiva read, que es igual a in sólo que no elimina la tupla del espacio de tuplas. También hay una primitiva eval que hace que sus parámetros se evalúen en paralelo y la tupla resultante se deposite en el espacio de tuplas. Este mecanismo

puede servir para realizar un cálculo arbitrario. Es así como se crean procesos paralelos en Linda.

Un paradigma de programación común en Linda es el **modelo de trabajador repetido**. Este modelo se basa en la idea de un **almacén de tareas** (*task bag*) llena de trabajos que hay que realizar. El proceso principal inicia ejecutando un ciclo que contiene

```
out("task-bag", job);
```

en el que se introduce en el espacio de tuplas una descripción de trabajo distinta en cada iteración. Cada trabajador inicia obteniendo una tupla de descripción de trabajo mediante

```
in("task-bag", ?job);
```

y luego lleva a cabo el trabajo; cuando termina, obtiene otro trabajo. También es posible colocar trabajos nuevos en el almacén de tareas durante la ejecución. De esta forma tan sencilla, el trabajo se reparte dinámicamente entre los trabajadores y todos los trabajadores se mantienen ocupados continuamente, con muy poco gasto extra.

Existen diversas implementaciones de Linda en sistemas de multicomputadora. En todos ellos, un aspecto clave es cómo se distribuyen las tuplas entre las máquinas y cómo se les puede localizar si es necesario. Diversas posibilidades incluyen difusión y directorios. El copiado es otro aspecto importante. Estos puntos se tratan en (Bjornson, 1993).

Orca

Una estrategia un tanto distinta para tener memoria compartida en el nivel de aplicaciones en una multicomputadora es usar como unidades de compartimiento objetos con todas las de la ley, en lugar de sólo tuplas. Los objetos consisten en un estado interno (oculto) más métodos para operar sobre ese estado. Como no se permite al programador acceder al estado directamente, se abren muchas posibilidades para poder compartir entre máquinas que no tienen una memoria física compartida.

Un sistema basado en objetos que da la ilusión de una memoria compartida en sistemas de multicomputadora es Orca (Bal, 1991; Bal *et al.*, 1992; Bal y Tanenbaum, 1988). Orca es un lenguaje de programación tradicional (basado en Modula 2) al cual se han añadido dos características nuevas: objetos y la capacidad para crear procesos nuevos. Un objeto Orca es un tipo de datos abstracto, análogo a un objeto en Java o un paquete en Ada. El objeto encapsula estructuras de datos internas y métodos escritos por el usuario, llamados **operaciones**. Los objetos son pasivos, es decir, no contienen enlaces a los que puedan enviarse mensajes. En vez de ello, los procesos acceden a los datos internos de un objeto invocando sus métodos.

Cada método Orca consiste en una lista de pares (guardia, bloque de enunciados). Un guardia (guard) es una expresión booleana que no contiene efectos secundarios, o el guardia vacío, que equivale al valor *true* (verdadero). Cuando se invoca una operación, todos sus guardias se evalúan en un orden no especificado. Si todos ellos son *false*, el proceso invocador espera hasta que uno de ellos se vuelve *true*. Cuando la evaluación de un guardia da *true*,

se ejecuta el bloque de enunciados que le siguen. En la figura 8-42 se muestra un objeto *pila* con dos operaciones, *meter* y *sacar*.

```
Object implementation pila;
   tope:integer;                          # memoria para la pila
   pila: array [integer 0..N-1] of integer;

   operation meter(elem: integer);        # función que no devuelve nada
   begin
      pila[tope]:= elem;                  # meter elem(ento) en la pila
      tope:= tope + 1;                    # incrementar apuntador de pila
   end;

   operation sacar( ): integer;           # función que devuelve un entero
   begin
      guard tope > 0 do                   # suspender si la pila está vacía
         tope:= tope –1;                  # decrementar apuntador de pila
         return pila[tope];               # devolver elemento del tope
      od;
   end;

begin
   tope:= 0;                              # inicialización
end;
```

Figura 8-42. Objeto de pila ORCA simplificado, con datos internos y dos operaciones.

Una vez definida una *pila*, es posible declarar variables de este tipo, como en

```
s, t: pila;
```

que crea dos objetos pila e inicializa la variable *tope* de cada uno con 0. Podemos meter la variable entera *k* en la pila *s* con el enunciado

```
s$push(k);
```

etcétera. La operación *sacar* tiene un guardia, por lo que un intento por sacar una variable de una pila vacía suspenderá al invocador hasta que otro proceso haya metido algo en la pila.

Orca cuenta con un enunciado **fork** para crear un proceso nuevo en un procesador especificado por el usuario. El nuevo proceso ejecuta el procedimiento nombrado en el enunciado **fork**. Es posible pasar parámetros, que podrían ser objetos, al nuevo proceso, y es así como se distribuyen los objetos entre las máquinas. Por ejemplo, el enunciado

```
for i in 1 .. n do fork foobar(s) on i; od;
```

genera un proceso nueve en cada una de las máquinas de la 1 hasta la *n*, ejecutando el programa *foobar* en cada una de ellas. Puesto que estos *n* procesos nuevos (y el padre) se ejecutan en paralelo, todos pueden meter elementos en la pila compartida *s*, y sacarlos de ella, como si todos se estuvieran ejecutando en un multiprocesador con memoria compartida. Correspon-

de al sistema de tiempo de ejecución mantener la ilusión de una memoria compartida que en realidad no existe.

Las operaciones con objetos compartidos son atómicas y secuencialmente consistentes. El sistema garantiza que si varios procesos realizan operaciones con el mismo objeto compartido casi al mismo tiempo, el sistema escogerá algún orden y todos los procesos verán el mismo orden.

Orca integra datos compartidos y sincronización en una forma que no está presente en los sistemas DSM basados en páginas. En los programas paralelos se requieren dos tipos de sincronización. El primero es la sincronización por exclusión mutua, para evitar que dos procesos ejecuten la misma región crítica al mismo tiempo. En Orca, cada operación con un objeto compartido es de hecho como una región crítica porque el sistema garantiza que el resultado final será el mismo que se obtendría si todas las regiones críticas se ejecutaran una por una (es decir, secuencialmente). En este sentido, un objeto Orca es como una forma distribuida de monitor (Hoare, 1975).

El otro tipo de sincronización es la sincronización por condición, en la que un proceso se bloquea hasta que se cumple una condición. En Orca, la sincronización por condición se efectúa con guardias. En el ejemplo de la figura 8-42, un proceso que trata de sacar un elemento de una pila vacía se suspende hasta que la pila deja de estar vacía.

El sistema de tiempo de ejecución Orca se encarga del copiado de objetos, migración, consistencia e invocación de operaciones. Cada objeto puede estar en uno de dos estados: copia única o repetido. Un objeto en estado de copia única existe en una sola máquina, por lo que todas las solicitudes que lo requieren se envían ahí. Un objeto repetido está presente en todas las máquinas que contienen un proceso que lo usa, lo cual facilita las operaciones de lectura (porque se pueden efectuar localmente) a expensas de hacer las actualizaciones más costosas. Cuando se ejecuta una operación que modifica un objeto repetido, primero debe obtener un número de secuencia de un proceso centralizado que los emite. Luego se envía un mensaje a cada máquina que tiene una copia del objeto, pidiéndole que ejecute la operación. Puesto que todas estas actualizaciones llevan números de secuencia, todas las máquinas realizan las operaciones en orden secuencial, lo que garantiza la consistencia secuencial.

Globe

Casi todos los sistemas DSM, Linda y Orca se ejecutan en sistemas locales, es decir, dentro de un solo edificio o campus universitario. Sin embargo, también es posible construir un sistema de memoria compartida en el nivel de aplicaciones en una multicomputadora distribuida en todo el mundo. En el sistema Globe, un objeto puede estar en el espacio de direcciones de varios procesos al mismo tiempo, posiblemente en continentes distintos (Kermarrec *et al.*, 1998; van Steen *et al.*, 1999). Para acceder a los datos de un objeto compartido, los procesos de usuario deben usar sus métodos, lo que permite a diferentes objetos tener diferentes estrategias de implementación. Por ejemplo, una opción es tener una sola copia de los datos que se solicita dinámicamente cuando se necesita (apropiado para datos que un solo dueño actualiza con frecuencia). Otra opción es tener todos los datos situados dentro de cada copia del objeto y enviar actualizaciones a cada copia con un protocolo de multidifusión confiable.

Lo que hace a Globe un tanto ambicioso es su meta de abarcar mil millones de usuarios y un billón de objetos (posiblemente móviles). Localizar objetos, administrarlos y controlar el cambio de escala son aspectos cruciales. Globe logra esto con la ayuda de un marco general en el que cada objeto puede tener su propia estrategia de copiado, estrategia de seguridad, etc. Esto evita el problema "unitalla" que se presenta en otros sistemas, sin perder la facilidad de programación que ofrece la memoria compartida.

Otros sistemas distribuidos de área extensa son Globus (Foster y Kesselman, 1998a; Foster y Kesselman, 1998b) y Legion (Grimshaw y Wulf, 1996; Grimshaw y Wulf, 1997), pero éstos no ofrecen la ilusión de memoria compartida como Globe.

8.5 RESUMEN

Las computadoras paralelas pueden dividirse en dos categorías principales: SIMD y MIMD. Las máquinas SIMD ejecutan una instrucción a la vez en paralelo con muchos conjuntos de datos. Estas máquinas incluyen los procesadores de arreglos y las computadoras vectoriales. Las máquinas MIMD, que dominan la computación en paralelo, ejecutan diferentes programas en diferentes máquinas. Las máquinas MIMD se pueden dividir en multiprocesadores, que comparten la memoria primaria, y multicomputadoras, que no lo hacen. Ambos tipos de sistemas consisten en CPU y módulos de memoria conectados por diversas redes de alta velocidad que transfieren paquetes de solicitud y respuesta entre las CPU y la memoria y entre las CPU. Se usan muchas topologías, que incluyen cuadrículas, toroides, anillos e hipercubos.

Para todos los multiprocesadores, uno de los aspectos clave es el modelo de consistencia de la memoria que se maneja. Entre los modelos más comunes están la consistencia secuencial, la consistencia de procesador, la consistencia débil y la consistencia de liberación. Se pueden construir multiprocesadores en los que se espía el bus (usando el protocolo MESI, por ejemplo). También son posibles diseños de barras cruzadas y redes de conmutación. Otras posibilidades son las máquinas NUMA y COMA basadas en directorios.

Las multicomputadoras se pueden dividir a grandes rasgos en MPP y COW, aunque la frontera es un tanto arbitraria. Los MPP son enormes sistemas comerciales, como el Cray T3E y el Intel/Sandia Option Red, que usan interconexiones de alta velocidad patentadas. En contraste, los COW se construyen con piezas comerciales, como Ethernet, ATM o Myrinet.

Las multicomputadoras a menudo se programan empleando un paquete de transferencia de mensajes como PVM o MPI. Ambos cuentan con llamadas de biblioteca para enviar y recibir mensajes, con muchas opciones, y se ejecutan encima de sistemas operativos existentes.

Una estrategia alternativa es usar memoria compartida en el nivel de aplicaciones, como un sistema DSM basado en páginas, el espacio de tuplas Linda o los objetos Orca o Globe. DSM simula una memoria compartida en el nivel de páginas, lo que lo hace similar a una máquina NUMA, excepto que las referencias remotas son mucho más lentas. Linda, Orca y Globe proporcionan al usuario la ilusión de un modelo de memoria compartida (restringido) empleando tuplas, objetos locales y objetos globales, respectivamente.

PROBLEMAS

1. Una mañana, la abeja reina de cierta colmena llama a todas sus abejas obreras y les dice que la tarea del día es recolectar néctar de caléndula. Entonces, las obreras vuelan en diferentes direcciones en busca de caléndulas. ¿Se trata de un sistema SIMD o MIMD?
2. Para cada una de las topologías de la figura 8-4, calcule el diámetro de la red.
3. Para cada una de las topologías de la figura 8-4, determine el grado de tolerancia ante fallos, que se define como el número máximo de enlaces que pueden perderse sin partir la red en dos.
4. Considere la topología de toroide doble de la figura 8-4(f) pero expandida a un tamaño de $k \times k$. ¿Qué diámetro tiene la red? Sugerencia: considere k par y k impar por separado.
5. Una red de interconexión tiene la forma de un cubo de $8 \times 8 \times 8$. Cada enlace tiene un ancho de banda dúplex de 1 GB/s. ¿Qué ancho de banda bisectriz tiene esta red?
6. En una red de interconexión con forma de cuadrícula rectangular de cuatro conmutadores de anchura por tres conmutadores de altura, los paquetes que parten de la esquina superior izquierda y se dirigen hacia la esquina inferior derecha pueden seguir cualquiera de varias rutas distintas. Numere la fila superior de conmutadores del 1 al 4, la siguiente del 5 al 8, y la de abajo del 9 al 12. Enumere todas las rutas en las que los paquetes sólo se muevan hacia la derecha o hacia abajo, partiendo del 1 y terminando en el 12.
7. La ley de Amdahl limita la aceleración que puede lograrse en una computadora paralela. Calcule, en función de f, la aceleración máxima posible a medida que el número de CPU se acerca al infinito. ¿Qué implicaciones tiene este límite para $f = 0.1$?
8. La figura 8-10 muestra cómo falla el aumento de escala con un bus pero tiene éxito con una cuadrícula. Suponiendo que cada bus o enlace tiene un ancho de banda b, calcule el ancho de banda medio por CPU para cada uno de los cuatro casos. Luego aumente la escala de cada sistema a 64 CPU y repita los cálculos. ¿Cuál es el límite a medida que el número de CPU se acerca a infinito?
9. La compañía de computadoras 2D tiene un producto que consiste en n computadoras con memoria disjunta dispuestas en una cuadrícula cuadrada. Cierta noche, al vicepresidente de topologías se le ocurre una idea para un producto nuevo: una cuadrícula tridimensional con los n procesadores dispuestos en un cubo perfecto (por ejemplo, esto es posible con $n = 4096$).
   a. ¿Qué efecto tiene este cambio sobre el retraso en el peor de los casos?
   b. ¿Qué efecto tiene este cambio sobre el ancho de banda total?
10. Cuando hablamos de los modelos de consistencia de la memoria, dijimos que un modelo de consistencia es una especie de contrato entre el software y la memoria. ¿Por qué se necesita semejante contrato?
11. Un procesador de vectores como el Cray-1 tiene unidades aritméticas con filas de procesamiento de cuatro etapas. Cada etapa tarda 1 ns. ¿Qué tiempo toma sumar dos vectores de 1024 elementos cada uno?
12. Considere un multiprocesador que usa un bus compartido. ¿Qué sucede si dos procesadores tratan de acceder a la memoria global en el mismo instante exactamente?
13. Suponga que por razones técnicas sólo es posible que una caché espía vigile líneas de direcciones, no líneas de datos. ¿Esto afectaría el protocolo de *escritura a través*?

14. Como modelo sencillo de sistema multiprocesador basado en bus sin cachés, suponga que una instrucción de cada cuatro hace una referencia a la memoria, y que una referencia a la memoria ocupa el bus durante todo un tiempo de instrucción. Si el bus está ocupado, la CPU solicitante se coloca en una cola FIFO. ¿Qué tanto más veloz será un sistema de 64 CPU que uno de una sola CPU?
15. El protocolo de coherencia de cachés MESI tiene cuatro estados. Otros protocolos de coherencia de cachés de escritura de vuelta sólo tienen tres estados. ¿Cuál de los cuatro estados MESI podría sacrificarse, y cuáles serían las consecuencias de cada opción? Si tuviera que escoger sólo tres estados, ¿cuáles escogería?
16. ¿Existen situaciones con el protocolo de coherencia de cachés MESI en las que una línea de caché está presente en la caché local pero para la cual de todos modos se necesita una transacción de bus? Explique.
17. Suponga que hay n CPU conectadas a un bus común. La probabilidad de que cualquier CPU trate de usar el bus en un ciclo dado es p. ¿Qué posibilidades hay de que
   a. El bus esté ocioso (0 solicitudes)?
   b. Se haya hecho exactamente una solicitud?
   c. Se haya hecho más de una solicitud?
18. Las CPU del Sun Enterprise 10000 operan a 333 MHz, pero los buses de espionaje operan a sólo 83.3 MHz. Con 64 CPU, es evidente que los buses nunca podrán manejar la carga. Sin embargo, la máquina funciona. Explique por qué.
19. En el texto calculamos que la capacidad del conmutador de barras cruzadas era suficiente para manejar 167 millones de espionajes por segundo cuando hay 16 tarjetas conectadas a él, aun tomando en cuenta el hecho de que la contención reduce el ancho de banda práctico al 60% del valor teórico. Un Enterprise 10000 pequeño podría tener sólo cuatro tarjetas (16 CPU). ¿Este sistema podría operar a toda velocidad?
20. Suponga que el alambre entre el conmutador 2A y el 3B de la red omega se rompe. ¿Quién queda aislado de quién?
21. Los puntos álgidos (localidades de memoria a las que se hace referencia muy a menudo) son obviamente un problema importante en las redes de conmutación multietapas. ¿También son un problema en los sistemas basados en bus?
22. Una red de conmutación omega conecta 4096 CPU RISC que tienen un tiempo de ciclo de 60 ns, con 4096 módulos de memoria infinitamente rápidos. Cada elemento de conmutación tiene un retraso de 5 ns. ¿Cuántas ranuras de retraso necesita una instrucción LOAD?
23. Considere una máquina que usa una red de conmutación omega como la que se muestra en la figura 8-28. Suponga que el programa y la pila para el procesador i se guardan en el módulo de memoria i. Proponga un cambio pequeño a la topología que mejore mucho el desempeño (la IBM RP3 y la BBN Butterfly usan esa topología modificada). ¿Qué desventajas tiene la nueva topología en comparación con el original?
24. En un multiprocesador NUMA, las referencias a la memoria local tardan 20 ns y las referencias remotas tardan 120 ns. Cierto programa efectúa un total de N referencias a la memoria durante su ejecución, de las cuales 1% son a una página P. Esa página inicialmente es remota, y se requieren C ns para copiarla localmente. ¿En qué condiciones deberá copiarse localmente la página si otros procesadores no hacen mucho uso de ella?

25. Un sistema DASH tiene b bytes de memoria divididos entre n cúmulos. Cada cúmulo tiene p procesadores. El tamaño de línea de caché es de c bytes. Dé una fórmula para la cantidad total de memoria dedicada a directorios (excluidos los dos bits de estado por entrada de directorio).

26. Considere un multiprocesador CC-NUMA como el de la figura 8-30 excepto que con 512 nodos de 8 MB cada uno. Si las líneas de caché son de 64 bytes, determine el porcentaje de gasto extra que representan los directorios. ¿Aumentar el número de nodos aumenta el gasto extra, reduce el gasto extra o no tiene ningún efecto?

27. ¿Cuál es la operación de peor caso (la que más tiempo consume) en SCI?

28. Un multiprocesador basado en SCI tiene 63 nodos, una línea de caché de 32 bytes y un espacio de direcciones total de 2^{32} bytes. La caché en cada nodo es de 1 MB. ¿Cuántos bytes se necesitan en cada directorio de ésta?

29. Proponga un cambio a la implementación NUMA-Q 2000 que permita tener 64 nodos en lugar de 63. ¿Por qué cree que Sequent haya escogido 63 en lugar de 64 como máximo?

30. En el texto se habló de tres variaciones de send: sincrónica, bloqueadora y no bloqueadora. Proponga un cuarto método que sea similar a la versión bloqueadora, pero que tenga propiedades un poco distintas. Cite una ventaja y una desventaja de su método en comparación con un send bloqueador.

31. Considere una multicomputadora que opera en una red con difusión por hardware, como Ethernet. ¿Por qué importa la proporción de operaciones de lectura (que no actualizan variables de estado internas) por cada operación de escritura (que sí actualiza variables de estado internas)?

32. Muchas de las cuestiones que se presentan en el diseño de multiprocesadores también se presentan en la memoria compartida en el nivel de aplicaciones. Entre ellas están la política de invalidar o actualizar. ¿Cuál política usa Orca?

9

LISTA DE LECTURAS Y BIBLIOGRAFÍA

En los ocho capítulos anteriores se trató un gran número de temas con distintos grados de detalle. El propósito de este capítulo es ayudar a los lectores interesados a profundizar en el estudio de la organización de las computadoras. La sección 9.1 contiene una lista de lecturas sugeridas organizadas por capítulo. La sección 9.2 es una bibliografía alfabética de todos los libros y artículos citados en este libro.

9.1 SUGERENCIAS PARA LECTURAS ADICIONALES

9.1.1 Introducción y obras generales

Hamacher *et al.*, *Computer Organization, 4a. ed.*

Libro de texto tradicional sobre organización de computadoras, CPU, memoria, E/S, aritmética y periféricos. Los ejemplos principales son el 68000 y la PowerPC.

Hayes, *Computer Architecture and Organization, 3a. ed.*

Otro texto tradicional sobre organización de computadoras, como Hamacher *et al.*, con orientación hacia el hardware. Los temas que se cubren incluyen la CPU, el camino de datos, microprogramación, filas de procesamiento, organización de la memoria y E/S.

Patterson y Hennessy, *Computer Organization and Design*

Con casi 1000 páginas, este libro monumental tiene mucho material sobre arquitectura de computadoras, sobre todo el diseño de CPU RISC. Se hace hincapié en cómo obtener un alto desempeño usando filas de procesamiento y otras técnicas.

Price, "A History of Calculating Machines"

Aunque las computadoras modernas iniciaron con Babbage en el siglo XIX, los seres humanos han estado calculando desde los albores de la civilización. Este fascinante artículo ilustrado sigue la historia del conteo, las matemáticas, los calendarios y el cómputo desde 3000 años A.C. hasta principios del siglo XX.

Slater, *Portraits in Silicon*

¿Por qué Dennis Ritchie no entregó su tesis de doctorado en Harvard? ¿Por qué Steve Jobs se convirtió en vegetariano? Las respuestas están en este fascinante libro que contiene biografías cortas de 34 personas que moldearon la industria de las computadoras, desde Charles Babbage hasta Donald Knuth.

Stallings, *Computer Organization and Architecture, 4a. ed.*

Texto general sobre arquitectura de computadoras. Algunos de los temas que se tratan en este libro también se cubren en el de Stallings.

Wilkes, "Computers Then and Now"

Historia personal de las computadoras desde 1946 hasta 1968 por un diseñador de computadoras pionero e inventor de la microprogramación, Maurice Wilkes. Él habla de las primeras batallas entre los "cadetes espaciales" que creían en la programación automática (antes de los compiladores FORTRAN) y los tradicionalistas, que preferían programar en octal.

9.1.2 Organización de sistemas de cómputo

Ng, "Advances in Disk Technology: Performance Issues"

La gente ha estado pronosticando el fin del disco magnético desde hace por lo menos 20 años. Hasta ahora, los discos siguen vivos. Y según este artículo, su tecnología está avanzando rápidamente, por lo que es probable que los tengamos entre nosotros durante varios años más.

Messmer, *The Indispensable PC Hardware Book, 3a. ed.*

Con 1384 páginas (divididas en 36 capítulos y 13 apéndices), este libro podría ser indispensable o no, pero ciertamente es grueso. Aquí encontrará prácticamente todas las intimidades de los procesadores 80x86, memorias, buses, chips de apoyo y periféricos. Si ha leído y digerido el libro de Norton y Goodman (vea en seguida) y quiere pasar al siguiente nivel de detalle técnico, comience aquí.

Norton y Goodman, *Inside the PC, 7a. ed.*

Casi todos los libros sobre hardware de PC están escritos para gente que estudió ingeniería eléctrica y son difíciles de leer por quienes están orientados hacia el software. Éste es distinto: explica el hardware de PC de una forma técnica pero muy accesible. Los temas incluyen la CPU, memoria, buses, discos, pantallas, dispositivos de E/S, PC portátiles, trabajo con redes y más. Un singular y valioso libro.

Pilgrim, *Build Your Own Pentium II PC*

Si es miembro fundador del club del destornillador y el cautín, y quiere construir su propia PC con trocitos de metal, este libro le dice cómo hacerlo. Sin embargo, aunque se conforme con comprar su PC en una tienda, este libro contiene abundante información acerca de los componentes de una PC, cómo funcionan y qué opciones hay.

9.1.3 El nivel de lógica digital

Floyd, *Digital Fundamentals, 6a. ed.*

Para lectores orientados hacia el hardware que quieren profundizar en su estudio del nivel de lógica digital, este enorme libro a cuatro colores, generosamente ilustrado, es una verdadera joya. Los capítulos cubren lógica combinatoria, dispositivos lógicos programables, flip-flops, registros de desplazamiento, memorias, interfaces y mucho más.

Katayama, "Trends in Semiconductor Memory"

Si bien las memorias no son tan rápidas como las CPU, la tecnología de memoria no está estancada. Están ocurriendo todo tipo de avances en tecnología DRAM. Este artículo reseña varios de ellos.

Mano y Kime, *Logic and Computer Design Fundamentals*

Aunque este libro no tiene el diseño elegante del de Floyd, también es una buena referencia para el nivel de lógica digital. La cobertura incluye circuitos combinatorios y secuenciales, registros, memorias, diseño de CPU y E/S.

Mazidi y Mazidi, *The 80x86 IBM PC and Compatible Computers, 2a. ed.*

Para lectores interesados en entender todos los chips que hay dentro de una PC, este libro tiene capítulos enteros sobre los principales chips, así como abundante información acerca del hardware de la IBM PC y la programación en lenguaje ensamblador.

McKee *et al.*, "Smarter Memory: Improving Bandwidth for Streamed References"

En comparación con las CPU, las memorias se han estado volviendo más lentas desde hace décadas. En este artículo se examinan varias cuestiones relacionadas con el desempeño de las memorias, y se analiza una posible solución.

Nelson *et al.*, *Digital Logic and Circuit Analysis and Design*

Otro libro muy completo sobre lógica digital; cubre circuitos tanto combinatorios como secuenciales con gran detalle.

Triebel, *The 80386, 80486, and Pentium Processor*

Es un poco difícil clasificar este libro porque se ocupa de hardware, software e interfaces. Puesto que el autor trabaja para Intel, llamémoslo un libro de hardware. Aquí se dice todo sobre los procesadores, memorias, dispositivos de E/S e interfaces de los chips 80x86, pero también se explica cómo programarlos en lenguaje ensamblador. Aunque sólo tiene 915 páginas, contiene casi tanto material como el libro de Messmer porque las páginas son más grandes.

9.1.4 El nivel de microarquitectura

Handy, *The Cache Memory Book, 2a. ed.*

El diseño de cachés es lo bastante importante como para que ya haya libros enteros sobre el tema. En éste se tratan las cachés lógicas *versus* las físicas, el tamaño de línea, las políticas de escritura a través y las de escritura de vuelta, las cachés unificadas *versus* las divididas, y también cuestiones de software. Además, hay un capítulo sobre coherencia de cachés en multiprocesadores.

Johnson, *Superscalar Microprocessor Design*

Para lectores interesados en los detalles del diseño de CPU superescalares, este libro es el mejor punto de partida: cubre la obtención y decodificación de instrucciones, la emisión de instrucciones en desorden, el cambio de nombre de registros, las estaciones de reservación, la predicción de ramas, y más.

Normoyle *et al.*, "UltraSPARC IIi: Expanding the Boundaries of a System on a Chip"

El UltraSPARC IIi es una versión para el bus PCI del UltraSPARC II. En este artículo sus diseñadores explican su funcionamiento interno.

McGhan y O'Connor, "picoJava: A Direct Execution Engine for Java Bytecode"

Si desea una breve introducción a la microarquitectura del diseño picoJava en hardware de Sun (y por tanto el chip microJava 701), este artículo es un buen punto de partida. Se presenta un diagrama de bloques, se analiza el conducto y se explica el funcionamiento de varias optimaciones.

Shriver y Smith, "The Anatomy of a High-Performance Microprocessor"

Si desea un estudio detallado de un chip de CPU moderno en el nivel de microarquitectura, le recomendamos este libro, que examina el chip AMD K6, un clon del Pentium, con gran detalle, haciendo hincapié en la fila de procesamiento, la planificación de instrucciones y optimación del desempeño.

Sima, "Superscalar Instruction Issue"

La emisión de instrucciones superescalares es cada vez más importante en las CPU modernas. Mencionamos algunos de los aspectos en este capítulo, como el cambio de nombres y la ejecución especulativa. En este artículo se examinan éstas y varias otras cuestiones.

Wilson, "Challenges and Trends in Processor Design"

¿El diseño de procesadores se ha estancado? De ninguna manera. Seis importantes arquitectos de CPU de Sun, Cyrix, Motorola, Mips, Intel y Digital nos dicen hacia dónde se dirige el diseño de CPU en los próximos años. Va a ser divertido leer esto en 2008 (pero también vale la pena leerlo ahora).

9.1.5 El nivel de arquitectura del conjunto de instrucciones

Antonakos, *The Pentium Microprocessor*

Los primeros nueve capítulos de este libro explican cómo programar el Pentium en lenguaje ensamblador. Los dos últimos se ocupan del hardware del Pentium. Se proporcionan muchos fragmentos de código y se trata exhaustivamente el BIOS.

Paul, *SPARC Architecture, Assembly Language, Programming, and C*

Maravilla de maravillas: un libro acerca de la programación en lenguaje ensamblador que no se refiere a la línea Intel 80x86. En vez de ello, se ocupa del SPARC y de cómo programarlo.

Weaver y Germond, *The SPARC Architecture Manual*

Con la creciente internacionalización de la industria de las computadoras, los estándares se vuelven cada vez más importantes, y es importante familiarizarse con ellos. Ésta es la definición de la versión 9 de SPARC y da una buena idea de cómo es un estándar, además de proporcionar abundante información sobre el funcionamiento de los SPARC de 64 bits.

9.1.6 El nivel de máquina del sistema operativo

Hart, "Win32 System Programming"

A diferencia de casi todos los demás libros sobre Windows, éste no se concentra en la interfaz gráfica con el usuario (de hecho, ni siquiera la trata); en vez de ello, se concentra en las llamadas al sistema que Windows ofrece y en la forma de utilizarlas para acceder a archivos, administrar la memoria, administrar procesos, comunicación entre procesos, líneas, E/S y otros temas.

Jacob y Mudge, "Virtual Memory: Issues of Implementation"

Si busca una buena introducción moderna a la memoria virtual, aquí está. Se explican diversas estructuras de tabla de páginas y TLB y se ilustran las ideas utilizando los procesadores MIPS, PowerPC y Pentium.

Korn, "Porting UNIX to Windows NT"

Aunque a primera vista podría pensarse que dado el gran número de llamadas al sistema que tiene NT sería fácil trasladar a él programas en UNIX, los intentos por hacerlo muestran que es mucho más difícil de lo que parece. En este artículo, alguien que lo ha intentado explica por qué.

McKusick *et al.*, *Design and Implementation of the 4.4 BSD Operating System*

A diferencia de la mayor parte de los libros sobre UNIX, éste inicia con una fotografía de los cuatro autores en una Conferencia USENIX; tres de ellos escribieron una buena parte del 4.4 BSD y están eminentemente calificados para explicar su funcionamiento interno. El libro cubre las llamadas al sistema, procesos, E/S, y tiene una sección excelente sobre trabajo con redes.

Ritchie y Thompson, "The UNIX Time-Sharing System"

Éste es el artículo original publicado acerca de UNIX. Todavía es muy provechoso leerlo. De esta pequeña semilla creció un gran sistema operativo.

Solomon, *Inside Windows NT, 2a. ed.*

Si quiere saber cómo NT funciona internamente, le recomendamos este libro. Se explica la arquitectura del sistema, sus mecanismos, procesos, líneas, gestión de memoria, seguridad, E/S, cachés y sistema de archivos, entre otros temas.

Tanenbaum y Woodhull, *Operating Systems: Design and Implementation, 2a. ed.*

A diferencia de la mayor parte de los libros sobre sistemas operativos, que sólo se ocupan de la teoría, éste cubre toda la teoría pertinente y la ilustra analizando el código real de un sistema operativo tipo UNIX, MINIX, que se ejecuta en la IBM PC y otras computadoras. El código fuente, con abundantes comentarios, se presenta en un apéndice.

9.1.7 El nivel de lenguaje ensamblador

Irvine, *Assembly Language for Intel-Based Computers, 3a. ed.*

La programación de las CPU Intel en lenguaje ensamblador es el tema de este libro. También se cubren la programación de E/S, macros, archivos, enlace, interrupciones y muchos otros temas relacionados.

Saloman, *Assemblers and Loaders*

Todo lo que quiera saber acerca del funcionamiento de los ensambladores de una y dos pasadas, así como el de los enlazadores y cargadores. También se cubren macros y ensamblado condicional.

9.1.8 Arquitecturas de computadoras paralelas

Adve y Gharachorloo, "Shared Memory Consistency Models: A Tutorial"

Muchas computadoras modernas, sobre todo multiprocesadores, manejan un modelo de memoria más débil que la consistencia secuencial. Este tutorial analiza varios modelos y

explica cómo funcionan; también plantea y refuta numerosos mitos acerca de la memoria con consistencia débil.

Almasi y Gottlieb, *Highly Parallel Computing, 2a. ed.*

Si desea un panorama general de la computación paralela, incluidas redes de interconexión, arquitectura, compiladores, modelos y aplicaciones, este libro es muy recomendable. Su contenido está balanceado entre temas de hardware, software y aplicaciones.

Hill, "Multiprocessors Shoud Support Simple Memory-Consistency Models"

La semántica de memoria relajada es un tema álgido del diseño de memorias para multiprocesadores, y causa grandes controversias. Los modelos más débiles permiten ciertas optimaciones en hardware, como hacer referencias a la memoria en desorden, pero hacen más difícil la programación. En este artículo el autor trata muchas cuestiones importantes para la consistencia de la memoria y luego concluye que la memoria relajada no vale la pena.

Hwang y Xu, *Scalable Parallel Computing*

Al tratar tanto el hardware como el software, los autores logran presentar una reseña completa y amena de la computación en paralelo. Los temas incluyen multiprocesadores UMA y NUMA, MPP y COW, transferencia de mensajes y programación paralela.

Pfister, *In Search of Clusters, 2a. ed.*

Aunque la definición de cúmulo no aparece sino hasta la página 72 (un grupo de computadoras completas que trabajan juntas), al parecer incluye todos los sistemas de multiprocesador y multicomputadora usuales. Se estudian con detalle su hardware, software, desempeño y disponibilidad. Sin embargo, el lector queda advertido de que si bien el monísimo estilo de redacción del autor al principio resulta divertido, después de 500 páginas es más bien empalagoso.

Snir *et al.*, *MPI: The Complete Reference Manual*

El título lo dice todo. Si quiere aprender a programar en MPI, aquí está lo que busca. El libro cubre la comunicación punto a punto y colectiva, comunicadores, gestión de entornos, perfiles y más.

Stenstrom *et al.*, "Trends in Shared Memory Multiprocessing"

Aunque muchos creen que los multiprocesadores con memoria compartida son supercomputadoras para realizar cálculos científicos masivos, en realidad ésa es sólo una fracción diminuta de su mercado. En este artículo los autores explican cuál es el verdadero mercado de estas máquinas, y qué implicaciones tiene para su arquitectura.

9.1.9 Números binarios y de punto flotante

Cody, "Analysis of Proposals for the Floating-Point Standard"

Hace algunos años, IEEE diseñó una arquitectura de punto flotante que se ha convertido en el estándar *de facto* para todos los chips de CPU modernos. Cody explica las diversas cuestiones, propuestas y controversias que surgieron durante el proceso de estandarización.

Garner, "Number Systems and Arithmetic"

Tutorial sobre conceptos avanzados de aritmética binaria, incluida la propagación de acarreo, sistemas de numeración redundantes, sistemas de numeración de residuo, y multiplicación y división no estándar. Muy recomendable para quien cree que ya sabe todo lo habido y por haber sobre aritmética en la primaria.

IEEE, *Proc. of the n-th Symposium on Computer Arithmetic*

A pesar de lo que comúnmente se cree, la aritmética es un área de investigación activa en la que se escriben muchos trabajos por y para especialistas en aritmética. En esta serie de simposios se presentan avances sobre suma y multiplicación de alta velocidad, hardware VLSI para aritmética, coprocesadores, tolerancia ante fallos y redondeo, entre otros temas.

Knuth, *Seminumerical Algorithms, 3a. ed.*

Abundante material sobre sistemas de numeración posicionales, aritmética de punto flotante, aritmética de múltiple precisión y números aleatorios. Este material requiere y merece un estudio cuidadoso.

Wilson, "Floating-Point Survival Kit"

Una bonita introducción a los números de punto flotante y sus estándares para gente que cree que el mundo acaba en 65,535. También se tratan algunas pruebas estándar para desempeño de punto flotante, como *Linpack.*

9.2 BIBLIOGRAFÍA EN ORDEN ALFABÉTICO

ADAMS, G.B. III, AGRAWAL, D.P., and SIEGEL, H.J.: "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *IEEE Computer Magazine*, vol. 20, pp. 14-27, June 1987.

ADVE, S.V., and CHARACHORLOO, K.: "Shared Memory Consistency Models: A Tutorial," *IEEE Computer Magazine*, vol. 29, pp. 66-76, Dec. 1996.

ADVE, S.V., and HILL, M.: "Weak Ordering: A New Definition," *Proc. 17th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 2-14, 1990.

AGERWALA, T., and COCKE, J.: "High Performance Reduced Instruction Set Processors," IBM T.J. Watson Research Center Technical Report RC12434, 1987.

ALMASI, G.S., and GOTTLIEB, A.: *Highly Parallel Computing, 2nd ed.* Redwood City, CA: Benjamin/Cummings, 1994.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., ZWAENEPOEL, W.: "TreadMarks: Shared Memory Computing on a Network of Workstations," *IEEE Computer Magazine*, vol. 29, pp. 18-28, Feb. 1996.

ANDERSON, D.: *Universal Serial Bus System Architecture*, Reading, MA: Addison-Wesley, 1997.

ANDERSON, T.E., CULLER, D.E., PATTERSON, D.A., and the NOW team: "A Case for NOW (Networks of Workstations)," *IEEE Micro Magazine*, vol. 15, pp. 54-64, Feb. 1995.

ANTONAKOS, J.L.: *The Pentium Microprocessor*, Upper Saddle River, NJ: Prentice Hall, 1997.

AUGUST, D.I., CONNORS, D.A., MSHLKE, S.A., SIAS, J.W., CROZIER, K.M., CHENG, B.-C., EATON, P.R., OLANIRAN, Q.B., and HWU, W.-M.: "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 227-237, 1998.

BAL, H.E.: *Programming Distributed Systems*, Hemel Hempstead, England: Prentice Hall Int'l., 1991.

BAL, H.E., BHOEDJANG, R., HOFMAN, R, JACOBS, C., LANGENDOEN, K., RUHL, T., and KAASHOEK, M.F.: "Performance Evaluation of the Orca Shared Object System," *ACM Trans. on Computer Systems*, vol. 16, pp. 1-40, Feb. 1998.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S.: "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Engineering*, vol. 18, pp. 190-205, March 1992.

BAL, H.E., and TANENBAUM, A.S.: "Distributed Programming with Shared Data," *Proc. 1988 Int'l. Conf. on Computer Languages*, IEEE, pp. 82-91, 1988.

BHUYAN, L.N., YANG, Q., and AGRAWAL, D.P.: "Performance of Multiprocessor Interconnection Networks," *IEEE Computer Magazine*, vol. 22, pp. 25-37, Feb. 1989.

BJORNSON, R.D.: "Linda on Distributed Memory Multiprocessors," Ph.D. Thesis, Yale Univ., 1993.

BLUMRICH, M.A., DUBNICKI, C., FELTEN, E.W., LI, K., and MESARINA, M.R. : "Virtual-Memory Mapped Network Interfaces," *IEEE Micro Magazine*, vol. 15, pp. 21-28, Feb. 1995.

BODEN, N.J., COHEN, D., FELDERMAN, R.E., KULAWIK, A.E., SEITZ, C.L., SEIZOVIC, J.N., and SU, W.-K.: "Myrinet: A Gigabit per second Local Area Network," *IEEE Micro Magazine*, vol. 15, pp. 29-36, Feb. 1995.

BOUKNIGHT, W.J., DENENBERG, S.A., MCINTYRE, D.E., RANDALL, J.M., SAMEH, A.H., AND SLOTNICK, D.L.: "The Illiac IV System," *Proc. IEEE*, pp. 369-388, April 1972.

BURKHARDT, H., FRANK, S., KNOBE, B., and ROTHNIE, J.: "Overview of the KSR-1 Computer System," Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.

CARRIERO, N., and GELERNTER, D.: "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems*, vol. 4, pp. 110-129, May 1986.

CARRIERO, N., and GELERNTER, D.: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444-458, April 1989.

CHARLESWORTH, A.: "Starfire: Extending the SMP Envelope," *IEEE Micro Magazine*, vol. 18, 39-49, Jan./Feb. 1998.

CHARLESWORTH, A., PHELPS, A., WILLIAMS, R., and GILBERT, G.: "Gigaplane-XB: Extending the Ultra Enterprise Family," *Proc. Hot Interconnects V*, IEEE, 1998.

CODY, W.J.: "Analysis of Proposals for the Floating-Point Standard," *IEEE Computer Magazine*, vol. 14, pp. 63-68, Mar. 1981.

COHEN, D.: "On Holy Wars and a Plea for Peace," *IEEE Computer Magazine*, vol. 14, pp. 48-54, Oct. 1981.

CORBATO, F.J.: "PL/1 as a Tool for System Programming," *Datamation*, vol. 15, pp. 68-76, May 1969.

CORBATO, F.J., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. FJCC*, pp. 185-196, 1965.

DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, May 1968.

DIJKSTRA, E.W.: "GOTO Statement Considered Harmful," *Commun. of the ACM*, vol. 11, pp. 147-148, Mar. 1968a.

DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys (ed.), New York: Academic Press, 1968b.

DRIESEN, K., and HOLZLE, URS: "Accurate Indirect Branch Prediction," *Proc. 25th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 167-177, 1998.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 434-442, 1986.

DULONG, C.: "The IA-64 Architecture at Work," *IEEE Computer Magazine*, vol. 31, pp. 24-32, July 1998.

FAGGIN, F., HOFF, M.E., Jr., MAZOR, S., and SHIMA, M.: "The History of the 4004," *IEEE Micro Magazine*, vol. 16, pp. 10-20, Dec. 1996.

FALSAFI, B., and WOOD, D.A.: "Reactive NUMA: A Design Unifying S-COMA and CC-NUMA," *Proc. 25th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 229-240, 1997.

FISHER, J.A., and FREUDENBERGER, S.M.: "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. 5th Int'l. Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85-95, 1992.

FLOYD, T.L.: *Digital Fundamentals, 6th ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.

FLYNN, M.J.: "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, vol. C-21, pp. 948-960, Sept. 1972.

FOSTER, I., and KESSELMAN, C.: "Globus: A Metacomputing Infrastructure Toolkit," *Int'l. J. of Supercomputer Applications*, vol. 11, pp. 115-128, 1998a.

FOSTER, I., and KESSELMAN, C.: "The Globus Project: A Status Report," *IPPS/SPDP '98 Heterogeneous Computing Workshop*, IEEE, pp. 4-18, 1998b.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Oct. 1961.

GAJSKI, D.D., and PIER, K.-K.: "Essential Issues in Multiprocessor Systems," *IEEE Computer Magazine*, vol. 18, pp. 9-27, June 1985.

GARNER, H.L.: "Number Systems and Arithmetic," in *Advances in Computers, vol. 6*, F. Alt and M. Rubinoff (eds.), New York: Academic Press, 1965, pp. 131-194.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., and SUNDERRAM, V.: *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*, Cambridge, MA: M.I.T. Press, 1994.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., and HENNESSY, J.: "Memory Consistency and Event Ordering in Salable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 15-26, 1990.

GOODMAN, J.R.: "Using Cache Memory to Reduce Processor Memory Traffic," *Proc. 10th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 124-131, 1983.

GOODMAN, J.R.: "Cache Consistency and Sequential Consistency," Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GRAHAM, R.: "Use of High Level Languages for System Programming," Project MAC Report TM-13, Project MAC, MIT, Sept. 1970.

GRIMSHAW, A.S., and WULF, W.: "Legion: A View from 50,000 Feet," *Proc. Fifth Int'l. Symp. on High-Performance Distributed Computing*, IEEE, pp. 89-99, Aug. 1996.

GRIMSHAW, A.S., and WULF, W.: "The Legion Vision of a Worldwide Virtual Computer," *Commun. of the ACM*, vol. 40, pp. 39-45, Jan. 1997.

GROPP, W., LUSK, E, and SKJELLUM, A: "Using MPI: Portable Parallel Programming with the Message Passing Interface," Cambridge, MA: M.I.T. Press, 1994.

HAGERSTEN, E., LANDIN, A., HARIDI, S.: "DDM—A Cache-Only Memory Architecture," *IEEE Computer Magazine*, vol. 25, pp. 44-54, Sept. 1992.

HAMACHER, V.V., VRANESIC, Z.G., and ZAKY, S.G.: *Computer Organization, 4th ed.*, New York: McGraw-Hill, 1996.

HAMMING, R.W.: "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, vol. 29, pp. 147-160, April 1950.

HANDY, J.: *The Cache Memory Book, 2nd ed.*, Orlando, FL: Academic Press, 1998.

HART, J.M.: *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.

HAYES, J.P.: *Computer Architecture and Organization, 3rd ed.*, New York: McGraw-Hill, 1998.

HENNESSY, J.L.: "VLSI Processor Architecture," *IEEE Trans. on Computers*, vol. C-33, pp. 1221-1246, Dec. 1984.

HILL, M.: "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer Magazine*, vol. 31, pp. 28-34, Aug. 1998.

HOARE, C.A.R.: "Monitors, An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549-557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.

HWANG, K., and XU, Z.: *Scalable Parallel Computing*, New York: McGraw-Hill, 1998.

HWU, W.-M.: "Introduction to Predicated Execution," *IEEE Computer Magazine*, vol. 31, pp. 49-50, Jan. 1998.

IRVINE, K: *Assembly Language for Intel-Based Computers, 3rd ed.*, Englewood Cliffs, NJ: Prentice Hall, 1999.

JACOB, B., and MUDGE, T.: "Virtual Memory: Issues of Implementation," *IEEE Computer Magazine*, vol. 31, pp. 33-43, June 1998a.

JACOB, B., and MUDGE, T.: "Virtual Memory in Contemporary Microprocessors," *IEEE Micro Magazine*, vol. 18, pp. 60-75, July/Aug. 1998b.

JOE, T., and HENNESSY, J.L.: "Evaluating the Memory Overhead Required for COMA Architectures," *Proc. 21th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 82-93, 1994.

JOHNSON, K.L., KAASHOEK, M.F., and WALLACH, D.A.: "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213-228, 1995.

JOHNSON, M.: *Superscalar Microprocessor Design*, Englewood Cliffs, NJ: Prentice Hall, 1991..

JUAN, T., SANJEEVAN, S., and NAVARRO, J.J.: "Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch prediction," *Proc. 25th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 155-166, 1998.

KATAYAMA, Y.: "Trends in Semiconductor Memories," *IEEE Micro Magazine*, pp. 10-17, Nov./Dec. 1997.

KERMARREC., A.-M., KUZ, I., VAN STEEN, M., and TANENBAUM, A.S.: "A Framework for Consistent Replicated Web Objects," *Proc. 18th Int'l. Conf. on Distr. Computing Syst.*, IEEE, pp. 276-284, 1998.

KNUTH, D.E.: "An Empirical Study of FORTRAN Programs," *Software—Practice & Experience*, vol. 1, pp. 105-133, 1971.

KNUTH, D.E.: *The Art of Computer Programming: Fundamental Algorithms, 3rd ed.*, Reading, MA: Addison-Wesley, 1997.

KNUTH, D.E.: *The Art of Computer Programming: Seminumerical Algorithms, 3rd ed.*, Reading, MA: Addison-Wesley, 1998.

KONTOTHANASSIS, L., HUNT, G., STETS, R., HARDAVELLAS, N., CIERNIAD, M., PARTHASARATHY, S., MEIRA, W., DWARKADAS, S., and SCOTT, M.: *VM-Based Shared Memory on Low Latency Remote Memory Access Networks, Proc. 24th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 157-169, 1997.

KORN, D.: "Porting UNIX to Windows NT," *Proc. Winter 1997 USENIX Conf.*, pp. 43-57, 1997.

KUMAR, V.P., and REDDY, S.M.: "Augmented Shuffle-Exchange Multistage Interconnection Networks," *IEEE Computer Magazine*, vol. 20, pp. 30-40, June 1987.

LAMPORT, L.: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol. C-28, pp. 690-691, Sept. 1979.

LaROWE, R.P., and ELLIS, C.S.: "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 319-363, Nov. 1991.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., and LAM, M.: "The Stanford Dash Multiprocessor," *IEEE Computer Magazine*, vol. 25, pp. 63-79, March 1992.

LI, K.: "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. 1988 Int'l. Conf. on Parallel Proc. (Vol. II)*, IEEE, pp. 94-101, 1988.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *Proc. 5th Ann. ACM Symp. on Prin. of Distr. Computing*, ACM, pp. 229-239, 1986.

LINDHOLM, T., and YELLIN, F.: *The Java Virtual Machine Specification*, Reading, MA: Addison-Wesley, 1997.

LOSHIN, D.: *High Performance Computing Demystified*, Cambridge, MA: AP Prof., 1994.

LU, H., COX, A.L., DWARKADAS, S., RAJAMONY, R., and ZWAENEPOEL, W.: "Software Distributed Shared Memory Support for Irregular Applications," *Proc. 6th Conf. on Prin. and Practice of Parallel Progr.*, pp. 48-56, June 1997.

LUKASIEWICZ, J.: *Aristotle's Syllogistic, 2nd ed.*, Oxford: Oxford University Press, 1958.

MANO, M.M., and KIME, C.R.: *Logic and Computer Design Fundamentals*, Upper Saddle River, NJ: Prentice Hall, 1997.

MARTIN, R.P., VAHDAT, A.M., CULLER, D.E., and ANDERSON, T.E.: "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 85-97, 1997.

MAZIDI, M.A., and MAZIDI, J.G.: *The 80x86 IBM PC and Compatible Computers, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1998.

McGHAN, H. and O'CONNOR, J.M.: "picoJava: A Direct Execution Engine for Java Bytecode," *IEEE Computer Magazine*, vol 31., Oct. 1998.

McKEE, S.A., KLENKE, R.H., WRIGHT, K.L., WULF, W.A., SALINAS, M.H., AYLOR, J.H., and BATSON, A.P.: "Smarter Memory: Improving Bandwidth for Streamed References," *IEEE Computer Magazine*, vol. 31, pp. 54-63, July 1998.

McKUSICK, M.K., BOSTIC, K., KARELS, M., and QUARTERMAN, J.S.: "The Design and Implementation of the 4.4 BSD Operating System," Reading, MA: Addison-Wesley, 1996.

McKUSICK, M.K., JOY, W.N., LEFFLER, S.J., AND FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Aug. 1984.

MESSMER, H.-P.: *The Indispensible PC Hardware Book, 3rd ed.*, Reading, MA: Addison-Wesley, 1997.

MORGAN, C.: *Portraits in Computing*, New York: ACM Press, 1997.

MORIN, C., GEFFLAUT, A., BANATRE, M., and KERMARREC, A.-M.: "COMA: An Opportunity for Building a Fault-Tolerant Scalable Shared Memory Multiprocessor," *Proc. 24th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 65-65, 1996.

MOUDGILL, M., and VASSILIADIS, S.: "Precise Interrupts," *IEEE Micro Magazine*, vol. 16, pp. 58-67, Feb. 1996.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365-368, 1984.

NELSON, V.P., NAGLE, H.T., CARROLL, B.D., and IRWIN, J.D.: *Digital Logic and Circuit Analysis and Design*, Englewood Cliffs, NJ: Prentice Hall, 1995.

NG, S.W.: "Advances in Disk Technology: Performance Issues," *IEEE Computer Magazine*, vol. 31, pp. 75-81, May 1998.

NORMOYLE, K.B., CSOPPENSZKY, M.A., TZENG, A., JOHNSON, T.P., FURMAN, C.D., and MOSTOUFI, J.: "UltraSPARC IIi: Expanding the Boundaries of a System on a Chip," *IEEE Micro Magazine*, vol. 18, pp. 14-24, March/April 1998.

NORTON, P., and GOODMAN, J.: *Inside the PC, 7th ed.*, Indianapolis, IN: Sams, 1997.

O'CONNOR, J.M., and TREMBLAY, M.: "PicoJava-I: The Java Virtual Machine in Hardware," *IEEE Micro Magazine*, vol. 17, pp. 45-53, March/April 1997.

ORGANICK, E.: *The MULTICS System*, Cambridge, MA: M.I.T. Press, 1972.

PAKIN, S., KARAMCHETI, V., and CHIEN, A.A.: "Fast Messages (FM): Efficient, Portable Communication for Workstation Cluster and Massively-Parallel Processors," *IEEE Concurrency*, vol. 5, pp. 60-73, April-June 1997.

PAN, S.-T., SO, K., and RAHMEH, J.T.: "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Int'l. Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 76-84, Oct. 1992.

PAPAMARCOS, M., and PATEL, J.: "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. 11th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 348-354, 1984.

PATTERSON, D.A.: "Reduced Instruction Set Computers," *Commun. of the ACM*, vol. 28, pp. 8-21, Jan. 1985.

PATTERSON, D.A., GIBSON, G., and KATZ, R.: "A case for redundant arrays of inexpensive disks (RAID)," *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, ACM, pp. 109-166,1988.

PATTERSON, D.A., and HENNESSY, J.L.: *Computer Organization and Design, 2nd ed.*, San Francisco, CA: Morgan Kaufmann, 1998.

PATTERSON, D.A., and SEQUIN, C.H.: "A VLSI RISC," *IEEE Computer Magazine*, vol. 15, pp. 8-22, Sept. 1982.

PAUL, R.P.: *SPARC Architecture, Assembly Language, Programming, and C*, Englewood Cliffs, NJ: Prentice Hall, 1994.

PFISTER, G.F.: *In Search of Clusters, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1998.

PILGRIM, A.: *Build Your Own Pentium II PC*, New York: McGraw-Hill, 1998.

POUNTAIN, D.: "Pentium: More RISC than CISC," *Byte*, vol. 18, pp. 195-204, Sept. 1993.

PRICE, D.: "A History of Calculating Machines," *IEEE Micro Magazine*, vol. 4, pp. 22-52, Feb. 1984.

RADIN, G.: "The 801 Minicomputer," *Computer Arch. News*, vol. 10, pp. 39-47, March 1982.

RITCHIE, D.M., and THOMPSON, K.: "The UNIX Time-Sharing System," *Commun. of the ACM*, vol. 17, pp. 365-375, July 1974.

ROSENBLUM, M., and OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1-15, 1991.

SALOMAN, D.: *Assemblers and Loaders*, Englewood Cliffs, NJ: Prentice Hall, 1993.

SAULSBURY, A., WILKINSON, T., CARGER, J., and LANDIN, A.: "An Argument for Simple COMA," *Proc. of First IEEE Symp. on High-Performance Comp. Arch.*, IEEE, pp. 276-285, 1995.

SCALES, D.J., GHARACHORLOO, K., and THEKKATH, C.A.: "Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proc. 7th Int'l. Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM, pp. 174-185, 1996.

SECHREST, S., LEE, C.-C., and MUDGE, T.: "Correlation and Aliasing in Dynamic Branch Predictors," *Proc. 23th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 22-32, 1996.

SELTZER, M., BOSTIC, K., McKUSICK, M.K., and STAELIN, C.: "An Implementation of a Log-Structured File System for UNIX," *Proc. Winter 1993 USENIX Technical Conf.*, pp. 307-326, 1993.

SHANLEY, T., and ANDERSON, D.: *ISA System Architecture*, Reading, MA: Addison-Wesley, 1995a.

SHANLEY, T., and ANDERSON, D.: *PCI System Architecture, 3rd ed.*, Reading, MA: Addison-Wesley, 1995b.

SHRIVER, B., and SMITH, B.: *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*, Los Alamitos, CA: IEEE Computer Society, 1998.

SIMA, D: "Superscalar Instruction Issue," *IEEE Micro Magazine*, vol. 17, pp. 28-39, Sept./Oct 1997.

SIMA, D., FOUNTAIN, T., KACSUK, P.: *Advanced Computer Architectures: A Design Space Approach*, Reading, MA: Addison-Wesley, 1997.

SLATER, R.: *Portraits in Silicon*, Cambridge, MA: M.I.T. Press, 1987.

SMITH, A.J.: "Cache Memories," *Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.

SOLARI, E.: *ISA & EISA Theory and Operation*, San Diego, CA: Annabooks, 1993.

SOLARI, E., and WILLSE, G.: *PCI Hardware and Software Architecture and Design, 4th ed.*, San Diego, CA: Annabooks, 1998.

SOLOMON, D.A.: *Inside Windows NT, 2nd ed.*, Redmond, WA: Microsoft Press, 1998.

SPRANGLE, E., CHAPPELL, R.S., ALSUP, M., PATT, Y.N.: "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," *Proc. 24th Ann. Int'l. Symp. on Computer Arch.*, ACM, pp. 284-291, 1997.

STALLINGS, W.: *Computer Organization and Architecture, 4th ed.*, Upper Saddle River, NJ: Prentice Hall, 1996.

STENSTROM, P., HAGERSTEN, E., LILJA, D.J., MARTONOSI, M., and VENUGOPAL, M.: "Trends in Shared Memory Multiprocessing," *IEEE Computer Magazine*, vol. 30, pp. 44-50, Dec. 1997.

STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., and SCOTT, M.: "CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170-183, 1997.

SUNDERRAM, VB.: "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315-339, Dec. 1990.

SWAN, R.J., FULLER, S.H., and SIEWIOREK, D.P.: "Cm*—A Modular Multiprocessor," *Proc. NCC*, pp. 645-655, 1977.

TAN, W.M.: *Developing USB PC Peripherals*, San Diego, CA: Annabooks, 1997.

TANENBAUM, A.S.: "Implications of Structured Programming for Machine Architecture," *Commun. of the ACM*, vol. 21, pp. 237-246, Mar. 1978.

TANENBAUM, A.S, and WOODHULL, A.W.: *Operating Systems: Design and Implementation, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.

THOMPSON, K.: "UNIX Implementation," *Bell Syst. Tech. J.*, vol. 57, pp. 1931-1946, July-Aug. 1978.

TRELEAVEN, P.: "Control-Driven, Data-Driven, and Demand-Driven Computer Architecture," *Parallel Computing*, vol. 2, 1985.

TREMBLAY, M. and O'CONNOR, J.M.: "UltraSPARC I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro Magazine*, vol. 16, pp. 42-50, April 1996.

TRIEBEL, W.A.: *The 80386, 80486, and Pentium Processor*, Upper Saddle River, NJ: Prentice Hall, 1998.

UNGER, S.H.: "A Computer Oriented Toward Spatial Problems," *Proc. IRE*, vol. 46, pp. 1744-1750, 1958.

VAHALIA, U.: *UNIX Internals*, Upper Saddle River, NJ: Prentice Hall, 1996.

VAN DER POEL, W.L.: "The Software Crisis, Some Thoughts and Outlooks," *Proc. IFIP Congr. 68*, pp. 334-339, 1968.

VAN STEEN, M., HOMBURG, P.C., and TANENBAUM, A.S.: "The Architectural Design of Globe: A Wide-Area Distributed System," *IEEE Concurrency*, **1999.**

VERSTOEP, K., LANGEDOEN, K., and BAL, H.E.: "Efficient Reliable Mulicast on Myrinet," *Proc. 1996 Int'l. Conf. on Parallel Processing*, IEEE, pp. 156-165, 1996.

WEAVER, D.L., and GERMOND, T.: *The SPARC Architecture Manual, Version 9*, Englewood Cliffs, NJ: Prentice Hall, 1994.

WILKES, M.V.: "Computers Then and Now," *J. ACM*, vol. 15, pp. 1-7, Jan. 1968.

WILKES.M.V.: "The Best Way to Design and Automatic Calculating Machine," *Proc. Manchester Univ. Computer Inaugural Conf.*, 1951.

WILKINSON, B.: *Computer Architecture: Design and Performance, 2nd ed.*, Englewood Cliffs, NJ: Prentice Hall, 1994.

WILKINSON, B. and ALLEN, M.: *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Upper Saddle River, NJ: Prentice Hall, 1999.

WILSON, J.: "Challenges and Trends in Processor Design," *IEEE Computer Magazine*, vol. 31, pp. 39-48, Jan. 1998.

WILSON, P.: "Floating-Point Survival Kit," *Byte*, vol. 13, pp. 217-226, March 1988.

YEH, T.-Y., and PATT, Y.-N.: "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Int'l. Symp. on Microarchitecture*, ACM/IEEE, pp. 51-61, 1991.

A

NÚMEROS BINARIOS

La aritmética que las computadoras usan difiere en ciertos aspectos de la aritmética que aplica la gente. La diferencia más importante es que las computadoras realizan operaciones con números cuya precisión es finita y fija. Otra diferencia es que casi todas las computadoras usan el sistema binario en lugar del decimal para representar números. Estos temas son materia del presente apéndice.

A.1 NÚMEROS DE PRECISIÓN FINITA

Al efectuar aritmética, pocas veces pensamos en cuántos dígitos decimales se necesitan para representar un número. Los físicos pueden calcular que existen 10^{78} electrones en el Universo sin preocuparse por el hecho de que se requieren 79 dígitos decimales para escribir este número con todas sus cifras. Alguien que calcula el valor de una función con lápiz y papel y que necesita una respuesta con seis cifras significativas simplemente guarda resultados intermedios con siete, ocho, o las que se necesiten. El problema de que el papel no tenga la anchura suficiente para números de siete dígitos nunca se presenta.

Con las computadoras las cosas son muy distintas. En casi todas las computadoras, la cantidad de memoria con que se cuenta para almacenar un número se fija en el momento en que se diseña la máquina. Con algo de esfuerzo, el programador puede representar números dos, tres o incluso más veces más grandes que esta cantidad física, pero hacerlo no altera la naturaleza de este problema. La naturaleza finita de la computadora nos obliga a manejar

sólo números que se puedan representar con un número fijo de dígitos, tales números se conocen como **números de precisión finita**.

A fin de estudiar las propiedades de los números de precisión finita, examinemos el conjunto de enteros positivos que pueden representarse con tres dígitos decimales, sin punto decimal y sin signo. Este conjunto tiene exactamente 1000 miembros: 000, 001, 002, 003, ..., 999. Con esta restricción, es imposible expresar cierto tipo de números, como

1. Números mayores que 999.
2. Números negativos.
3. Fracciones.
4. Números irracionales.
5. Números complejos.

Una propiedad importante de la aritmética del conjunto de todos los enteros es la **cerradura** respecto a las operaciones de suma, resta y multiplicación. En otras palabras, para todo par de enteros i y j, $i+j$, $i-j$ e $i \times j$ son también enteros. El conjunto de enteros no está cerrado respecto a la división, porque existen valores de i y j para los cuales i/j no puede expresarse como un entero (p. ej., 7/2 y 1/0).

Los números de precisión finita no están cerrados respecto a ninguna de estas cuatro operaciones básicas, como se muestra en seguida usando números decimales de tres dígitos como ejemplo:

600 + 600 = 1200	(demasiado grande)
003 – 005 = –2	(negativo)
050 × 050 = 2500	(demasiado grande)
007 / 002 = 3.5	(no es entero)

Las violaciones pueden dividirse en dos clases mutuamente exclusivas: operaciones cuyo resultado es más grande que el número más grande del conjunto (error de desbordamiento) o más pequeño que el número más pequeño del conjunto (error de subdesbordamiento), y operaciones cuyo resultado no es ni demasiado grande ni demasiado pequeño; simplemente no es miembro del conjunto. De las cuatro violaciones anteriores, las primeras tres son ejemplos del primer caso, y la cuarta es un ejemplo del segundo.

Puesto que las computadoras tienen memorias finitas y por tanto forzosamente realizan aritmética con números de precisión finita, los resultados de ciertos cálculos serán, desde el punto de vista de las matemáticas clásicas, equivocados. Un dispositivo de cálculo que da una respuesta errónea aunque está funcionando perfectamente podría parecer extraño a primera vista, pero el error es una consecuencia lógica de su naturaleza finita. Algunas computadoras tienen hardware especial que detecta errores de desbordamiento.

El álgebra de los números de precisión finita es diferente del álgebra normal. Por ejemplo, consideremos la ley asociativa:

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

Evaluemos ambos miembros para $a = 700$, $b = 400$, $c = 300$. Para calcular el miembro izquierdo, calculemos primero $(b - c)$, que es 100, y sumemos luego esta cantidad a a, lo que

da 800. Para calcular el miembro derecho, calculemos primero $(a + b)$, que produce un desbordamiento en la aritmética finita de enteros de tres dígitos. El resultado depende de la máquina que se esté usando, pero no será 1100. Restar 300 a un número que no es 1100 no da 800. La ley asociativa no se cumple. El orden de las operaciones es importante.

Como ejemplo adicional, consideremos la ley distributiva:

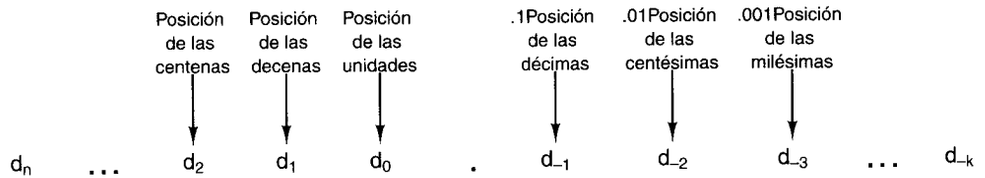
$$a \times (b - c) = a \times b - a \times c$$

Evaluemos ambos miembros para $a = 5$, $b = 210$, $c = 195$. El miembro izquierdo es 5×15, que da 75. El miembro derecho no es 75 porque $a \times b$ causa un desbordamiento.

A juzgar por estos ejemplos, podríamos concluir que si bien las computadoras son dispositivos de propósito general, su naturaleza finita los hace poco apropiados para la aritmética. Claro que esta conclusión no es correcta, pero sirve para ilustrar la importancia de entender cómo funcionan las computadoras y qué limitaciones tienen.

A.2 SISTEMAS NUMÉRICOS CON BASE

Un número decimal ordinario como los que todos conocemos consiste en una serie de dígitos decimales y, posiblemente, un punto decimal. La forma general y su interpretación usual se muestran en la figura A-1. Se tomó la decisión de usar 10 como **base** para la exponenciación porque estamos usando números decimales, es decir, base 10. Al tratar con computadoras, a menudo es mejor utilizar bases distintas de 10. Las bases más importantes son 2, 8 y 16. Los sistemas de numeración basados en estas bases se llaman **binario**, **octal** y **hexadecimal**, respectivamente.



$$\text{Número} = \sum_{i=-k}^{n} d_i \times 10^i$$

Figura A-1. Forma general de un número decimal.

Un sistema de numeración base k requiere k símbolos distintos para representar los dígitos del 0 a $k - 1$. Los números decimales se representan con los diez dígitos decimales

0 1 2 3 4 5 6 7 8 9

En contraste, los números binarios no usan estos diez dígitos; todos se representan exclusivamente con los dos dígitos binarios

0 1

Los números octales se representan con los ocho dígitos octales

0 1 2 3 4 5 6 7

En el caso de los números hexadecimales, se necesitan 16 dígitos. Por tanto requerimos seis símbolos nuevos. La convención es usar las letras mayúsculas de la A a la F para los seis dígitos que siguen al 9. Así, los números hexadecimales se representan con los dígitos

0 1 2 3 4 5 6 7 8 9 A B C D E F

La expresión "dígito binario", que se refiere al 1 o al 0, dio origen al término **bit** (*BInary digiT*). La figura A-2 muestra el número decimal 2001 expresado en forma binaria, octal y hexadecimal. El número 7B9 obviamente es hexadecimal, porque el número B sólo puede ocurrir en números hexadecimales. Sin embargo, el número 111 podría estar en cualquiera de los cuatro sistemas de numeración que hemos mencionado. A fin de evitar la ambigüedad, se usa un subíndice de 2, 8, 10 o 16 para indicar la base cuando no es obvio por el contexto.

Binario 1 1 1 1 1 0 1 0 0 0 1

$1\times2^{10}+1\times2^{9}+1\times2^{8}+1\times2^{7}+1\times2^{6}+0\times2^{5}+1\times2^{4}+0\times2^{3}+0\times2^{2}+0\times2^{1}+1\times2^{0}$

1024 + 512 + 256 + 128 + 64 + 0 +16 + 0 + 0 + 0 + 1

Octal 3 7 2 1

$3\times8^{3}+7\times8^{2}+2\times8^{1}+1\times8^{0}$

1536 + 448 + 16 + 1

Decimal 2 0 0 1

$2\times10^{3}+0\times10^{2}+0\times10^{1}+1\times10^{0}$

2000 + 0 + 0 + 1

Hexadecimal 7 D 1

$7\times16^{2}+13\times16^{1}+1\times16^{0}$

1792 + 208 + 1

Figura A-2. El número 2001 en binario, octal y hexadecimal.

Como ejemplo de la notación binaria, octal, decimal y hexadecimal, considere la figura A-3, que muestra una serie de enteros no negativos expresados en cada uno de estos cuatro sistemas. Tal vez algún arqueólogo miles de años en el futuro descubrirá esta tabla y la usará como Piedra Roseta para descifrar los sistemas de numeración de fines del siglo XX y principios del siglo XXI.

Decimal	Binario	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5665	BA

Figura A-3. Números decimales y sus equivalentes binarios, octales y hexadecimales.

A.3 CONVERSIÓN DE UNA BASE A OTRA

La conversión entre números octales o hexadecimales y números binarios es fácil. Para convertir un número binario en octal, basta con dividirlo en grupos de tres bits, de modo que los tres bits que están inmediatamente a la izquierda (o derecha) del punto decimal (en este caso llamado punto binario) formen un grupo, los tres bits que siguen a la izquierda formen otro grupo y así. Cada grupo de 3 bits se puede convertir directamente en un solo dígito octal, del 0 al 7, según la conversión dada en las primeras ocho líneas de la figura A-3. Podría ser necesario añadir uno o dos ceros a la izquierda o a la derecha para completar un grupo de 3 bits. La conversión de octal a binario es igualmente trivial. Cada dígito octal simplemente se sustituye por el número binario de tres bits equivalente. La conversión de hexadecimal a

binario es en lo esencial igual a la de octal a binario, sólo que cada dígito hexadecimal corresponde a un grupo de 4 bits, en lugar de 3 bits. La figura A-4 da algunos ejemplos de conversiones.

Ejemplo 1

Hexadecimal		1		9		4		8	.	B	6
Binario	0	0	0	1	1	0	0	1	0	1	0	0	1	0	0	0	.	1	0	1	1	0	1	1	0	0
Octal		1		4		5		1		0	.	5		5		4										

Ejemplo 2

Hexadecimal		7		B		A		3	.	B	C	4
Binario	0	1	1	1	1	0	1	1	1	0	1	0	0	0	1	1	.	1	0	1	1	1	1	0	0	0	1	0	0
Octal		7		5		6		4		3	.	5		7		0		4										

Figura A-4. Ejemplos de conversión de octal a binario y de hexadecimal a binario.

La conversión de números decimales a binarios puede efectuarse de dos formas distintas. El primer método es consecuencia directa de la definición de números binarios. Se resta al número la potencia de 2 menor que el número. Luego se repite el proceso con la diferencia. Una vez que el número se ha descompuesto en potencias de 2, el número binario puede armarse con unos en las posiciones de bit que corresponden a potencias de 2 usadas en la descomposición, y ceros en las demás posiciones.

El otro método (sólo para enteros) consiste en dividir el número entre 2. El cociente se escribe directamente debajo del número original y el residuo, 0 o 1, se escribe junto al cociente. Luego se considera el cociente y se repite el proceso hasta llegar a un cociente de 0. El resultado de este proceso es dos columnas de números, los cocientes y los residuos. Ahora el número binario puede leerse directamente de la columna de los residuos comenzando desde abajo. La figura A-5 muestra un ejemplo de conversión de decimal a binario.

Los enteros binarios también pueden convertirse a decimal de dos maneras. Un método consiste en obtener la sumatoria de las potencias de 2 que corresponden a los bits 1 del número. Por ejemplo,

$$10110 \text{ es } 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

En el otro método, el número binario se escribe verticalmente, con un bit en cada línea, con el bit de la extrema izquierda hasta abajo. La línea de hasta abajo se llama línea 1, la que está arriba, línea 2, etc. El número decimal se construye en una columna paralela junto al número binario. Comenzamos escribiendo un 1 en la línea 1. La entrada de la línea n corresponde a dos veces la entrada de la línea $n - 1$ más el bit que está en la línea n (0 o 1). La entrada de la línea de hasta arriba es la respuesta. En la figura A-6 se da un ejemplo de este método de conversión de binario a decimal.

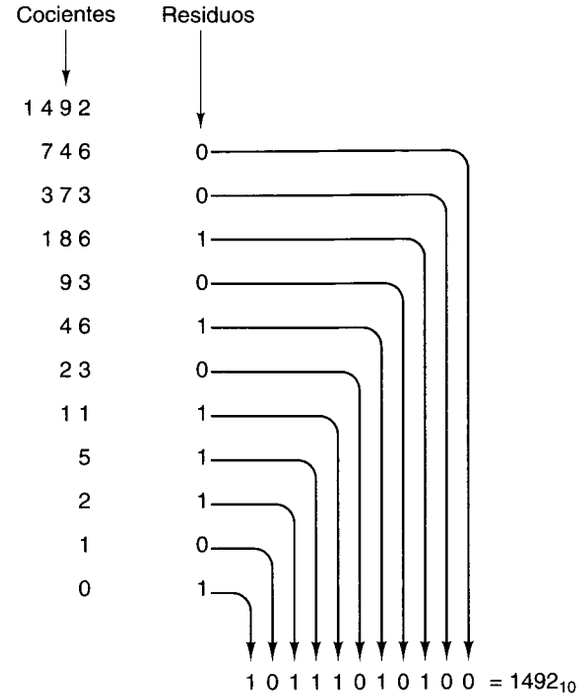


Figura A-5. Conversión del número decimal 1492 a binario por división sucesiva entre 2, comenzando desde arriba y trabajando hacia abajo. Por ejemplo, 93 dividido entre 2 da un cociente de 46 y un residuo de 1, que se escribe en la línea de abajo.

La conversión de decimal a octal y de decimal a hexadecimal puede efectuarse convirtiendo primero a binario y luego al sistema deseado, o restando potencias de 8 o de 16.

A.4 NÚMEROS BINARIOS NEGATIVOS

Se han usado cuatro sistemas distintos para representar números negativos en las computadoras digitales en un momento u otro de la historia. El primero se llama **magnitud con signo**. En este sistema el bit de la extrema izquierda es el bit de signo (0 es + y 1 es –) y los bits restantes contienen la magnitud absoluta del número.

El segundo sistema, llamado **complemento a uno**, también tiene un bit de signo, y se usa 0 para el signo positivo y 1 para el signo negativo. Para negar un número, sustituya cada 1 por 0 y cada 0 por 1. Esto también se hace con el bit de signo. El complemento a uno ya es obsoleto.

El tercer sistema, llamado **complemento a dos**, también tiene un bit de signo que es 0 para el signo positivo y 1 para el signo negativo. La negación de un número es un proceso de dos pasos. Primero, cada 1 se sustituye por un 0 y cada 0 por un 1, igual que en el complemento a 1. Luego se suma 1 al resultado. La suma binaria es igual a la suma decimal, excepto

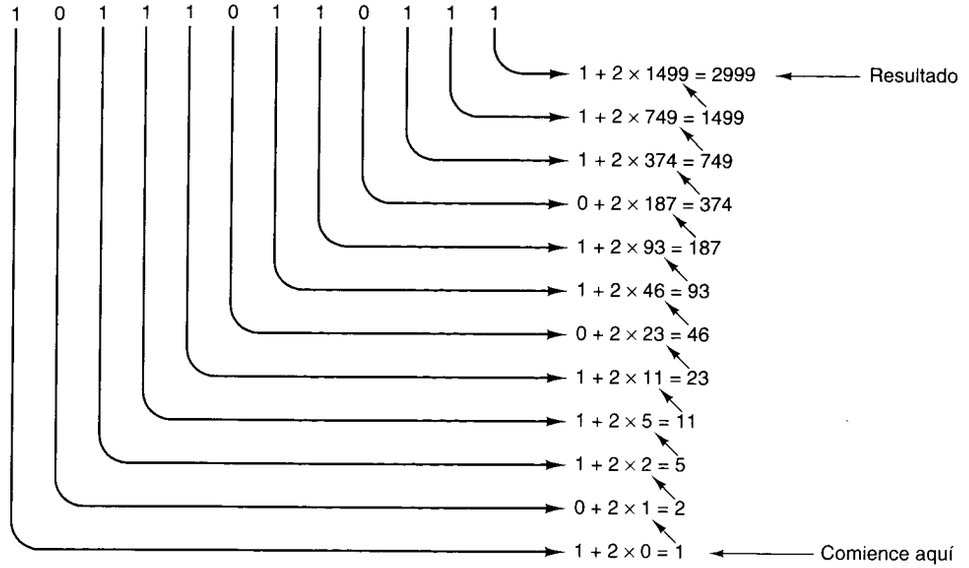


Figura A-6. Conversión del número binario 101110110111 a decimal por duplicación sucesiva, comenzando hasta abajo. Cada línea se forma duplicando la de abajo y sumándole el bit correspondiente. Por ejemplo, 749 es dos veces 374 más el bit 1 que está en la misma línea que 749.

que se genera un acarreo si la suma es mayor que 1, no si la suma es mayor que 9. Por ejemplo, la conversión de 6 a complemento a dos se efectúa en dos pasos:

00000110 (+6)
11111001 (–6 en complemento a uno)
11111010 (–6 en complemento a dos)

Si hay un acarreo hacia la izquierda del bit de la extrema izquierda, se desecha.

El cuarto sistema, que para números de m bits se llama **exceso de 2^{m-1}**, representa un número almacenándolo como la suma del mismo número y 2^{m-1}. Por ejemplo, en el caso de números de 8 bits, $m = 8$, el sistema se llama exceso de 128, y un número se almacena como su valor verdadero más 128. Así, –3 se convierte en –3 + 128 = 125, y –3 se representa con el número binario de 8 bits que corresponde a 125 (01111101). Los números de –128 a +127 se transforman en los números de 0 a 255, todos los cuales pueden expresarse como un entero positivo de 8 bits. Resulta interesante que este sistema es idéntico al de complemento a dos pero con el bit de signo invertido. La figura A-7 proporciona ejemplos de números negativos en los cuatro sistemas.

Los sistemas tanto de magnitud con signo como de complemento a uno tienen dos representaciones para el cero: un cero positivo y un cero negativo. Esta situación es indeseable. El sistema de complemento a dos no tiene este problema porque el complemento a dos de más cero también es más cero. Sin embargo, este sistema tiene una singularidad distinta. El patrón de bits que consiste en un 1 seguido de puros ceros es su propio complemento. El resultado es

N decimal	N binario	–N mag. c/signo	–N comp. a 1	–N comp. a 2	–N exceso en 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11011010	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	No existe	No existe	No existe	10000000	00000000

Figura A-7. Números negativos de 8 bits en cuatro sistemas.

que la gama de números positivos y negativos es asimétrica; hay un número negativo que no tiene contraparte positiva.

La razón de estos problemas no es difícil de encontrar: queremos un sistema de codificación que tenga dos propiedades:

1. Sólo una representación para el cero.
2. Exactamente tantos números positivos como negativos.

El problema es que cualquier conjunto de números que tenga tantos números positivos como negativos y sólo un cero tendrá un número impar de miembros, mientras que m bits dan origen a un número par de patrones de bits. Siempre habrá un patrón de bits de más o de

menos, sea cual sea la representación que se escoja. Este patrón de bits extra se puede usar para –0 o para un número negativo grande, o para algo más, pero sea cual sea el uso que se le dé, siempre será un problema.

A.5 ARITMÉTICA BINARIA

La tabla de la suma para números binarios se da en la figura A-8.

Sumando	0	0	1	1
Sumando	+0	+1	+0	+1
Suma	0	1	1	0
Acarreo	0	0	0	0

Figura A-8. Tabla de suma en binario.

Dos números binarios pueden sumarse comenzando en el bit de la extrema derecha y sumando los bits correspondientes de los dos sumandos. Si se genera un acarreo, se lleva una posición a la izquierda, igual que en la aritmética decimal. En aritmética de complemento a uno, un acarreo generado por la suma de los bits de la extrema izquierda se suma al bit de la extrema derecha. Este proceso se llama acarreo al otro extremo. En aritmética de complemento a dos, un acarreo generado por la suma de los bits de la extrema izquierda simplemente se desecha. En la figura A-9 se muestran ejemplos de aritmética binaria.

Decimal	Complemento a 1	Complemento a 2
10	00001010	00001010
+ (–3)	11111100	11111101
+7	1 00000110	1 00000111
	acarreo 1	se desecha
	00000111	

Figura A-9. Suma en complemento a uno y en complemento a dos.

Si los sumandos tienen signos opuestos, no puede haber un error de desbordamiento. Si tienen el mismo signo y el resultado tiene el signo opuesto, ha ocurrido un error de desbordamiento y la respuesta no es correcta. En aritmética tanto de complemento a uno como de complemento a dos, ocurre un desbordamiento si y sólo si el acarreo hacia el bit de signo difiere del acarreo desde el bit de signo. Casi todas las computadoras conservan el acarreo desde el bit de signo, pero el acarreo hacia el bit de signo no es visible cuando se examina la respuesta. Es por esto que casi siempre se incluye un bit de desbordamiento especial.

PROBLEMAS

1. Convierta los siguientes números a binario: 1984, 4000, 8192.
2. ¿Qué es 1001101001 (binario) en decimal? ¿En octal? ¿En hexadecimal?
3. ¿Cuáles de los siguientes son números hexadecimales válidos? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.
4. Exprese el número decimal 100 en todas las bases de 2 a 9.
5. ¿Cuántos enteros positivos distintos se pueden expresar con k dígitos empleando números de base r?
6. Casi todos nosotros sólo podemos contar hasta 10 con los dedos; sin embargo, los especialistas en computación pueden contar más allá del 10. Si consideramos cada dedo como un bit binario, y un dedo extendido representa 1 mientras que uno doblado representa 0, ¿hasta qué número podemos contar usando ambas manos? ¿Y usando también ambos pies? Ahora use ambas manos y ambos pies, pero el dedo gordo de su pie izquierdo hará las veces de bit de signo para números en complemento a dos. ¿Qué intervalo de números podemos expresar?
7. Realice los siguientes cálculos con números de 8 bits en complemento a dos:

00101101	11111111	00000000	11110111
+01101111	+11111111	–11111111	–11110111

8. Repita el cálculo del problema anterior pero ahora en complemento a uno.
9. Considere los siguientes problemas de suma para números binarios de tres bits en complemento a dos. Para cada suma, indique

   a. Si el bit de signo del resultado es 1.
   b. Si los tres bits de orden bajo son 0.
   c. Si ocurrió un desbordamiento.

000	000	111	100	100
+001	+111	+110	+111	+100

10. Los números decimales con signo que constan de n dígitos se pueden representar con $n + 1$ dígitos sin usar un signo. Los números positivos tienen 0 como dígito de la izquierda. Los números negativos se forman restando cada dígito de 9. Así, el negativo de 014725 es 985274. Tales números se llaman números en complemento a nueve y son análogos a los números binarios en complemento a uno. Exprese los siguientes como números de tres dígitos en complemento a nueve: 6, –2, 100, –14, –1, 0.
11. Determine la regla para la suma de números en complemento a nueve y luego realice las siguientes sumas.

0001	0001	9997	9241
+9999	+9998	+9996	+0802

12. El complemento a 10 es análogo al complemento a dos. Un número negativo en complemento a 10 se forma sumando 1 al número en complemento a nueve correspondiente, desechando el acarreo final. ¿Cuál es la regla para la suma en complemento a 10?

13. Construya las tablas de multiplicación para los números base 3.

14. Multiplique 0111 por 0011 en binario.

15. Escriba un programa que tome un número decimal con signo como cadena ASCII e imprima su representación en complemento a dos en binario, octal y hexadecimal.

16. Escriba un programa que tome dos cadenas de 32 caracteres ASCII que contengan ceros y unos; cada cadena representa un número binario de 32 bits en complemento a dos. El programa deberá imprimir su suma como una cadena de 32 caracteres ASCII con ceros y unos.

B

NÚMEROS DE PUNTO FLOTANTE

En muchos cálculos el intervalo de números que se usan es muy grande. Por ejemplo, en un cálculo astronómico podrían intervenir la masa del electrón, 9×10^{-28} gramos, y la masa del Sol, 2×10^{33} gramos. Este intervalo es mayor que 10^{60}. Tales números podrían representarse con

00000000000000000000000000000000.00000000000000000000000000000009
20000000000000000000000000000000.00000000000000000000000000000000

y todos los cálculos podrían efectuarse llevando 34 dígitos a la izquierda del punto decimal y 28 posiciones a la derecha. Ello permitiría tener 62 dígitos significativos en el resultado. En una computadora binaria se podría usar aritmética de múltiple precisión para tener suficiente significancia. Sin embargo, la masa del Sol no se conoce ni siquiera con una precisión de cinco dígitos significativos, mucho menos 62. De hecho, pocas mediciones de cualquier tipo se pueden (o tienen que) efectuar con una precisión de 62 dígitos significativos. Aunque sería posible llevar todos los resultados intermedios con 62 dígitos significativos y luego desechar 50 o 60 de ellos antes de imprimir los resultados finales, hacerlo sería un desperdicio de tiempo de CPU y de memoria.

Lo que se necesita es un sistema para representar números en el que el intervalo de los números que pueden expresarse sea independiente del número de dígitos significativos. En este apéndice veremos un sistema así, que se basa en la notación científica de uso común en física, química e ingeniería.

B.1 PRINCIPIOS DE PUNTO FLOTANTE

Una forma de separar el intervalo y la precisión es expresar los números en la conocida notación científica

$$n = f \times 10^e$$

donde f es la **fracción** o **mantisa** y e es un entero positivo o negativo llamado **exponente**. La versión para computadora de esta notación se llama **punto flotante**. He aquí algunos ejemplos de números expresados en esta forma:

3.14	$= 0.314$	$\times 10^{1}$	$= 3.14$	$\times 10^{0}$
0.000001	$= 0.1$	$\times 10^{-5}$	$= 1.0$	$\times 10^{-6}$
1941	$= 0.1941$	$\times 10^{4}$	$= 1.941$	$\times 10^{3}$

El intervalo está determinado por el número de dígitos del exponente, y la precisión está determinada por el número de dígitos de la fracción. Puesto que hay más de una forma de representar un número dado, normalmente se escoge una forma como estándar. A fin de investigar las propiedades de este método de representación de los números, consideremos una representación, R, con una fracción de tres dígitos y signo dentro del intervalo $0.1 \le |f| < 1$ o cero, y un exponente de dos dígitos con signo. La magnitud de estos números varía entre $+0.100 \times 10^{-99}$ y $0.999 \times 10^{+99}$, un intervalo de casi 199 órdenes de magnitud, y sin embargo sólo se necesitan cinco dígitos y dos signos para almacenar un número.

Podemos usar números de punto flotante para modelar el sistema de números reales de las matemáticas, aunque hay varias diferencias importantes. La figura B-1 muestra un esquema muy exagerado de la línea de los números reales. La línea real se divide en siete regiones:

1. Números negativos grandes menores que -0.999×10^{99}.
2. Números negativos entre -0.999×10^{99} y -0.100×10^{-99}.
3. Números negativos pequeños con una magnitud menor que 0.100×10^{-99}.
4. Cero.
5. Números positivos pequeños con una magnitud menor que 0.100×10^{-99}.
6. Números positivos entre 0.100×10^{-99} y 0.999×10^{99}.
7. Números positivos grandes mayores que 0.999×10^{99}.

Una diferencia importante entre el conjunto de los números que pueden representarse con tres dígitos en la fracción y dos en el exponente, y los números reales, es que los primeros no se pueden usar para expresar números en las regiones 1, 3, 5 o 7. Si el resultado de una operación aritmética es un número en las regiones 1 o 7 —por ejemplo, $10^{60} \times 10^{60} = 10^{120}$— ocurrirá un **error de desbordamiento** y la respuesta será incorrecta. La razón está en la naturaleza finita de la representación de los números y es inevitable. Así mismo, un resultado

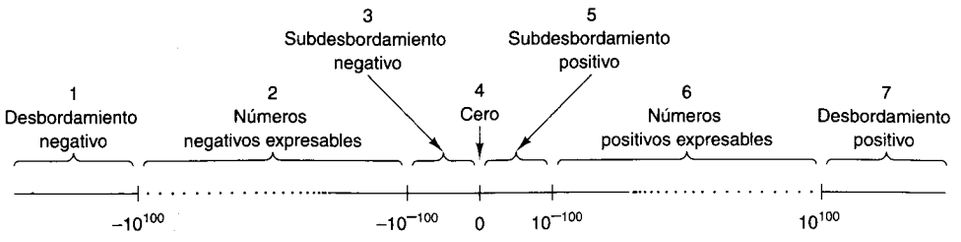


Figura B-1. La línea de los números reales se puede dividir en siete regiones.

en la región 3 o 5 tampoco puede expresarse. Esta situación se llama **error de subdesbordamiento**. Este tipo de error es menos grave que el de desbordamiento, porque en muchos casos 0 es una aproximación satisfactoria para números de las regiones 3 y 5. Un saldo en el banco de 10^{-102} dólares no es mucho mejor que un saldo de 0.

Otra diferencia importante entre los números de punto flotante y los números reales es su densidad. Entre cualesquier dos números reales x y y hay otro número real, por más cercano que esté x a y. Esta propiedad proviene del hecho de que para cualesquier números reales distintos, x y y, $z = (x + y)/2$ es un número real que está entre ellos. Los números reales forman un continuo.

Los números de punto flotante, en cambio, no forman un continuo. Se pueden expresar exactamente 179,100 números positivos en el sistema de cinco dígitos y dos signos que usamos antes, 179,100 números negativos y 0 (que se puede expresar de muchas formas), para un total de 358,201 números. Del número infinito de números reales que hay entre -10^{+100} y $+0.999 \times 10^{99}$, sólo 358,201 se pueden especificar con esta notación. Dichos números se simbolizan con los puntos de la figura B-1. Es muy posible que el resultado de un cálculo sea uno de los otros números, aunque esté en la región 2 o 6. Por ejemplo, $+0.100 \times 10^{3}$ dividido entre 3 no se puede expresar *con exactitud* en nuestro sistema. Si el resultado de un cálculo no se puede expresar en la representación numérica que se está usando, lo que obviamente debe hacerse es usar el número más cercano que sí pueda expresarse. Este proceso se llama **redondeo**.

La distancia entre números adyacentes que pueden expresarse no es constante a lo largo de las regiones 2 o 6. La separación entre $+0.998 \times 10^{99}$ y $+0.999 \times 10^{99}$ es inmensamente mayor que la distancia entre $+0.998 \times 10^{0}$ y $+0.999 \times 10^{0}$. Sin embargo, si expresamos la distancia entre un número y su sucesor como un porcentaje del número, no hay variación sistemática dentro de la región 2 o 6. En otras palabras, el **error relativo** introducido por el redondeo es aproximadamente el mismo para números pequeños que para números grandes.

Aunque la explicación anterior se refirió a un sistema de representación con una fracción de tres dígitos y un exponente de dos dígitos, las conclusiones que se sacan son válidas también para otros sistemas de representación. Modificar el número de dígitos de la fracción o exponente simplemente desplaza los límites de las regiones 2 y 6, y cambia el número de puntos expresables y que aquéllas contienen. Incrementar el número de dígitos en la fracción aumenta la densidad de puntos y por tanto mejora la precisión de las aproximaciones. In-

crementar el número de dígitos del exponente aumenta el tamaño de las regiones 2 y 6 al encoger las regiones 1, 3, 5 y 7. En la figura B-2 se muestran las fronteras aproximadas de la región 6 para números decimales de punto flotante con diversos tamaños de la fracción y el exponente.

Dígitos en la fracción	Dígitos en el exponente	Cota inferior	Cota superior
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figura B-2. Límites superior e inferior aproximados de los números decimales de punto flotantes expresables (no normalizados).

En las computadoras se usa una variación de esta representación. Por eficiencia, la exponenciación es base 2, 4, 8 o 16, no 10, y entonces la fracción consiste en una cadena de dígitos binarios, base 4, octales o hexadecimales. Si el primero de esos dígitos a la izquierda es cero, todos los dígitos pueden desplazarse una posición a la izquierda y el exponente reducirse en 1, sin alterar el valor del número (siempre que no haya subdesbordamiento). Se dice que una fracción cuyo dígito de la izquierda no es cero está **normalizada**.

Los números normalizados generalmente son preferibles a los no normalizados porque sólo hay una representación normalizada, mientras que hay muchas representaciones no normalizadas. En la figura B-3 se dan ejemplos de números de punto flotante normalizados para dos bases de exponenciación. En estos ejemplos se muestra una fracción de 16 bits (incluido el bit de signo) y un exponente de siete bits empleando notación exceso de 64. El punto base está a la izquierda del bit de la extrema izquierda de la fracción, es decir, a la derecha del exponente.

B.2 ESTÁNDAR DE PUNTO FLOTANTE IEEE 754

Hasta cerca de 1980, cada fabricante de computadoras tenía su propio formato de punto flotante. Sobra decir que todos eran diferentes. Peor aún, algunos de ellos efectuaban aritmética incorrecta porque la de punto flotante tiene algunas sutilezas que no son obvias para el diseñador de hardware ordinario.

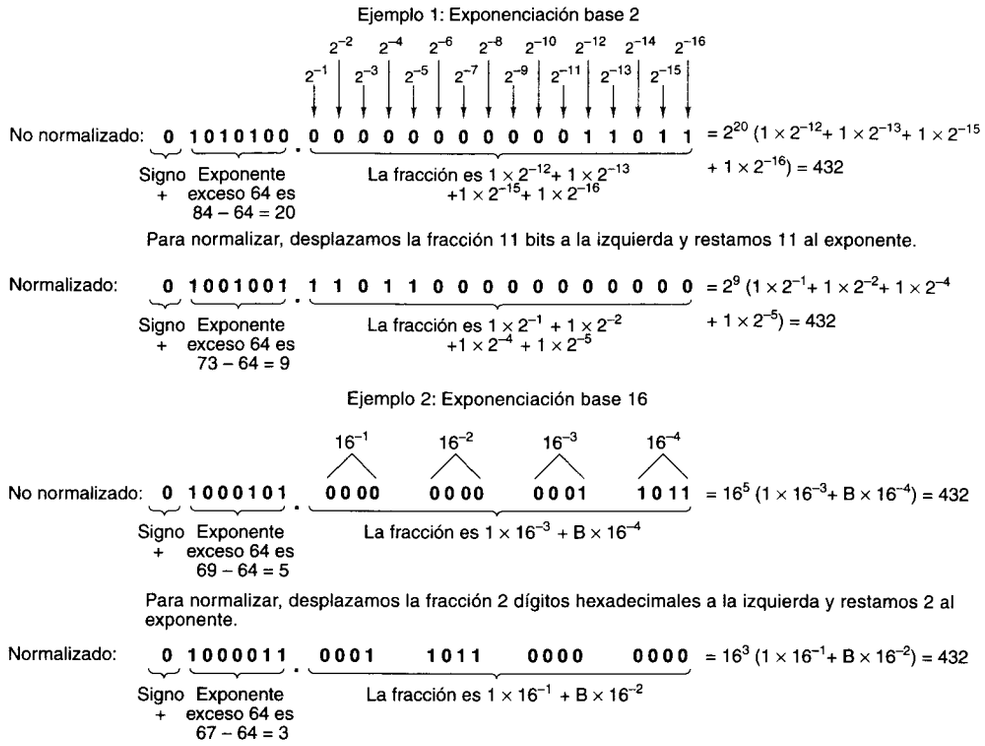


Figura B-3. Ejemplos de números de punto flotante normalizados.

A fin de rectificar esta situación, a fines de los años setenta el IEEE formó un comité para estandarizar la aritmética de punto flotante. La meta no sólo era poder intercambiar datos de punto flotante entre diferentes computadoras, sino también proporcionar a los diseñadores de hardware un diseño que se sabía era correcto. Los trabajos resultantes dieron pie al estándar IEEE 754 (IEEE, 1985). Casi todas las CPU actuales (incluidas las Intel, SPARC y JVM que estudiamos en este libro) tienen instrucciones de punto flotante que se ajustan al estándar de punto flotante IEEE. A diferencia de muchos estándares, que tienden a ser términos medios con los que nadie está contento, éste no está nada mal, en parte porque fue en gran medida el trabajo de una sola persona, el profesor de matemáticas de Berkeley William Kahan. En el resto de la sección describiremos el estándar.

El estándar define tres formatos: precisión sencilla (32 bits), doble precisión (64 bits) y precisión extendida (80 bits). El formato de precisión extendida pretende reducir los errores de redondeo; se usa primordialmente dentro de unidades aritméticas de punto flotante, por lo que no hablaremos más de él. Los formatos tanto de precisión sencilla como doble usan la base 2 para las fracciones y notación en exceso para los exponentes. Los formatos se muestran en la figura B-4.

Ambos formatos comienzan con un bit de signo para el número en su totalidad: 0 para positivo y 1 para negativo. Luego viene el exponente, que usa exceso en 127 en el caso de la

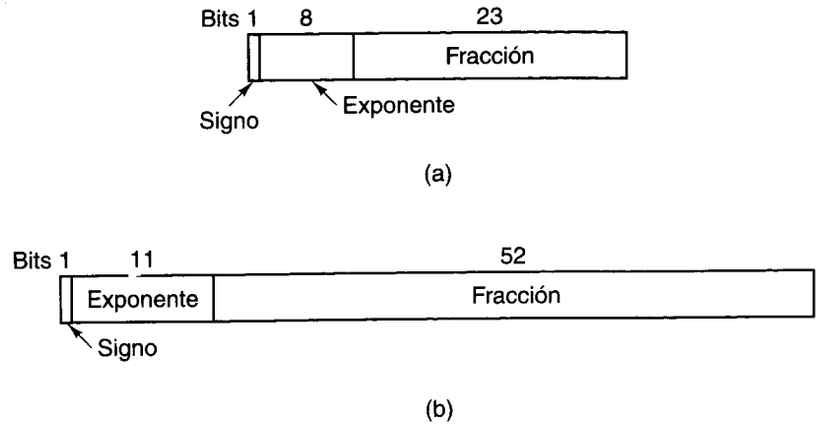


Figura B-4. Formatos de punto flotante IEEE. (a) Precisión sencilla. (b) Doble precisión.

precisión sencilla y exceso en 1023 para la doble precisión. Los exponentes mínimo (0) y máximo (255 y 2047) no se usan con números normalizados; tienen usos especiales que se describen más adelante. Por último, tenemos las fracciones, de 23 y 52 bits, respectivamente.

Una fracción normalizada comienza con un punto binario, seguido de un bit 1, y luego el resto de la fracción. Siguiendo una práctica iniciada en el PDP-11, los autores del estándar se dieron cuenta de que el bit inicial de la fracción no tiene que almacenarse, ya que puede simplemente darse por hecho que está presente. Por tanto, el estándar define la fracción de una forma un poco distinta a lo acostumbrado. La fracción consiste en un bit 1 implícito, un punto binario implícito, y 23 o 52 bits arbitrarios. Si todos los bits de la fracción son ceros, la fracción tiene el valor numérico 1.0; si todos son unos, la fracción es numéricamente un poco menos que 2.0. A fin de evitar confusiones con una fracción convencional, la combinación del 1 implícito, el punto binario implícito y los 23 o 52 bits explícitos se llama **significando** en vez de fracción o mantisa. Todos los números normalizados tienen un significando, s, dentro del intervalo $1 \leq s < 2$.

Las características numéricas de los números de punto flotante IEEE se dan en la figura B-5. Consideremos, por ejemplo, los números 0.5, 1 y 1.5 en formato de precisión sencilla normalizado. Éstos se representan en hexadecimal como 3F000000, 3F800000 y 3FC00000, respectivamente.

Uno de los problemas tradicionales con los números de punto flotante es cómo manejar el subdesbordamiento, el desbordamiento y los números no inicializados. El estándar IEEE aborda los problemas explícitamente, tomando su enfoque en parte de la CDC 6600. Además de los números normalizados, el estándar tiene otros cuatro tipos numéricos, que se describen a continuación y se muestran en la figura B-6.

Surge un problema cuando el resultado de un cálculo tiene una magnitud menor que el número de punto flotante normalizado más pequeño que puede representarse en el sistema. Antes, casi todo el hardware adoptaba uno de dos enfoques: igualar el resultado a cero y continuar, o causar una trampa de desbordamiento de punto flotante hacia cero. Ninguno de

Concepto	Precisión sencilla	Doble precisión
Bits del signo	1	1
Bits del exponente	8	11
Bits de la fracción	23	52
Total de bits	32	64
Sistema de exponente	Exceso en 127	Exceso en 1023
Intervalo del exponente	−126 a +127	−1022 a + 1023
Número normalizado más pequeño	2^{-126}	2^{-1022}
Número normalizado más grande	aprox. 2^{128}	aprox. 2^{1024}
Intervalo decimal	aprox. 10^{-38} a 10^{38}	aprox. 10^{-308} a 10^{308}
Número desnormalizado más pequeño	aprox. 10^{-45}	aprox. 10^{-324}

Figura B-5. Características de los números de punto flotante IEEE.

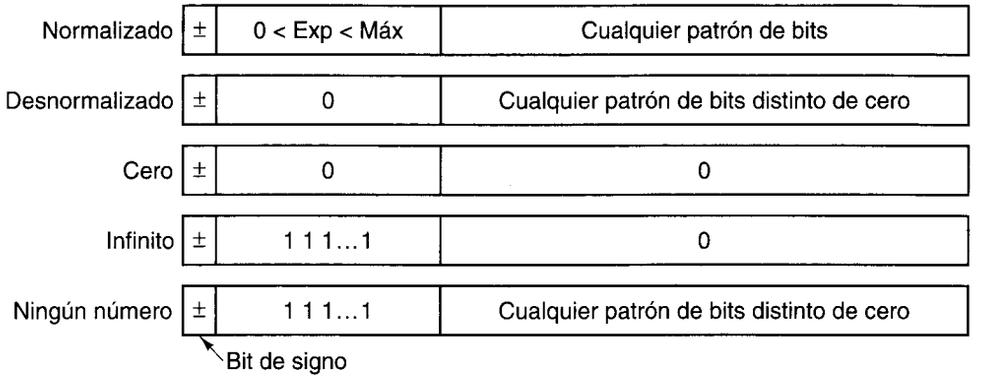


Figura B-6. Representaciones numéricas IEEE.

éstos es realmente satisfactorio, por lo que IEEE inventó los **números desnormalizados**. Estos números tienen un exponente cero y una fracción dada por los siguientes 23 o 52 bits. El bit implícito a la izquierda del punto binario ahora se convierte en 0. Los números desnormalizados se pueden distinguir de los normalizados porque estos últimos no pueden tener un exponente cero.

El número de precisión sencilla normalizado más pequeño tiene 1 como exponente y 0 como fracción, y representa 1.0×2^{-126}. El número desnormalizado más grande tiene 0 como exponente y sólo unos en la fracción, y representa cerca de $0.9999999 \times 2^{-127}$, que es casi la misma cosa. Sin embargo, una cosa que debemos tener presente es que este número sólo tiene 23 bits de significancia, en vez de 24 como todos los números normalizados.

Si los cálculos reducen aún más este resultado, el exponente sigue siendo 0, pero los primeros bits de la fracción se van convirtiendo en ceros, lo que reduce tanto el valor como el número de bits significativos de la fracción. El número desnormalizado más pequeño distinto

de cero consiste en un 1 en el bit de la extrema derecha, con todos los demás 0. El exponente representa 2^{-127} y la fracción representa 2^{-23}, de modo que el valor es 2^{-150}. Este esquema hace posible un subdesbordamiento "gradual", sacrificando significancia en lugar de saltar a 0 cuando el resultado no puede expresarse como número normalizado.

Este esquema cuenta con dos ceros, positivo y negativo, determinados por el bit de signo. Ambos tienen un exponente de 0 y una fracción de 0. Aquí, también, el bit a la izquierda del punto binario es implícitamente 0 en lugar de 1.

El desbordamiento no puede manejarse de forma gradual. No quedan más combinaciones de bits. En vez de ello, se proporciona una representación especial para infinito, que consiste en un exponente solamente de unos (lo cual no está permitido en los números normalizados) y una fracción de 0. Este número puede usarse como operando y se comporta según las reglas matemáticas usuales para el infinito. Por ejemplo, infinito más cualquier cosa es infinito, y cualquier número finito dividido entre infinito es cero. Así mismo, cualquier número finito dividido entre cero da infinito.

¿Y qué pasa con infinito dividido entre infinito? El resultado no está definido. Para manejar este caso se proporciona otro formato especial, llamado **ningún número** (**NaN**, *Not a Number*), que también puede usarse como operando con resultados predecibles.

PROBLEMAS

1. Convierta los números siguientes al formato IEEE de precisión sencilla. Represente los resultados como ocho dígitos hexadecimales.

   a. 9
   b. 5/32
   c. –5/32
   d. 6.125

2. Convierta los siguientes números de punto flotante IEEE de precisión sencilla de hexadecimal a decimal:

   a. 42E48000H
   b. 3F880000H
   c. 00800000H
   d. C7F00000H

3. El formato de los números de punto flotante de precisión sencilla en la 370 tiene un exponente de 7 bits en el sistema exceso de 64, y una fracción que contiene 24 bits más un bit de signo, con el punto binario en el extremo izquierdo de la fracción. La base de exponenciación es 16. El orden de los campos es bit de signo, exponente, fracción. Exprese el número 7/64 como número normalizado en este sistema, en hexadecimal.

4. Los siguientes números binarios de punto flotante consisten en un bit de signo, un exponente base 2 en exceso de 64 y una fracción de 16 bits. Normalícelos.

   a. 0 1000000 0001010100000001
   b. 0 0111111 0000001111111111
   c. 0 1000011 1000000000000000

5. Para sumar dos números de punto flotante hay que ajustar los exponentes (desplazando la fracción) de modo que sean iguales. Luego se pueden sumar las fracciones y normalizar el resultado, si es necesario. Sume los números IEEE de precisión sencilla 3EE00000H y 3D800000H y exprese el resultado normalizado en hexadecimal.

6. La Compañía de Computadoras Económicas decidió sacar una máquina que maneje números de punto flotante de 16 bits. El Modelo 0.001 tiene un formato de punto flotante con un bit de signo, un exponente de 7 bits en exceso de 64, y una fracción de 8 bits. El Modelo 0.002 tiene un bit de signo, un exponente de 5 bits en exceso de 16, y una fracción de 10 bits. Ambas usan exponenciación base 2. ¿Cuáles son los números normalizados más grande y más pequeño en cada modelo? ¿Cuántos dígitos decimales de precisión tiene aproximadamente cada uno? ¿Compraría alguno de ellos?

7. Hay una situación en la que una operación con dos números de punto flotante puede causar una reducción drástica del número de bits significativos en el resultado. ¿Cuál es?

8. Algunos chips de punto flotante tienen incorporada una instrucción de raíz cuadrada. Un posible algoritmo es iterativo (por ejemplo, el de Newton-Raphson). Los algoritmos iterativos necesitan una aproximación inicial y luego la mejoran continuamente. ¿Cómo se puede obtener una raíz cuadrada aproximada rápida de un número de punto flotante?

9. Escriba un procedimiento que sume dos números de punto flotante IEEE de precisión sencilla. Cada número se representa con un arreglo booleano de 32 elementos.

10. Escriba un procedimiento para sumar dos números de punto flotante de precisión sencilla que usan base 16 para el exponente y base 2 para la fracción pero no tienen un bit 1 implícito a la izquierda del punto binario. Un número normalizado tiene 0001, 0010, ..., 1111 como los cuatro bits de extrema izquierda de la fracción, pero no 0000. Un número se normaliza desplazando la fracción a la izquierda 4 bits y sumando 1 al exponente.

ÍNDICE

A

B

C

D

E

F

G

H

I

J

K

L

M

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

ACERCA DEL AUTOR

Andrew S. Tanenbaum estudió la licenciatura en ciencias en el MIT y un doctorado en la University of California en Berkeley. Actualmente es profesor de ciencias de la computación en la Vrije Universiteit en Amsterdam, Países Bajos, donde encabeza el Grupo de Sistemas Computacionales; también es decano de la Advanced School for Computing and Imaging, una escuela de posgrado interuniversitaria dedicada a la investigación de sistemas paralelos, distribuidos y de imágenes avanzados. No obstante, está esforzándose mucho por no convertirse en un burócrata.

El profesor Tanenbaum ha realizado investigaciones sobre compiladores, sistemas operativos, redes y sistemas distribuidos de área local. Sus investigaciones actuales se enfocan principalmente en el diseño de sistemas distribuidos de área extensa que pueden ampliarse a millones de usuarios. Estos proyectos de investigación han dado pie a más de 85 artículos arbitrados en revistas y memorias de congresos, y a cinco libros.

El profesor Tanenbaum también ha producido un volumen considerable de software. Él fue el arquitecto principal del Amsterdam Compiler Kit, una herramienta ampliamente usada para escribir compiladores portátiles, así como de MINIX, un pequeño clon de UNIX creado para ser usado por estudiantes en laboratorios de programación. Junto con sus estudiantes de doctorado y programadores, contribuyó en el diseño de Amoeba, un sistema operativo distribuido de alto rendimiento basado en microkernel. Los sistemas MINIX y Amoeba ya se encuentran disponibles gratuitamente a través de Internet.

Los estudiantes de doctorado del profesor Tanenbaum han cosechado grandes triunfos después de obtener sus grados. Él está muy orgulloso de ellos; en este sentido se asemeja a una mamá gallina.

El profesor Tanenbaum es Socio de la ACM, Socio del IEEE, miembro de la Academia Real de Artes y Ciencias de los Países Bajos, ganador del Karl. V. Karlstrom Outstandig Educator Award de la ACM en 1994 y del ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education; además, está listado en *Who's Who in the World*. La dirección de su página en internet es *http://www.cs.vu.nl/~ast/*.