ESTRUCTURA Y DISEÑO DE COMPUTADORES

LA INTERFAZ HARDWARE / SOFTWARE

DAVID A. **PATTERSON**
JOHN L. **HENNESSY**

Cuarta edición original

EDITORIAL REVERTÉ

Datos de referencia de **MIPS**

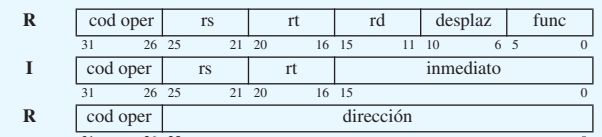
①

NÚCLEO DEL REPERTORIO DE INSTRUCCIONES

NOMBRE	MNE-MO-TÉC-NICO	FOR-MA-TO	OPERACIÓN (en verilog)		COD OP/ FUNC (Hex)
Suma	add	R	R[rd] = R[rs] + R[rt]	1	0/20hex
Suma inmediata	addi	I	R[rt] = R[rs] + ExtSignInm	1,2	8hex
Suma inm. sin signo	addiu	I	R[rt] = R[rs] + ExtSignInm	2	9hex
Suma sin signo	addu	R	R[rd] = R[rs] + R[rt]		0/21hex
And	and	R	R[rd] = R[rs] & R[rt]		0/24hex
And inmediato	andi	I	R[rt] = R[rs] & ExtSignInm	3	chex
Salto si igual	beq	I	if(R[rs] = R[rt] PC = PC + 4 + DirSalto	4	4hex
Salto si distinto	bne	I	if(R[rs] != R[rt] PC = PC + 4 + DirSalto	4	5hex
Salto incondicional	j	J	PC = DirSaltoInc	5	2hex
Saltar y enlazar	jal	J	R[31] = PC +8; PC = DirSaltoInc	5	3hex
Salto con registro	jr	R	PC = R[rs]		0/08hex
Carga de un byte sin signo	lbu	I	R[rt] = {24`b0, M[R[rs] + ExtSignInm](7:0)	2	24hex
Carga de media palabra sin signo	lhu	I	R[rt] = {16`b0, M[R[rs] + ExtSignInm](15:0)	2	25hex
Carga enlazada	ll	I	R[rt] = M[Rs[rs] + ExtSignInm]	2,7	30hex
Carga superior inm.	lui	I	R[rt] = {inm, 16`b0}		fhex
Carga de una palabra	lw	I	R[rt] = M[Rs[rs] + ExtSignInm]	2	23hex
Nor	nor	R	R[rd] = ~(R[rs] | R[rt])		0/27hex
Or	or	R	R[rd] = R[rs] | R[rt]		0/25hex
Or inmediato	ori	I	R[rd] = R[rs] | ExtCeroInm	3	dhex
Fijar si menor que	slt	R	R[rd] = (R[rs] < R[rt]) ? 1:0		0/2ahex
Fijar si menor que inm.	slti	I	R[rd] = (R[rs] < ExtSignInm) ? 1:0	2	ahex
Fijar si menor que inm. sin signo	sltiu	I	R[rd] = (R[rs] < ExtSignInm) ? 1:0	2,6	bhex
Fijar si menor que sin signo	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1:0	6	0/2bhex
Desplazamiento lógico a la izquierda	sll	R	R[rd] = R[rt] << desplaz		0/00hex
Desplazamiento a lógico a la derecha	srl	R	R[rd] = R[rt] >> desplaz		0/02hex
Almacenamiento de un byte	sb	I	M[R[rs] + ExtSignInm](7:0) = R[rt](7:0)	2	28hex
Almacenamiento condicional	sc	I	M[R[rs] + ExtSignInm] = R[rt]; R[rt] =(atomic) ? 1:0	2,7	38hex
Almacenamiento de media palabra	sh	I	M[R[rs] + ExtSignInm](15:0) = R[rt](15:0)	2	29hex
Almacenamiento de una palabra	sw	I	M[R[rs] + ExtSignInm] = R[rt]	2	2bhex
Resta	sub	R	R[rd] = R[rs] - R[rt]	1	0/22hex
Resta sin signo	subu	R	R[rd] = R[rs] - R[rt]		0/23hex

(1) Puede producirse una excepción de desbordamiento
(2) ExtSignImn = {16{inmediato[15], inmediato}
(3) ExtCeroInm = {16{1b'0}, inmediato}
(4) DirSalto = {14{inmediato[15], inmediato, 2`b0}
(5) DirSaltoCond = {PC+4[31:28], direccion, 2`b0}
(6) Los operandos se consideran números sin signo
(7) Pareja atómica comprobar y fijar; R[rt]=1si pareja atómica 0 si no atómica

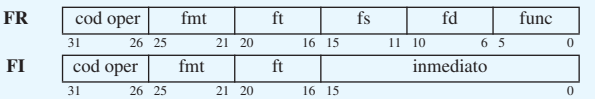
FORMATOS BÁSICOS DE INSTRUCCIÓN



NÚCLEO ARITMÉTICO DEL REPERTORIO DE INSTRUCCIONES ②

NOMBRE	MNE-MO-TÉC-NICO	FOR-MA-TO	OPERACIÓN		COD OP/ FMT/FT/ FUNC (Hex)
Salto si FP cierto	bclt	FI	If (FPcond) PC = PC + 4 + DirSalto	4	11/8/1/--
Salto si FP falso	bclf	FI	If (FPcond) PC = PC + 4 + DirSalto	4	11/8/0/--
División	div	R	Lo = R[rs]/R[rt]; Hi = R[rs]%R[rt]		0/--/--/1a
División sin signo	divu	R	Lo = R[rs]/R[rt]; Hi = R[rs]%R[rt]	6	11/--/--/1b
Suma FP simple	add.s	FR	F[fd] = F[fs] + F[ft]		11/10/--/0
Suma FP doble	add.d	FR	{F[fd],F[fd+1]} = {F[fs], F[fs+1]} + {F[ft], F[ft+1]}		11/11/--/0/
Comparación FP simple	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1:0		11/10/--/y
Comparación FP doble	c.x.d*	FR	Fpcond = ({F[fs], F[fs+1]} *op* {F[ft], F[ft+1]} ? 1:0		11/11/--/y
*(x es eq, lt o le) (*op* es ==, < o <=) (*y* es 32, 3c o 3e)					
División FP simple	div.s	FR	F[fd] = F[fs]/F[ft]		11/10/--/3
División FP doble	div.d	FR	{F[fd],F[fd+1]} = {F[fs], F[fs+1]}/ {F[ft], F[ft+1]}		11/11/--/3
Multiplicación FP simple	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
Multiplicación FP doble	mul.d	FR	{F[fd],F[fd+1]} = {F[fs], F[fs+1]} * {F[ft], F[ft+1]}		11/11/--/2
Resta FP simple	sub.s	FR	F[fd] = F[fs] - F[ft]		11/10/--/1
Resta FP doble	sub.d	FR	{F[fd],F[fd+1]} = {F[fs], F[fs+1]} - {F[ft], F[ft+1]}		11/11/--/1
Carga FP simple	lwcl	I	F[rt] = M[R[rs] + ExtSignInm]	2	31/--/--/--
Carga FP doble	ldcl	I	F[rt] = M[R[rs] + ExtSignInm] F[rt+1] = M[R[rs] + ExtSignInm +4]	2	35/--/--/--
Mover de parte alta	mfhi	R	R[rd] = Hi		0/--/--/10
Mover de parte baja	mflo	R	R[rd] = Lo		0/--/--/12
Mover de control	mfc0	R	R[rd] = CR[rs]		10/0/--/0
Multiplicación	mult	R	{Hi, Lo} = R[rs] * R[rt]		0/--/--/18
Multiplicación sin signo	multu	R	{Hi, Lo} = R[rs] * R[rt]	6	0/--/--/19
Desplazamiento aritmético a la derecha	sra	R	R[rd] = R[rt] >>> desplaz		0/--/--/3
Almacenamiento de FP simple	swcl	I	M[R[rs]+ExtSignInm] = F[rt]	2	39/--/--/--
Almacenamiento de FP doble	sdcl	I	M[R[rs]+ExtSignInm] = F[rt]; M[R[rs]+ExtSignInm+4] = F[rt+1]	2	3d/--/--/--

FORMATOS DE INSTRUCCIÓN PUNTO FLOTANTE



REPERTORIO DE PSEUDOINSTRUCCIONES

NOMBRE	MNEMOTÉCNICO	OPERACIÓN
Salto si menor que	blt	If (R[rs] < R[rt]) PC = etiqueta
Salto si mayor que	bgt	If (R[rs] > R[rt]) PC = etiqueta
Salto si menor que o igual	ble	If (R[rs] <= R[rt]) PC = etiqueta
Salto si mayor que o igual	bge	If (R[rs] >= R[rt]) PC = etiqueta
Carga inmediata	li	R[rd] = inmediato
Mover	move	R[rd] = R[rs]

NOMBRE DE REGISTRO, NÚMERO, USO Y CONVENIO DE LLAMADA

NOMBRE	NÚMERO	USO	¿SE CONSERVA EN UNA LLAMADA?
$zero	0	Valor constante 0	No disponible
$at	1	Ensamblador temporal	No
$v0-$v1	2 - 3	Valores de resultado de funciones y evaluación de expresiones	No
$a0-$a3	4 - 7	Argumentos	No
$t0-$t7	8 - 15	Temporales	No
$s0-$s7	16 - 23	Temporales almacenados	Sí
$t8-$t9	24 - 25	Temporales	No
$k0-$k1	26 - 27	Reservados para el núcleo del Sistema Operativo	No
$gp	28	Puntero global	Sí
$sp	29	Puntero de pila	Sí
$fp	30	Puntero de marco	Sí
$ra	31	Dirección de retorno	Sí

CÓDIGOS DE OPERACIÓN, CONVERSIÓN DE BASE, SÍMBOLOS ASCII ③

MIPS cod oper (31:26)	MIPS cod oper (31:26)	(2)MIPS func (5:0)	Binario	Decimal	Hexadecimal	Carácter ASCII	Decimal	Hexadecimal	Carácter ASCII
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	BNQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jl		00 1000	8	8	BS	72	48	H
addiu	jslr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.v.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	rafhi		01 0000	10	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	mul tu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1e	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	-
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22		98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt. w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	c
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	‘	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwcl	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
		c.ult.*f*	11 0101	53	35	5	117	75	u
ldcl	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
ldc2		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swcl		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	
		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdcl		c.le.*f*	11 1110	62	3e	>	126	7e	
sdc2		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) cod oper(31:26) == 0
(2) cod oper(31:26) ==17_{dec} (11_{hex}); si fmt(25:21)==16_{dec} (10_{hex}) *f*=s (simple)
si fmt(52:21)==17_{dec} (11_{hex}) *f*=d (doble)

ESTÁNDAR DE PUNTO FLOTANTE IEEE 754 ④

$(-1)^5 \times (1 + \text{Fracción}) \times 2^{(\text{exponente} - \text{sesgo})}$

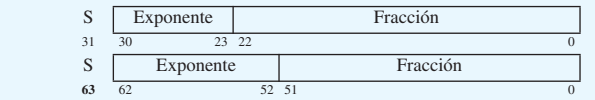
donde Sesgo de Precisión Simple = 127
Sesgo de Precisión Doble = 1023

Símbolos IEEE 754

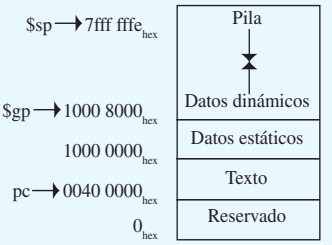
Exponente	Fracción	Objeto
0	0	±0
0	≠0	±Denormal
1 a MAX – 1	Cualquiera	±Número punto flotante
MAX	=0	±∞
MAX	≠0	NaN

S.P.MAx = 255. D.P.MAX = 2047

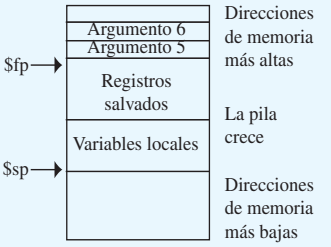
Formatos de Precisión Simple y Precisión Doble de IEEE



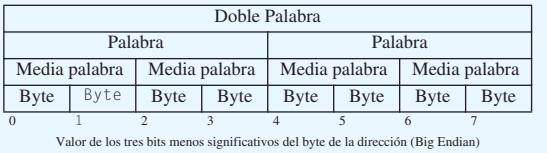
ASIGNACIÓN DE MEMORIA



CUADRO DE PILA

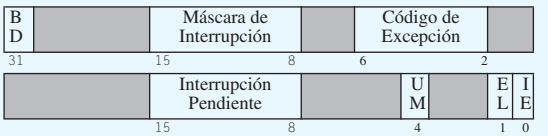


ALINEAMIENTO DE DATOS



Valor de los tres bits menos significativos del byte de la dirección (Big Endian)

REGISTROS DE CONTROL DE EXCEPCIONES: CAUSA Y ESTADO



BD: Retraso de Salto (Branch Delay); UM: Modo Usuario (User Mode); EL: Nivel de Excepción (Excepción Level); IE: Habilitación de Interrupción (Interrupt Enable)

CÓDIGOS DE EXCEPCIÓN

Nº	Nombre	Causa de la excepción	Nº	Nombre	Causa de la excepción
0	Int	Interrupción (hardware)	9	Bp	Excepción de Punto de Ruptura
4	AdEL	Excepción de Error de Dirección (carga o búsqueda de instrucción)	10	RI	Instrucción de Excepción Reservada
5	AdES	Excepción de Error de Dirección (almacenamiento)	11	CpU	Coprocesador no Implementado
6	IBE	Error de Bus en Búsqueda de Instrucción	12	Ov	Excepción de Desbordamiento Aritmético
7	DBE	Error de Bus en Carga o Almcenamiento	13	Tr	Trap
8	Sys	Excepción de Llamada al Sistema	15	FPE	Excepción de Punto Flotante

TAMAÑOS DE PREFIJOS (10x para discos y comunicaciones; 2x para memoria)

TAMAÑO	PREFIJO	TAMAÑO	PREFIJO	TAMAÑO	PREFIJO	TAMAÑO	PREFIJO
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	mili-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

El símbolo de cada prefijo es justamente su primera letra, excepto μ que se utiliza para micro.

Estructura y diseño de computadores

LA INTERFAZ SOFTWARE/HARDWARE

TRADUCCIÓN DE LA CUARTA EDICIÓN EN LENGUA INGLESA

Estructura y diseño de computadores

LA INTERFAZ SOFTWARE/HARDWARE

TRADUCCIÓN DE LA CUARTA EDICIÓN EN LENGUA INGLESA

David A. Patterson
University of California, Berkeley

John L. Hennessy
Stanford University

Con contribuciones de

Perry Alexander
The University of Kansas

Peter J. Ashenden
Ashenden Designs Pty Ltd

Javier Bruguera
Universidade de Santiago de Compostela

Jichuan Chang
Hewlett-Packard

Matthew Farrens
University of California, Davis

David Kaeli
Northeastern University

Nicole Kaiyan
University of Adelaide

David Kirk
NVIDIA

James R. Larus
Microsoft Research

Jacob Leverich
Hewlett-Packard

Kevin Lim
Hewlett-Packard

John Nickolls
NVIDIA

John Oliver
Cal Poly, San Luis Obispo

Milos Prvulovic
Georgia Tech

Partha Ranganathan
Hewlett-Packard

EDITORIAL
REVERTÉ
Barcelona - Bogotá - Buenos Aires - Caracas - México

Registro bibliográfico (ISBD)

PATTERSON, DAVID A.
[Computer Organization and Design. Español]
Estructura y diseño de computadores / David A. Patterson, John L. Hennessy; versión española por: Dr. Javier Díaz Bruguera. –Barcelona : Reverté. D.L. 2011
XXV, 703 p., [184] p. : il. col. ; 24 cm
Ed. orig.: Computer organization and design: the hardware/software interface. 4.ª ed. Burlington: Elsevier Inc., cop. 2007. – Índice.
DL B-15281-2011. – ISBN 978-84-291-2620-4
1. Estructura y diseño de computadores. I. Hennessy, John L., coaut. II. Díaz Bruguera, Javier, trad. III. Título.

Título de la obra original:

Computer Organization and Design. The Hardware / Software Interface. Fourth Edition

Edición original en lengua inglesa publicada por:

ELSEVIER INC of 200 Wheeler Road, 6th floor, Burlington, MA 01803, USA

Edición en español:

© Editorial Reverté, S. A., 2011

Edición en papel

ISBN: 978-84-291-2620-4

Edición e-book (PDF)

ISBN: 978-84-291-9418-0

Versión española por:

Prof. Dr. Javier Díaz Bruguera
Catedrático de Universidad en el área de arquitectura y tecnología de computadores
Universidad de Santiago de Compostela

Maquetación: REVERTÉ-AGUILAR, SL

Propiedad de:

EDITORIAL REVERTÉ, S. A.
Loreto, 13-15, Local B
08029 Barcelona – España
Tel: (34) 93 419 33 36
Fax: (34) 93 419 51 89
reverte@reverte.com
www.reverte.com

1361

A Linda,
que ha sido, es y siempre será el amor de mi vida

Elogios a *Computer Organization and Design. The Hardware / Software Interface, fourth edition*

"Patterson y Hennessy no sólo mejoran la pedagogía del material tradicional sobre procesadores segmentados y jerarquía de memoria, sino que también extienden enormemente la cobertura de los sistemas multiprocesador para incluir arquitecturas emergentes como los procesadores multinúcleo y las GPUs. La cuarta edición de *Estructura y Diseño de Computadores* es un nuevo punto de referencia con el cual deben compararse los restantes libros sobre arquitectura de computadores."

—David A. Wood, *University of Wisconsin-Madison*

"Patterson y Hennessy han mejorado enormemente lo que ya era el estándar de referencia de los libros de texto. En un campo en continua y rápida evolución como el de la arquitectura de los computadores, han entrelazado un número impresionante de casos reales y cuestiones actuales en un marco de fundamentos suficientemente comprobados."

—Fer Chong, *University of California at Santa Barbara*

"Desde la publicación de la primera edición en 1994, *Estructura y diseño de computadores* ha iniciado en el campo de la arquitectura de los computadores a una generación de estudiantes de informática e ingeniería. Ahora, muchos de aquellos estudiantes son los líderes en este campo. La tradición continúa en la universidad, porque los profesores utilizan las últimas ediciones del libro que los inspiró para captar a la siguiente generación. Con la cuarta edición, los lectores se preparan para la próxima era de la computación."

—David I. August, *Princeton University*

"La nueva cobertura de los sistemas multiprocesador y el paralelismo está a la altura de este excelente clásico. Los nuevos temas se introducen de forma progresiva y bien argumentada, y además se proporcionan muchos ejemplos y detalles extraídos del hardware actual."

—John Greiner, *Rice University*

"A medida que la arquitectura de los computadores se mueve de los monoprocesadores a los sistemas multinúcleo, los entornos de programación paralela usados para aprovecharlos serán un punto clave para el éxito de estos nuevos sistemas. En los sistemas multinúcleo, la interfaz entre el hardware y el software adquiere una importancias especial. Esta nueva edición de *Estructura y diseño de computadores* es obligatoria para cualquier estudiante que quiera comprender la arquitectura de los multinúcleo, incluida la interfaz entre la programación y la arquitectura."

—Jesse Fang, *Director of Programming System Lab at Intel*

"La cuarta edición de *Estructura y diseño de computadores* sigue mejorando los altos estándares de calidad de las ediciones anteriores. Los nuevos contenidos, tendencias que están cambiando los ordenadores incluyendo los multinúcleo, memorias flash, GPUs, etc., hacen de esta edición un libro que todos aquellos que hemos crecido con las anteriores ediciones debemos leer."

—Parthasarathy Ranganathan, *Principal Research Scientist, HP Labs*

AGRADECIMIENTOS

Figuras 1.7 y 1.8. Gentileza de Other World Computing (www.macsales.com).
Figuras 1.9, 1.19 y 5.37. Gentileza de AMD.
Figura 1.10. Gentileza de Storage Technology Corp.
Figuras 1.10.1, 1.10.2 y 4.15.2. Gentileza del Charles Babbage. Institute, University of Minnesota Libraries, Minneapolis.
Figuras 1.10.3, 4.15.1, 4.15.3, 5.12.3 y 6.14.2. Gentileza de IBM.
Figura 1.10.4. Gentileza de Cray Inc.
Figura 1.10.5. Gentileza de Apple Computer Inc.
Figura 1.10.6. Gentileza del Computer History Museum.
Figuras 5.12.1 y 5.12.2. Gentileza de Museum of Science. Boston.
Figura 5.12.4. Gentileza de MIPS Technologies Inc.
Figuras 6.15, 6.16 y 6.17. Gentileza de Sun Microsystems Inc.
Figura 6.4. © Peg Skorpinski.
Figura 6.14.1. Gentileza de Computer Museum of America.
Figura 6.14.3. Gentileza de Commercial Computing Museum.
Figuras 7.13.1. Gentileza de NASA Ames Research Center.

Contenido

CAPÍTULOS

Nota importante: En la presente edición en castellano, los contenidos del CD incluidos en la edición original son accesibles (en lengua inglesa) a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

CONTENIDOS EN EL CD *

Conceptos clásicos de diseño lógico C-2

Implementación del control en hardware D-2

Estudio de arquitecturas RISC para ordenadores de sobremesa, servidores y sistemas empotrados E-2

* **Nota importante:** En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Prefacio

Lo más bello que podemos experimentar es el misterio.
Es la fuente de todo el arte verdadero y la ciencia.

Albert Einstein, *What I Believe*, 1930

Sobre este libro

Creemos que la enseñanza de la ciencia e ingeniería de computadores debería reflejar el estado actual de este campo, así como introducir los principios que dan forma a la computación. También opinamos que los lectores de cualquier especialidad de la computación necesitan conocer los paradigmas de organización que determinan las capacidades, prestaciones y, en definitiva, el éxito de los sistemas informáticos.

La tecnología de los computadores moderna necesita que los profesionales de todas las especialidades de la informática conozcan el hardware y el software. La interacción entre estos dos aspectos a diferentes niveles ofrece, al mismo tiempo, un entorno para la compresión de los fundamentos de la computación. Independientemente de que su interés principal sea el hardware o el software, la informática o la electrónica, las ideas centrales de la estructura y el diseño del computador son las mismas. Por lo tanto, hemos centrado este libro en mostrar la relación entre el hardware y el software y en los conceptos básicos de los computadores actuales.

El paso reciente de los monoprocesadores a los microprocesadores multinúcleo ha confirmado la solidez de esta perspectiva, establecida ya en la primera edición. El tiempo en el que los programadores podían ignorar este aviso y confiar en los arquitectos de ordenadores, diseñadores de compiladores e ingenieros electrónicos para que sus programas se ejecutasen cada vez más rápido sin introducir cambio alguno, ha pasado. Para que los programas se ejecuten más rápido debe introducirse paralelismo. El objetivo de muchos investigadores de introducir el paralelismo sin que los programadores tengan que preocuparse de la naturaleza paralela del hardware que están programando, todavía tardará muchos años en hacerse realidad. Según nuestra visión, durante al menos la próxima década, la mayoría de los programadores van a necesitar conocer la interfaz entre el hardware y el software para que sus programas se ejecuten eficientemente en los computadores paralelos.

La audiencia de este libro incluye tanto a aquellos que, con poca experiencia en lenguaje ensamblador o diseño lógico, necesitan entender la organización básica de un computador, como a los que, con conocimientos de lenguaje ensamblador y/o diseño lógico, quieren aprender como se diseña un ordenador o entender como trabaja un sistema y por qué se comporta como lo hace.

Sobre el otro libro

Algunos lectores seguramente están familiarizados con el libro *Arquitectura de Computadores: un Enfoque Cuantitativo*, conocido popularmente como Hennessy y Patterson. (Este libro, por el contrario, a menudo se llama Patterson y Hennessy). Nuestra motivación al escribir ese libro era describir los principios de la arquitectura de computadores haciendo uso de fundamentos sólidos de ingeniería y compromisos coste/prestaciones cuantitativos. Utilizamos un enfoque que combinaba ejemplos y medidas, basados en sistemas comerciales, para crear experiencias de diseño realistas. Nuestro objetivo fue demostrar que la arquitectura de los computadores se podía aprender con una metodología cuantitativa en lugar de un enfoque descriptivo. El libro estaba dirigido a los profesionales de la informática que querían adquirir un conocimiento detallado de los computadores.

La mayoría de los lectores de este libro no tienen previsto convertirse en arquitectos de computadores. Sin embargo, las prestaciones y la eficiencia energética de los sistemas software, en el futuro, dependerán drásticamente de la adecuada comprensión de las técnicas básicas del hardware por parte de los diseñadores de software. De este modo, los diseñadores de compiladores, los diseñadores de sistemas operativos, programadores de bases de datos y la mayor parte de los ingenieros del software necesitan un conocimiento firme de los principios presentados en este libro. De manera similar, los diseñadores del hardware deben comprender claramente los efectos de su trabajo sobre las aplicaciones software.

Por lo tanto, sabíamos que este libro tenía que ser mucho más que un subconjunto del material incluido en el libro *Arquitectura de Computadores*, y el material se revisó ampliamente para adecuarse a una audiencia diferente. Quedamos tan satisfechos del resultado que se revisaron las siguientes ediciones de *Arquitectura de Computadores* para eliminar la mayor parte del material de introducción; así, hay mucho menos solape entre los dos libros ahora que en las primeras ediciones.

Cambios en la cuarta edición

Teníamos cinco objetivos principales para la cuarta edición de *Estructura y diseño de computadores*: dada la revolución multinúcleo de los microprocesadores, resaltar los aspectos paralelos del hardware y el software a lo largo del libro; racionalizar el material existente para hacer hueco a los aspectos relacionados con el paralelismo; mejorar la pedagogía en general; actualizar el contenido técnico para reflejar los cambios producidos en la industria desde la publicación de la tercera edición en 2004; restablecer la utilidad de los ejercicios en la era de internet.

Antes de discutir estos objetivos con más detalle, echemos un vistazo a la tabla de la página siguiente. Muestra los caminos del hardware y el software a lo largo del libro. Los capítulos 1, 4, 5 y 7 están en ambos caminos, sin importar la experiencia o el enfoque. El capítulo 1 es un introducción nueva que incluye una discusión sobre la importancia de la potencia y como ha alentado el paso de los sistemas con un núcleo a los microprocesadores multinúcleo. Incluye también material sobre prestaciones y evaluación mediante programas de prueba, material que en la tercera edición estaba incluido en un capítulo aparte. El capítulo 2 es más probable que esté orientado hacia el hardware, pero es una lectura esencial para los lectores orientados al software, especialmente para

Capítulo o apéndice	*Secciones*	*Enfoque software*	*Enfoque hardware*
1. Abstracciones y tecnología de los computadores	1.1 a 1.9		
	1.10 (Historia)		
2. Instrucciones: el lenguaje del computador	2.1 a 2.14		
	2.15 (Compiladores y java)		
	2.16 a 2.19		
	2.20 (Historia)		
E. Arquitecturas de repertorio de instrucciones RISC	E.1 a E.9		
3. Aritmética para computadores	3.1 a 3.9		
	3.10 (Historia)		
C. Conceptos básicos de diseño lógico	C.1 a C.13		
4. El procesador	4.1 (Visión general)		
	4.2 (Convenciones lógicas)		
	4.3 a 4.4 (Implementación simple)		
	4.5 (Introducción a segmentación)		
	4.6 (Camino de datos segmentado)		
	4.7 a 4.9 (Riesgos, excepciones)		
	4.10 a 4.11 (Paralelo, caso real)		
	4.12 (Control segmentado en Verilog)		
	4.13 a 4.14 (Falacias)		
	4.15 (Historia)		
D. Implementación del control en hardware	D.1 a D.6		
5. Grande y rápida: aprovechamiento de la jerarquía de memoria	5.1 a 5.8		
	5.9 (Controlador de cache en Verilog)		
	5.10 a 5.12		
	5.13 (Historia)		
6. Almacenamiento y otros aspectos de la E/S	6.1 a 6.10		
	6.11 (Redes)		
	6.12 a 6.13		
	6.14 (Historia)		
7. Multinúcleo, multiprocesadores y clústeres	7.1 a 7.3		
	7.14 (Historia)		
A. Unidades de procesamiento gráfico	A.1 a A.12		
B. Ensambladores, enlazadores y el simulador SPIM	B.1 a B.12		

Leer detenidamente 
Leer si se dispone de tiempo 
Referencia 
Revisar o leer 
Leer por cultura 

aquellos interesados en aprender más sobre compiladores y lenguajes de programación orientada a objetos. Incluye material del capítulo 3 de la tercera edición, de forma que toda la arquitectura MIPS, excepto las instrucciones de punto flotante, se describe ahora en un único capítulo. El capítulo 3 está dirigido a los lectores interesados en el diseño de un camino de datos o en aprender algo más sobre aritmética de punto flotante. Sin embargo, algunos lectores podrán saltarse este capítulo, bien porque no lo necesitan, bien porque es una revisión. El capítulo 4, donde se explica el procesador segmentado, es una combinación de dos capítulos de la tercera edición. Las secciones 4.1, 4.5 y 4.10 proporcionan una visión general para aquellos lectores interesados en los aspectos software. Sin embargo, constituye el material central para los interesados en los aspectos hardware; dependiendo de sus conocimientos previos, estos lectores pueden necesitar o querer leer antes el apéndice C sobre diseño lógico. El capítulo 6, sobre sistemas de almacenamiento, es de importancia crucial para los lectores con un enfoque software y los restantes lectores deberían leerlo también si disponen de tiempo suficiente. El último capítulo sobre multinúcleos, multiprocesadores y clústeres es mayoritariamente material nuevo y debería ser leído por todos.

El primer objetivo fue hacer del paralelismo un ciudadano de primera clase en esta edición, cuando en la anterior edición era un capítulo aparte en el CD. El ejemplo más evidente es el capítulo 7. En este capítulo se introduce el modelo Roofline para la evaluación de las prestaciones y se muestra su validez con la evaluación de cuatro microprocesadores multinúcleo recientes con dos núcleos computacionales. Podría demostrarse que este modelo puede ser tan intuitivo para la evaluación de los multinúcleos como el modelo de las 3Cs para las caches.

Dada la importancia del paralelismo, no sería inteligente esperar hasta el último capítulo para abordarlo, por lo tanto hay una sección sobre paralelismo en cada uno de los seis primeros capítulos:

- *Capítulo 1: Paralelismo y potencia*. Se muestra como los límites impuestos por el consumo de potencia han forzado a los principales fabricantes a mirar hacia el paralelismo, y como el paralelismo ayuda en este problema.
- *Capítulo 2: Paralelismo e instrucciones: Sincronización*. Esta sección analiza los bloqueos para variables compartidas, de forma específica las instrucciones MIPS carga enlazada y almacenamiento condicional.
- *Capítulo 3. Paralelismo y aritmética del computador: Asociatividad punto flotante*. En esta sección se analizan los retos en precisión numérica y en las operaciones punto flotante.
- *Capítulo 4. Paralelismo y paralelismo a nivel de instrucciones avanzado*. Trata el paralelismo a nivel de instrucciones (ILP) avanzado —superescalares, especulación, VLIW, desenrollamiento de lazos y OOO— así como la relación entre la profundidad del procesador segmentado y el consumo de potencia.
- *Capítulo 5. Paralelismo y jerarquías de memoria: Coherencia de cache*. Introduce los protocolos de coherencia, consistencia y fisgoneo (*snooping*) de cache.
- *Capítulo 6. Paralelismo y E/S: Conjuntos redundantes de discos económicos (RAID)*. Describe los sistemas RAID como un sistema paralelo de E/S, así como un sistema ICO de alta disponibilidad.

El capítulo 7 concluye con varias razones para ser optimista con respecto a que esta incursión en el paralelismo debería tener más éxito que las que se realizaron anteriormente.

Particularmente, estoy entusiasmado con la incorporación de un apéndice sobre Unidades de Procesamiento Gráfico escrito por el científico jefe de NVIDIA, David Kirk, y el líder del grupo de arquitectos, John Nicolls. En este sentido, el apéndice A es la primera descripción detallada de una GPU, un nuevo e interesante empuje para la arquitectura de los computadores. El apéndice articula los temas paralelos de esta edición para presentar un estilo de computación que permite al programador pensar en un sistema MIMD aunque el hardware ejecute los programas como un SIMD cuando es posible. Como las GPUs son baratas y están disponibles en casi cualquier sistema —incluso en computadores portátiles— y sus entornos de programación son de acceso libre, proporciona una plataforma hardware paralela con la que experimentar.

El segundo objetivo ha sido racionalizar el libro para hacer sitio a nuevo material sobre paralelismo. El primer paso fue mirar con lupa todos los párrafos presentes en las tres ediciones anteriores para determinar si seguían siendo necesarios. Los cambios de grano grueso fueron la combinación de capítulos y la eliminación de temas. Mark Hill sugirió eliminar la implementación multiciclo de los procesadores y añadir, en su lugar, un controlador de cache multiciclo en el capítulo dedicado a la jerarquía de memoria. Este cambio permitió presentar el procesador en un único capítulo en lugar de en los dos capítulos de ediciones anteriores, mejorando el material sobre el procesador por omisión. Los contenidos del capítulo sobre prestaciones de la tercera edición se añadieron al primer capítulo.

El tercer objetivo fue mejorar los aspectos pedagógicos del libro. El capítulo 1 es ahora más jugoso, e incluye material sobre prestaciones, circuitos integrados y consumo de potencia, y crea el marco para el resto del libro. Los capítulos 2 y 3 originalmente estaban escritos con un estilo evolutivo, comenzando con una arquitectura sencilla y terminando con la arquitectura completa del MIPS al final del capítulo 3. Sin embargo, este estilo pausado no le gusta a los lectores modernos. Por lo tanto, en esta edición se junta toda la descripción del repertorio de instrucciones enteras en el capítulo 2 —haciendo el capítulo 3 opcional para muchos lectores— y ahora cada sección es independiente, de modo que el lector no necesita ya leer todas la secciones anteriores. Así, el capítulo 2 es mejor como referencia ahora que en ediciones anteriores. El capítulo 4 está mejor organizado ahora porque se dedica un único capítulo al procesador, ya que la implementación multiciclo no es de utilidad actualmente. El capítulo 5 tiene una nueva sección sobre diseño de controladores de cache y el código Verilog para este controlador se incluye en una sección del CD.[1]

El CD-ROM que acompaña al libro, que se introdujo en la tercera edición, nos permitió reducir el coste del libro, que tenía menos páginas, y profundizar en algunos temas que eran de interés sólo para algunos lectores, pero no para todos. Desafortunadamente,

1**Nota importante:** En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

como consecuencia de nuestro entusiasmo por reducir el número de páginas del libro, el lector tenía que ir del libro al CD y del CD al libro más veces de las que le gustaría. Esto no ocurre en esta edición. Cada capítulo tiene la sección de Perspectivas históricas en el CD y, además, todos los ejercicios están en el libro impreso. Por lo tanto, en esta edición será muy raro que el lector tenga que ir del libro al CD y del CD al libro.

Para aquellos que se sorprenden de que incluyamos un CD-ROM con el libro, la respuesta es sencilla: el CD contiene material que hemos considerado que debería ser de acceso fácil e inmediato sin importar el lugar en el que se encuentre el lector. Si usted está interesado en los contenidos avanzados o le gustaría leer una guía práctica de VHDL (por ejemplo), está en el CD, lista para que la utilice. El CD-ROM incluye también una característica que debería mejorar significativamente el acceso al material: un motor de búsqueda que permite hacer búsquedas de cualquier cadena de texto en el CD o en el libro. Si está buscando algún contenido que podría no estar incluido en el índice del libro, simplemente indicando el texto que se quiere buscar el motor devuelve el número de la página en la que aparece ese texto. Esta característica es muy útil y esperamos que usted haga uso de ella frecuentemente.

Este es un campo en continua evolución, y como ocurre siempre que se lanza una nueva edición, uno de los objetivos prioritarios es actualizar los contenidos técnicos. El procesador Opteron X4 2356 (nombre en clave "Barcelona") de AMD se utiliza como ejemplo guía a lo largo de todo el libro, concretamente para los capítulos 1, 4, 5 y 7. Los capítulos 1 y 6 añaden los resultados obtenidos con los programa de prueba para la evaluación del consumo de potencia de SPEC. El capítulo 2 incorpora una sección sobre la arquitectura ARM, que actualmente es la ISA (arquitectura del repertorio de instrucciones) de 32 bits más popular en todo el mundo. El capítulo 5 tiene una sección sobre Máquinas Virtuales, que están adquiriendo una importancia creciente. En el capítulo 5 se analizan detalladamente las medidas de rendimiento de la cache en el microprocesador multinúcleo Opteron X4 y se proporcionan algunos datos, menos detallados, sobre su rival, el procesador Nehalem de Intel, que no será anunciando antes de que se publique esta edición. En el capítulo 6 se describen por primera vez las memorias Flash así como un destacado servidor de Sun, que empaqueta 8 núcleos, 16 DIMMs y 8 discos en una única unidad 1U. También se incluyen resultados recientes sobre fallos de discos a largo plazo. El capítulo 7 cubre un gran número de temas relacionados con el paralelismo —ejecución multihilo, SIMD, procesamiento vectorial, modelos de prestaciones, programas de prueba, redes de multiprocesadores, entre otros— y describe tres microprocesadores multinúcleo, además del Opteron X4: Intel Xeon e5345 (Clovertown), IBM Cell QS20 y Sun Microsystems T2 5120 (Niagara).

El objetivo final fue intentar que los ejercicios fuesen útiles a los profesores en esta era de Internet, porque los ejercicios que los estudiantes tienen que hacer en su propia casa han sido desde hace mucho tiempo una forma importante de aprender el material contenido en el libro. Desafortunadamente, hoy en día las soluciones se cuelgan en la web casi al mismo tiempo que el libro sale a la venta. Tenemos una propuesta en dos partes. Primero, colaboradores expertos han desarrollado ejercicios totalmente nuevos para todos los capítulos del libro. Segundo, la mayoría de los ejercicios tiene una descripción cualitativa apoyada en una tabla que proporciona varios parámetros cuantitativos alternativos que son necesarios para resolver las preguntas. El elevado número de ejercicios

unidos a la flexibilidad que tiene el profesor para introducir variaciones hará difícil que el estudiante pueda encontrar las soluciones en la red. Los profesores podrán cambiar estos parámetros cuantitativos como quieran, haciendo que aquellos estudiantes que confiaban en internet para encontrar las soluciones de un conjunto de ejercicios estático y fijo se sientan frustrados. Creemos que este enfoque es un nuevo valor añadido al libro; por favor, le pedimos que nos haga saber si este enfoque funciona correctamente, ¡tanto como estudiante o como profesor!

Hemos mantenido elementos útiles de las anteriores ediciones del libro. Para conseguir que el libro se mejore como libro de referencia, mantenemos las definiciones de los nuevos términos en los márgenes de las páginas en las que aparecen por primera vez. Los elementos del libro que hemos llamado secciones "Comprender las prestaciones de los programas" ayudan al lector a comprender las prestaciones de sus programas y como mejorarlas, del mismo modo que los elementos llamados "Interfaz Hardware/Software" ayudan a entender las soluciones de compromiso que se adoptan en esta interfaz. Las secciones "Idea clave" se han mantenido para que el lector vea el bosque a pesar de los árboles. Las secciones "Autoevaluación" ayudan a confirmar la comprensión del material estudiado, ya que las respuestas se incluyen al final de cada capítulo. Esta edición incluye también la tarjeta de referencia MIPS, inspirada en la "Tarjeta Verde" del IBM System/360. Esta tarjeta ha sido actualizada y debería ser una referencia accesible al escribir programas en lenguaje ensamblador MIPS.

Apoyo a los profesores

Hemos recopilado una gran cantidad de material de ayuda a los profesores que utilicen este libro en sus cursos. Tienen a su disposición soluciones a los ejercicios, puntos críticos de cada capítulo, figuras del libro, notas, dispositivas, etc., en la web de los editores

www.reverte.com/microsites/pattersonhennessy

Conclusiones finales

Si usted lee la sección de agradecimientos que viene a continuación, verá que hemos hecho un gran esfuerzo para corregir los posibles errores. Dado que un libro pasa por muchas etapas, hemos tenido la oportunidad de hacer aún más correcciones. Si usted descubre algún error que se nos ha resistido, por favor contacte con el editor a través de correo electrónico en la dirección *produccion@reverte.com* o por correo ordinario a la dirección de la página de *copyright*.

Esta edición marca una ruptura en la colaboración entre Hennessy y Patterson, que empezó en 1989 y se mantuvo durante muchos años. La oportunidad de dirigir una de las mayores universidades del mundo implica que el Rector Hennessy no podrá asumir en adelante la responsabilidad de crear una nueva edición. El otro autor se siente como un malabarista que siempre ha trabajado con un socio y de repente es empujado a actuar en solitario. En consecuencia, todos aquellos incluidos en los agradecimientos y los colegas de Berkeley han jugado un papel aun más importante en darle forma a los contenidos de este libro. Sin embargo, esta vez hay un único autor al que culpar por el nuevo material que va a leer.

Agradecimientos de la cuarta edición

Me gustaría dar las gracias a **David Kirk**, **John Nicholls** y sus colegas de NVIDIA (Michael Garland, John Montrym, Dough Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman y Vasily Volkov) por escribir el primer apéndice que analiza en profundidad las GPUs. Quiero expresar otra vez mi agradecimientoa **Jim Larus** de Microsoft Research por su buena disposición a contribuir como experto en programación en lenguaje ensamblador y por alentar a los lectores de este libro a utilizar el simulador que él mismo desarrolló y mantiene.

También estoy muy agradecido por las contribuciones de muchos expertos que desarrollaron los nuevos ejercicio de esta edición. Escribir buenos ejercicios no es una tarea fácil y todos los colaboradores tuvieron que trabajar mucho y duro para plantear unos problemas exigentes y atractivos.

- *Capítulo 1*: **Javier Bruguera** (Universidade de Santiago de Compostela)
- *Capítulo 2*: **John Oliver** (Cal Poly, San Luis Obispo), con contribuciones de **Nicole Kaiyan** (University of Adelaide) y **Milos Prvulovic** (Georgia Tech)
- *Capítulo 3*: **Matthew Farrens** (University of California, Davis)
- *Capítulo 4*: **Milos Prvulovic** (Georgia Tech)
- *Capítulo 5*: **Jichuan Chang**, **Jacob Leverich**, **Kevin Li**m y **Partha Ranganathan** (todos de Hewlett-Packard), con contribuciones de Nicole Kaiyan (University of Adelaide)
- *Capítulo 6*: **Perry Alexander** (The University of Kansas)
- *Capítulo 7*: **David Kaeli** (Northeastern University)

Peter Asheden llevó a cabo la hercúlea tarea de editar y evaluar *todos* los nuevos ejercicios. Además, incluso asumió la elaboración del CD que acompaña al libro y de las diapossitivas para las clases.

Gracias a **David August** y **Prakash Prabhu** de Princenton University por su trabajo sobre los concursos de los capítulos que están disponibles para los profesores en la Web de los editores.

Conté con el apoyo de mis colegas del Silicon Valley para la elaboración de gran parte del material técnico del libro:

- **AMD**, por los detalles y medidas del Opteron X4 (Barcelona): **William Brantley**, **Vasileios Liaskovitis**, **Chuck Moore** y **Brian Waldecker**.
- **Intel**, por la información prelanzamiento del Intel Nehalem: **Faye Briggs**.
- **Micron**, por la información sobre memorias Flash del capítulo 6: **Dean Klein**.
- **Sun Microsystems**, por las medidas de mezclas de instrucciones de los programas de prueba SPEC2006 en el capítulo 2 y los detalles y medidas del Sun Server x4150 en el capítulo 6: **Yan Fisher**, **John Fowler**, **Darryl Gove**, **Paul Joyce**, **Shenik Mehta**, **Pierre Reynes**, **Dimitry Stuve**, **Durgam Vahia** y **David Weaver**.

- **U.C. Berkeley**, **Krste Asanovic** (que proporcionó la idea para la concurrencia software frente a paralelismo hardware del capítulo 7), **James Demmel** y **Velvel Kahan** (con sus comentarios sobre paralelismo y cálculos punto flotante), **Zhangxi Tan** (que diseñó en controlador de cache y escribió el código Verilog para el mismo en el capítulo 5), **Sam Williams** (que proporcionó el modelo Roofline y las medidas para los multinúcleos en el capítulo 7) y el resto de mis colegas en el **Par Lab** que hicieron sugerencias y comentarios para los distintos aspectos del paralelismo que se pueden encontrar en este libro.

Estoy agradecido a los muchos profesores que contestaron los cuestionarios enviados por los editores, revisaron nuestras propuestas y asistieron a los grupos de discusión para analizar y responder a nuestros planes para esta edición. Esto incluye a las siguientes personas: *Grupo de discusión*: Mark Hill (University of Wisconsin, Madison), E.J. Kim (Texas A&M University), Jihong Kim (Seoul National University), Lu Peng (Louisiana State University), Dean Tullsen (UC San Diego), Ken Vollmar (Missouri State University), David Wood (University of Wisconsin, Madison), Ki Hwan Yum (University of Texas, San Antonio); *Revisiones*: Mahmoud Abou-Nasr (Wayne State University), Perry Alexander (The University of Kansas), Hakan Aydin (George Mason University), Hussein Badr (State University of New York at Stony Brook), Mac Baker (Virginia Military Institute), Ron Barnes (George Mason University), Douglas Blough (Georgia Institute of Technology), Kevin Bolding (Seattle Pacifi c University), Miodrag Bolic (University of Ottawa), John Bonomo (Westminster College), Jeff Braun (Montana Tech), Tom Briggs (Shippensburg University), Scott Burgess (Humboldt State University), Fazli Can (Bilkent University), Warren R. Carithers (Rochester Institute of Technology), Bruce Carlton (Mesa Community College), Nicholas Carter (University of Illinois at Urbana-Champaign), Anthony Cocchi (The City University of New York), Don Cooley (Utah State University), Robert D. Cupper (Allegheny College), Edward W. Davis (North Carolina State University), Nathaniel J. Davis (Air Force Institute of Technology), Molisa Derk (Oklahoma City University), Derek Eager (University of Saskatchewan), Ernest Ferguson (Northwest Missouri State University), Rhonda Kay Gaede (The University of Alabama), Etienne M. Gagnon (UQAM), Costa Gerousis (Christopher Newport University), Paul Gillard (Memorial University of Newfoundland), Michael Goldweber (Xavier University), Georgia Grant (College of San Mateo), Merrill Hall (The Master's College), Tyson Hall (Southern Adventist University), Ed Harcourt (Lawrence University), Justin E. Harlow (University of South Florida), Paul F. Hemler (Hampden-Sydney College), Martin Herbordt (Boston University), Steve J. Hodges (Cabrillo College), Kenneth Hopkinson (Cornell University), Dalton Hunkins (St. Bonaventure University), Baback Izadi (State University of New York—New Paltz), Reza Jafari, Robert W. Johnson (Colorado Technical University), Bharat Joshi (University of North Carolina, Charlotte), Nagarajan Kandasamy (Drexel University), Rajiv Kapadia, Ryan Kastner (University of California, Santa Barbara), Jim Kirk (Union University), Geoffrey S. Knauth (Lycoming College), Manish M. Kochhal (Wayne State), Suzan Koknar-Tezel (Saint Joseph's University), Angkul Kongmunvattana (Columbus State University), April Kontostathis (Ursinus

College), Christos Kozyrakis (Stanford University), Danny Krizanc (Wesleyan University), Ashok Kumar, S. Kumar (The University of Texas), Robert N. Lea (University of Houston), Baoxin Li (Arizona State University), Li Liao (University of Delaware), Gary Livingston (University of Massachusetts), Michael Lyle, Douglas W. Lynn (Oregon Institute of Technology), Yashwant K Malaiya (Colorado State University), Bill Mark (University of Texas at Austin), Ananda Mondal (Clafl in University), Alvin Moser (Seattle University), Walid Najjar (University of California, Riverside), Danial J. Neebel (Loras College), John Nestor (Lafayette College), Joe Oldham (Centre College), Timour Paltashev, James Parkerson (University of Arkansas), Shaunak Pawagi (SUNY at Stony Brook), Steve Pearce, Ted Pedersen (University of Minnesota), Gregory D Peterson (The University of Tennessee), Dejan Raskovic (University of Alaska, Fairbanks) Brad Richards (University of Puget Sound), Roman Rozanov, Louis Rubinfi eld (Villanova University), Md Abdus Salam (Southern University), Augustine Samba (Kent State University), Robert Schaefer (Daniel Webster College), Carolyn J. C. Schauble (Colorado State University), Keith Schubert (CSU San Bernardino), William L. Schultz, Kelly Shaw (University of Richmond), Shahram Shirani (McMaster University), Scott Sigman (Drury University), Bruce Smith, David Smith, Jeff W. Smith (University of Georgia, Athens), Philip Snyder (Johns Hopkins University), Alex Sprintson (Texas A&M), Timothy D. Stanley (Brigham Young University), Dean Stevens (Morningside College), Nozar Tabrizi (Kettering University), Yuval Tamir (UCLA), Alexander Taubin (Boston University), Will Thacker (Winthrop University), Mithuna Thottethodi (Purdue University), Manghui Tu (Southern Utah University), Rama Viswanathan (Beloit College), Guoping Wang (Indiana-Purdue University), Patricia Wenner (Bucknell University), Kent Wilken (University of California, Davis), David Wolfe (Gustavus Adolphus College), David Wood (University of Wisconsin, Madison), Mohamed Zahran (City College of New York), Gerald D. Zarnett (Ryerson University), Nian Zhang (South Dakota School of Mines & Technology), Jiling Zhong (Troy University), Huiyang Zhou (The University of Central Florida), Weiyu Zhu (Illinois Wesleyan University).

Me gustaría dar las gracias especialmente a la gente de Berkeley que dio una retroalimentación clave para el capítulo 7 y el apéndice A, que fueron las dos partes de este libro más complicadas de escribir: **Krste Asanovic**, **Christopher Batten**, **Rastilav Bodik**, **Bryan Catanzaro**, **Jike Chong**, **Kaushik Data**, **Greg Giebling**, **Anik Jain**, **Jae Lee**, **Vasily Volkov** y **Samuel Williams**.

Un agradecimiento especial también para **Mark Smotherman** por sus múltiples revisiones para encontrar errores técnicos y de escritura, que contribuyeron a mejorar significativamente la calidad de esta edición. Su papel ha sido incluso más importante esta vez puesto que esta edición se hizo en solitario.

Queremos agradecer a la familia de Morgan Kaufmann por estar de acuerdo en la publicación de este libro otra vez bajo la dirección de **Denise Penrose**. **Nathaniel McFadden** fue el editor de desarrollo (developmental editor) para esta edición y trabajó codo con codo conmigo en los contenidos del libro. **Kimberlee Honjo** coordinó las encuestas de los usuarios y sus respuestas.

Dawnmarie Simpson llevó a cabo la supervisión del proceso de producción del libro. También queremos dar las gracias a muchos agentes por cuenta propia (*fre-*

elance vendors) que contribuyeron a que este volumen vea la luz, especialmente a Alan Rose de Multiscience Press y diacriTech, nuestro corrector (compositor).

Las contribuciones de las casi 200 personas mencionadas aquí han ayudado a hacer de esta cuarta edición lo que espero sea nuestro mejor libro hasta el momento. ¡Gracias a todos!

David A. Patterson

1

Abstracciones y tecnología de los computadores

La civilización avanza extendiendo el número de operaciones importantes que se pueden hacer sin pensar en ellas

Alfred North Whitehead
An Introduction to Mathematics, 1911

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

1.1 Introducción

¡Bienvenido a este libro! Estamos encantados de tener esta oportunidad de transmitir el entusiasmo del mundo de los computadores. Éste no es un campo árido y aburrido, donde el progreso es glacial y donde las nuevas ideas se atrofian por negligencia. ¡No! Los computadores son el producto de la increíblemente vibrante industria de las tecnologías de la información, que en su conjunto es responsable como mínimo del 10% del producto nacional bruto de los Estados Unidos y cuya economía se ha vuelto, en parte, dependiente de las rápidas mejoras en tecnologías de la información prometidas por la ley de Moore. Esta insólita industria impulsa la innovación a una velocidad asombrosa. En los últimos 25 años aparecieron varios computadores nuevos cuya introducción parecía que iba a revolucionar la industria de la computación; estas revoluciones duraban poco tiempo simplemente porque alguien construía un computador aun mejor.

Esta carrera por innovar condujo a un progreso sin precedentes desde el inicio de la computación electrónica en los últimos años de la década de 1940. Si los medios de transporte hubiesen ido a la par con la industria de la computación, por ejemplo, hoy en día se podría viajar de Nueva York a Londres en aproximadamente un segundo por unos pocos céntimos. Pensemos un momento cómo tal adelanto habría cambiado la sociedad (vivir en Tahití y trabajar en San Francisco, ir a Moscú para ver el Ballet Bolshoi) y podremos apreciar las implicaciones de tal cambio.

Los computadores nos han llevado a una tercera revolución de la civilización, la revolución de la información, que se sitúa a la par de las revoluciones agrícola e industrial. El resultado de la multiplicación de la potencia y el alcance intelectual de la humanidad ha afectado profundamente a nuestras vidas cotidianas y también ha cambiado la manera de obtener nuevos conocimientos. Hay ahora una nueva forma de investigación científica, en la cual científicos informáticos trabajan junto a científicos teóricos y experimentales en la exploración de nuevas fronteras en astronomía, biología, química y física entre otras.

La revolución de los computadores continúa. Cada vez que el coste de la computación se mejora en un factor 10, las oportunidades para los computadores se multiplican. Aplicaciones que eran económicamente inviables repentinamente se convierten en factibles. Hasta hace muy poco, las siguientes aplicaciones eran "ciencia ficción computacional".

- *Computadores en los coches:* hasta que los microprocesadores mejoraron drásticamente en precio y prestaciones a principios de la década de 1980, el control por computador en los coches era risible. Hoy en día los computadores reducen la contaminación y mejoran las prestaciones del combustible vía controles en el motor, y también mejoran la seguridad al prevenir peligrosos patinazos e inflando los *air bags* para proteger a los ocupantes en caso de colisión.
- *Teléfonos portátiles*: ¿quién habría podido soñar que los avances en sistemas de computación llevarían al desarrollo de teléfonos móviles, permitiendo una comunicación persona-a-persona casi en cualquier parte del mundo?
- *Proyecto genoma humano:* el coste del equipamiento informático para cartografiar y analizar las secuencias del ADN humano es de centenares de millones de dólares. Es improbable que alguien hubiese considerado este proyecto si los costes de los computadores hubiesen sido entre 10 y 100 veces mayores, tal y como eran hace 10 o 20 años. Además, el coste continúa bajando; podríamos ser capaces de adquirir nuestro propio genoma, permitiendo que los cuidados médicos se adapten a nosotros.
- *World Wide Web (La telaraña mundial):* la World Wide Web, que no existía en la primera edición de este libro, ha cambiado nuestra sociedad. Para muchos, la www ha reemplazado a las bibliotecas.
- *Motores de búsqueda:* Dado que el contenido de la WWW ha crecido en tamaño y valor, encontrar información relevante es cada vez más importante. En la actualidad, mucha gente confía tanto en los motores de búsqueda que se verían en apuros si no pudiesen utilizarlos.

Claramente, los avances en esta tecnología influyen hoy en día en casi todos los aspectos de nuestra sociedad. Los avances en el hardware han permitido a los programadores crear programas maravillosamente útiles, y explican por qué los computadores son omnipresentes. Lo que hoy es ciencia ficción, serán las aplicaciones normales en el futuro: ya hay mundos virtuales, reconocimiento de voz y asistencia médica personalizada.

Tipos de aplicaciones de computador y sus características

Aunque se usan un conjunto común de tecnologías hardware (presentadas en las secciones 1.3 y 1.4) que van desde los electrodomésticos caseros inteligentes a los teléfonos móviles o celulares o los mayores supercomputadores, estas aplicaciones diferentes tienen diversos requerimientos de diseño y utilizan las tecnologías hardware de manera diferente. Grosso modo, los computadores se utilizan en tres clases diferentes de aplicaciones.

Los **computadores de sobremesa** son posiblemente la forma más conocida de computación y están representados por el computador personal, que muchos lectores de este libro habrán usado extensamente. Los computadores de sobremesa se caracterizan por dar buenas prestaciones a bajo coste a único usuario, y a menudo se usan para ejecutar programas de terceros, también llamado software estándar. La evolución de muchas tecnologías es impulsada por este tipo de computadores, ¡que sólo tiene 30 años de antigüedad!

Computador de sobremesa: computador diseñado para un único usuario, y que incorpora una pantalla, un teclado y un ratón.

Los **servidores** son la versión moderna de lo que fueron los computadores centrales, los minicomputadores y los supercomputadores, y generalmente solo se accede a ellos vía una red. Los servidores están pensados para soportar grandes cargas de trabajo, que pueden consistir en una única aplicación compleja, generalmente científica o de ingeniería, o en muchos trabajos pequeños, como ocurre en un servidor Web. Estas aplicaciones están basadas en programas de otras fuentes (tales como una base de datos o un sistema de simulación), pero frecuentemente se modifican o adaptan a una función concreta. Los servidores se construyen con la misma tecnología básica que los computadores de sobremesa, pero permiten una mayor ampliación de su capacidad tanto de computación como de entrada/salida. En general, los servidores también ponen gran énfasis en la confiabilidad, puesto que un fallo es generalmente más costoso que en un computador de sobremesa de un único usuario.

Servidor: computador que se utiliza para ejecutar grandes programas para muchos usuarios, a menudo simultáneamente; generalmente sólo se accede a él vía un red.

Los servidores abarcan la mayor gama de costes y capacidades. En el extremo más bajo, un servidor puede ser un poco más que una máquina de sobremesa sin pantalla ni teclado y con un coste un poco mayor. Estos servidores de gama baja se usan típicamente para almacenar archivos o ejecutar pequeñas aplicaciones de empresas o un servicio web sencillo (véase la sección 6.10). En el otro extremo están los **supercomputadores**, que en la actualidad disponen de cientos de miles de procesadores, y generalmente **terabytes** de memoria y **petabytes** de almacenamiento, y cuestan de millones a cientos de millones de dólares. Los supercomputadores se utilizan para cálculos científicos y de ingeniería de alta calidad, tales como predicción del clima, prospección petrolífera, determinación de la estructura de proteínas, y otros problemas de gran envergadura. Aunque estos supercomputadores representan el pico de la capacidad de computación, en términos relativos constituyen una pequeña fracción del número total de servidores y también del total del mercado de los computadores en términos de facturación.

Supercomputador: computador con la capacidad de computación y coste más altos; se configuran como servidores y generalmente su coste es de millones de dólares.

Terabyte: originalmente son 1 099 511 627 776 (2^{40}) bytes, aunque algunos sistemas de comunicaciones y de almacenamiento secundario lo han redefinido como 1 000 000 000 000 (10^{12}) bytes.

Petabyte: 1000 o 1024 terabytes.

Aunque no se les llama supercomputadores, los **centros de datos** de internet utilizados por compañías como eBay y Google disponen también de miles de procesadores, terabytes de memoria y petabytes de almacenamiento. Habitualmente se consideran como grandes clústeres de computadores (véase capítulo 7).

Centro de datos: una habitación o edificio diseñado con todo lo que se necesita para un número elevado de servidores: alimentación y potencia eléctrica, aire acondicionado y red.

Computadores empotrados: computador que se encuentra dentro de otro dispositivo y que se utiliza para ejecutar una aplicación predeterminada o un conjunto de aplicaciones relacionadas.

Los **computadores empotrados** constituyen la clase de computadores más amplia y son los que tienen la gama más amplia de aplicaciones y prestaciones. Los computadores empotrados incluyen los microprocesadores que se encuentran en los coches, los computadores de los teléfonos móviles o de los computadores de los videojuegos o de los televisores digitales, y las redes de procesadores que controlan los aviones modernos o los barcos de carga. Los sistemas de computación empotrada se diseñan para ejecutar una aplicación o un conjunto de aplicaciones relacionadas, que normalmente están integradas con el hardware y se proporcionan como un único sistema; así, a pesar del gran número de computadores empotrados, ¡muchos usuarios nunca ven realmente que están usando un computador!

La figura 1.1. muestra que durante los últimos años el incremento de los teléfonos móviles, que dependen de los computadores empotrados, ha sido mucho más rápido que el incremento de los computadores de sobremesa. Obsérvese que las televisiones digitales, coches, cámaras digitales, reproductores de música, videojuegos y otros muchos dispositivos de consumo incorporan también computadores empotrados, lo que incrementa aún más la diferencia entre el número de computadores empotrados y computadores de sobremesa.

Las aplicaciones empotradas a menudo tienen un único requisito de aplicación que combina unas prestaciones mínimas con fuertes limitaciones en coste o consumo de potencia. Por ejemplo, pensemos en un reproductor de música: El procesador necesita solamente tener la velocidad suficiente para llevar a cabo esta función limitada, y a partir de aquí, los objetivos más importantes son reducir el coste y el consumo de potencia. A pesar de su coste reducido, los computadores

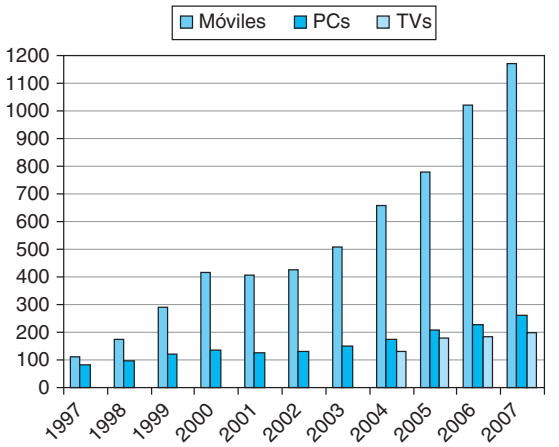


FIGURA 1.1 El número de teléfonos móviles, computadores personales y televisores fabricados cada año entre 1997 y 2007. (Solo hay datos de televisores del año 2004). Más de mil millones de nuevos teléfonos móviles se vendieron en 2006. En 1997, la venta de teléfonos móviles superaba a la de PC en un factor 1.4; y este factor creció hasta 4.5 en 2007. En 2004, se estimaba que había 2000 millones de televisores en uso, 1800 millones de teléfonos móviles y 800 millones de PCs. Cómo la población mundial era 6400 millones de personas, había aproximadamente 1 PC, 2.2 teléfonos móviles y 2.5 televisores por cada 8 habitantes del planeta. En 2006, un estudio estimó que en Estados Unidos había una media de 12 dispositivos por familia, incluyendo 3 televisiones, 2 PCs y otros aparatos como consolas para videojuegos, reproductores de MP3 y teléfonos móviles.

empotrados tienen a menudo menor tolerancia a fallos, porque las consecuencias de un fallo pueden variar desde ser molestas (cuando nuestra televisión nueva se estropea) a devastadoras (algo que puede ocurrir cuando falla el computador de un avión o de un barco de carga). En aplicaciones empotradas orientadas al consumo, como por ejemplo electrodomésticos digitales, la funcionalidad se obtiene principalmente a través de la sencillez; se intenta hacer una función tan perfectamente como sea posible. En grandes sistemas empotrados, se emplean a menudo técnicas de redundancia desarrolladas para el mundo de los servidores (véase sección 6.9). Aunque este libro se centra en los computadores de propósito general, la mayoría de los conceptos son aplicables directamente, o con pocas modificaciones, a los computadores empotrados.

Extensión: Las extensiones son secciones cortas que se usan a lo largo del texto para proporcionar un mayor detalle sobre algún tema en particular, que puede ser de interés. Los lectores que no están interesados en una extensión, pueden saltársela, ya que el material siguiente nunca dependerá de su contenido.

Muchos procesadores empotrados se diseñan usando *núcleos de procesador*, una versión de un procesador escrito en un lenguaje de descripción hardware, tal como Verilog o VHDL (véase capítulo 4). El núcleo permite a un diseñador integrar otro hardware específico de la aplicación con el núcleo del procesador para fabricar un único circuito integrado.

Qué puede aprender en este libro

Los programadores de éxito siempre han estado comprometidos con las prestaciones de sus programas, porque proporcionar de forma rápida resultados al usuario es fundamental para crear software de éxito. En las décadas de 1960 y 1970, la principal restricción para las prestaciones era el tamaño de la memoria del computador. Así, los programadores frecuentemente seguían una sencilla regla: minimizar el espacio de la memoria para hacer programas rápidos. En la última década, los avances en el diseño de los computadores y en la tecnología de la memoria redujeron drásticamente la importancia del pequeño tamaño de la memoria en muchas aplicaciones, excepto las de los sistemas de computación empotrada.

Los programadores interesados en las prestaciones necesitan ahora conocer las cuestiones que han reemplazado el modelo de memoria simple de la década de 1960: la naturaleza jerárquica de las memorias y la naturaleza paralela de los procesadores. Los programadores que buscan construir versiones competitivas de los compiladores, los sistemas operativos, las bases de datos, e incluso de aplicaciones tendrán consecuentemente que incrementar su conocimiento de la organización de los computadores.

Nosotros tenemos la oportunidad de explicar qué hay dentro de esta máquina revolucionaria, desentrañando el software bajo su programa y el hardware bajo las cubiertas de su computador. Cuando complete este libro, creemos que será capaz de contestar a las siguientes preguntas:

- ¿Cómo se escriben los programas en un lenguaje de alto nivel, tales como C o Java, cómo se traducen al lenguaje del hardware, y cómo ejecuta el hardware

el programa resultante? Comprender estos conceptos forma la base para entender los aspectos tanto del hardware y como del software que afectan a las prestaciones de los programas.

- ¿Cuál es la interfaz entre el software y el hardware, y cómo el software instruye al hardware para realizar las funciones necesarias? Estos conceptos son vitales para comprender cómo se escriben muchos tipos de software.
- ¿Qué determina las prestaciones de un programa, y cómo un programador puede mejorarlo? Como veremos, esto depende del programa original, el software de traducción de este programa al lenguaje del computador, y de la efectividad del hardware al ejecutar el programa.
- ¿Qué técnicas pueden usar los diseñadores de hardware para mejorar las prestaciones? Este libro introducirá los conceptos básicos del diseño de los computadores actuales. El lector interesado encontrará mucho más material sobre este tema en nuestro libro avanzado, *Arquitectura del Computador: Una aproximación cuantitativa*.
- ¿Cuáles son las razones y consecuencias del reciente paso del procesamiento secuencial al paralelo? En este libro se indican las motivaciones, se describe el hardware disponible actualmente para dar soporte al paralelismo y se revisa la nueva generación de **microprocesadores multinúcleo** (véase capítulo 7).

Microprocesador multinúcleo: procesador que contiene varios procesadores o núcleos en un único circuito integrado.

Sin comprender las respuestas a estas preguntas, mejorar las prestaciones de sus programas en un computador moderno, o evaluar qué características podrían hacer un computador mejor que otro para una aplicación particular, sería un complejo proceso de prueba y error, en lugar de un procedimiento científico conducido por la comprensión y el análisis.

El primer capítulo contiene los fundamentos para el resto del libro. Introduce las ideas básicas y las definiciones, pone en perspectiva los mayores componentes del hardware y del software, muestra como evaluar prestaciones y potencia e introduce los circuitos integrados, la tecnología que alimenta la revolución de los computadores y explica la evolución hacia los multinúcleos.

En este capítulo, y otros posteriores, probablemente usted verá un montón de nuevas palabras, o palabras que puede haber oído, pero que no está seguro de lo que significan. ¡No se preocupe! Sí, hay un montón de terminología especial que se usa en la descripción de los computadores modernos, pero la terminología realmente es una ayuda que nos permite describir con precisión una función o capacidad. Además, a los diseñadores de computadores (incluidos a los autores de este libro) les encanta usar **acrónimos**, ¡que son más fáciles de comprender cuando se sabe lo que significa cada una de sus letras! Para ayudarle a recordar y localizar los términos, hemos incluido una definición resaltada de cada uno de ellos la primera vez que aparece en el texto. Después de un breve periodo trabajando con la terminología, usted estará habituado, y sus amigos se quedarán impresionados de la manera tan correcta como usa palabras como BIOS, CPU, DIMM, DRAM, PCIE, SATA y tantas otras.

Acrónimo: palabra construida tomando las letras iniciales de cada una de las palabras de una frase; por ejemplo, RAM es un acrónimo de *Random Access Memory* (memoria de acceso aleatorio), y CPU es un acrónimo de for *Central Processing Unit* (unidad central de proceso).

Para reforzar la comprensión de cómo los sistemas hardware y software que se usan para ejecutar un programa afectan a las prestaciones, a lo largo del libro usamos la sección especial "Comprender las prestaciones de los programas". A

continuación veremos la primera. Estas secciones resumen detalles importantes de las prestaciones de los programas.

Comprender las prestaciones de los programas

Las prestaciones de un programa dependen de una combinación de efectividad de los algoritmos usados en el programa, de los sistemas de software usados para crear y traducir el programa en instrucciones máquina y de la efectividad del computador al ejecutar esas instrucciones, las cuales pueden incluir operaciones de entrada y salida (E/S). La siguiente tabla resume cómo afectan a las prestaciones tanto el hardware como el software.

Componente Hardware o software	Cómo afecta este componente a las prestaciones	Dónde se cubre este tema
Algoritmo	Determina el número de sentencias de alto nivel y el número de operaciones de E/S que se ejecutarán	¡Otros libros!
Lenguaje de programación, compilador y arquitectura	Determina el número de instrucciones máquina que se ejecutarán por cada sentencia de alto nivel	capítulos 2 y 3
Procesador y sistema de memoria	Determina cuán rápido se pueden ejecutar las instrucciones	capítulos 4, 5 y 7
Sistema de E/S (hardware y sistema operativo)	Determina cuán rápido se pueden ejecutar las operaciones de E/S	capítulo 6

Autoevaluación

Las secciones "Autoevaluación" se han diseñado para ayudar a los lectores a valorar si han comprendido los conceptos principales introducidos en un capítulo y han entendido las implicaciones de los mismos. Algunas preguntas de estas secciones "Autoevaluación" tienen respuestas sencillas; otras son para debate en grupo. Las respuestas a las cuestiones específicas se pueden encontrar al final del capítulo. Las preguntas de "Autoevaluación" aparecen sólo al final de una sección, de manera que es fácil saltarlas si usted está seguro de que entiende el material.

1. La sección 1.1 mostraba que el número de procesadores empotrados vendidos cada año supera significativamente el número de procesadores de computadores de sobremesa. ¿Puede confirmar o negar esta percepción basándose en su propia experiencia? Intente contar el número de procesadores empotrados que hay en su casa. ¿Cómo es este número comparado con el número de computadores de sobremesa que hay en su casa?
2. Tal y como se mencionó antes, tanto el hardware como el software afectan a las prestaciones de un programa. ¿Puede pensar ejemplos dónde cada uno de los ítems siguientes sea el lugar adecuado para buscar un cuello de botella de las prestaciones?
   - El algoritmo elegido
   - El lenguaje de programación o el compilador
   - El sistema operativo
   - El procesador
   - El sistema de E/S y los dispositivos

En París sólo me miraban fijamente cuando les hablaba en francés; nunca conseguí hacer entender a esos idiotas su propio idioma.

Mark Twain, *The Innocents Abroad*, 1869

1.2 Bajo los programas

Una aplicación típica, tal como un procesador de textos o un gran sistema de base de datos, puede consistir de cientos de miles o millones de líneas de código y depender de sofisticadas bibliotecas de software que implementan funciones complejas de soporte a la aplicación. Tal y como veremos, el hardware de un computador sólo puede ejecutar instrucciones extremadamente simples de bajo nivel. Para ir de una aplicación compleja hasta las instrucciones simples se ven involucradas varias capas de software que interpretan o trasladan las operaciones de alto nivel en instrucciones simples del computador.

Estas capas de software están organizadas principalmente en forma jerárquica, donde las aplicaciones son el anillo más externo y una variedad de **software de sistemas** se coloca entre el hardware y las aplicaciones software, como muestra la figura 1.2.

Software de sistemas: software que proporciona servicios que habitualmente son útiles, entre ellos los sistemas operativos, los compiladores y los ensambladores.

Hay muchos tipos de software de sistemas, pero actualmente hay dos tipos que son fundamentales para todos los computadores: un sistema operativo y un compilador. Un **sistema operativo** interactúa entre el programa del usuario y el hardware y proporciona una variedad de servicios y funciones de supervisión. Entre sus funciones más importantes están:

Sistema operativo: programa de supervisión que gestiona los recursos de un computador para provecho de los programas que se ejecutan en esa máquina.

- manejo de las operaciones básicas de entrada y salida
- asignación de espacio de almacenamiento y de memoria
- facilitar la compartición del computador entre múltiples aplicaciones simultáneas

Ejemplos de sistemas operativos en uso hoy en día son Windows, Linux y MacOS.

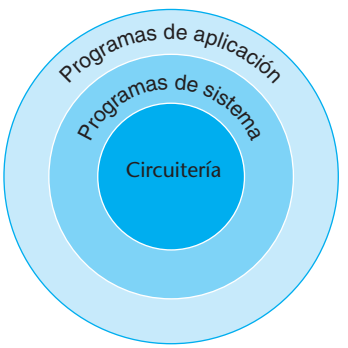


FIGURA 1.2 Vista simplificada del hardware y el software como capas jerárquicas, mostradas como círculos concéntricos con el hardware en el centro y el software de las aplicaciones en el exterior. En aplicaciones complejas frecuentemente se encuentran múltiples capas software. Por ejemplo, un sistema de base de datos puede ejecutarse sobre el software de sistemas que aloja una aplicación, el cual a su vez se ejecuta sobre la base de datos.

Los **compiladores** realizan otra función vital: la traducción de un programa escrito en un lenguaje de alto nivel, como C, C++, Java o Visual Basic a instrucciones que el hardware puede ejecutar. Dada la sofisticación de los modernos lenguajes de programación y las instrucciones simples ejecutadas por el hardware, la traducción desde un programa en un lenguaje de alto nivel a instrucciones hardware es compleja. Daremos una breve visión general del proceso y volveremos a este tema en el capítulo 2 y apéndice B.

Compiladores: programa que traduce sentencias en un lenguaje de alto nivel a sentencias en lenguaje ensamblador.

Del lenguaje de alto nivel al lenguaje del hardware

Para hablar realmente a una máquina electrónica es necesario enviar señales eléctricas. Las señales eléctricas más fáciles de entender para las máquinas son encendido (*on*) y apagado *(off),* y por lo tanto el alfabeto de la máquina tiene sólo dos letras. Del mismo modo que las 26 letras del alfabeto inglés no limitan cuánto se puede escribir, las dos letras del alfabeto de los computadores no limitan lo que los éstos pueden hacer. Los dos símbolos para estas letras son los números 0 y 1 y habitualmente pensamos en el lenguaje de las máquinas como números en base 2, o números binarios. Nos referimos a cada "letra" como **dígito binario** o **bit**.[1] Los computadores son esclavos de nuestras órdenes. De ahí que el nombre para una orden individual sea **instrucción**. Las instrucciones, que son simples colecciones de bits que el computador puede comprender, se pueden pensar como números. Por ejemplo, los bits

Dígito binario: también llamado bit. Uno de los dos números en base 2 (0 ó 1) que son los componentes de la información.

Instrucción: orden que el hardware del computador entiende y obedece.

```
1000110010100000
```

indican a un computador que sume dos números. En el capítulo 3 explicamos por qué usamos números para instrucciones *y* datos; no queremos adelantar acontecimientos, pero el uso de números tanto para las instrucciones como para los datos es de importancia capital para la informática.

Los primeros programadores se comunicaban con los computadores mediante números binarios, pero eso era tan laborioso que rápidamente inventaron nuevas notaciones más próximas a la forma de pensar de los humanos. Al principio estas notaciones se traducían a binario a mano, pero ese proceso aún era fatigoso. Usando la propia máquina para ayudar a programar la máquina, los pioneros inventaron programas para traducir de notación simbólica a binario. El primero de estos programas fue llamado **ensamblador**. Este programa traduce la versión simbólica de una instrucción a su versión binaria. Por ejemplo, el programador escribiría

Ensamblador: programa que traduce una versión simbólica de las instrucciones a su versión binaria.

```
add A, B
```

y el ensamblador traduciría esta notación a

Lenguaje ensamblador: representación simbólica de las instrucciones de la máquina.

```
1000110010100000
```

Esta instrucción indica al computador que sume los números A y B. El nombre acuñado para este lenguaje simbólico, aún usado hoy en día, es **lenguaje ensamblador**. Por el contrario, el lenguaje binario que entiende el computador se llama **lenguaje máquina**.

Lenguaje máquina: representación binaria de las instrucciones máquina.

1. Originariamente, contracción inglesa de *binary digit.* (N. del T.)

Aunque sea una enorme mejora, el lenguaje ensamblador todavía queda muy lejos de la notación que le gustaría usar a un científico para simular el flujo de fluidos o la que podría usar un contable para hacer balance de sus cuentas. El lenguaje ensamblador requiere que el programador escriba una línea para cada instrucción que desee que la máquina ejecute; por tanto, este lenguaje fuerza al programador a pensar como la máquina.

El reconocimiento de que se podía escribir un programa para traducir un lenguaje más potente a instrucciones del computador fue uno de los grandes avances en los primeros días de la computación. Los programadores de hoy en día deben su productividad (y su cordura) a la creación de los **lenguajes de programación de alto nivel** y los compiladores que traducen los programas en tales lenguajes en instrucciones. La figura 1.3 muestra la relación entre estos programas y lenguajes.

Lenguaje de programación de alto nivel: lenguaje transportable tal como C, Fortran o Java compuesto por palabras y notación algebraica que un compilador puede traducir en lenguaje ensamblador.

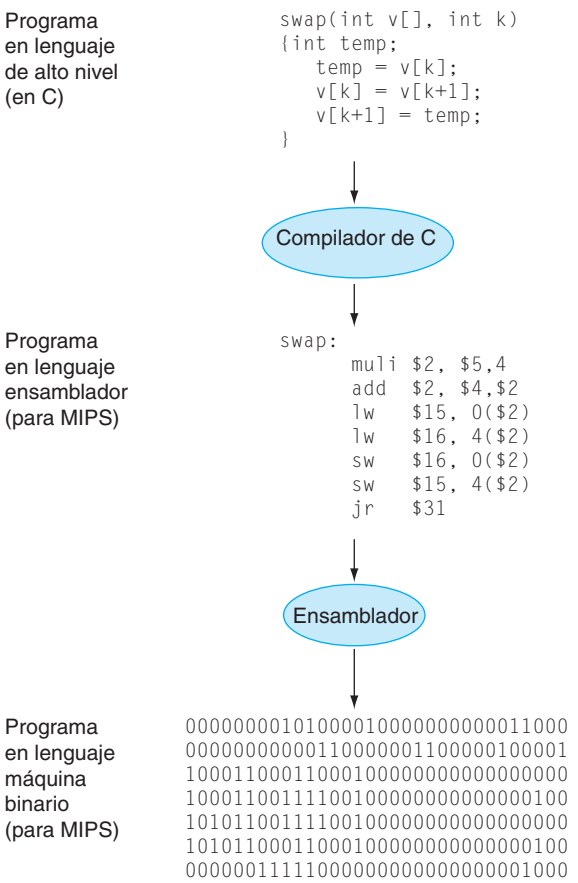


FIGURA 1.3 Programa en C compilado a lenguaje ensamblador y luego ensamblado a lenguaje máquina binario. Aunque la traducción del lenguaje de alto nivel al lenguaje máquina binario se muestra en dos pasos, algunos compiladores eliminan el paso intermedio y producen lenguaje máquina binario directamente. En el capítulo 2 se examinan con más detalle estos lenguajes y este programa.

Un compilador permiten que un programador escriba esta expresión en lenguaje de alto nivel:

```
A + B
```

El compilador compilaría esto en esta sentencia de lenguaje ensamblador:

```
add A, B
```

El ensamblador traduciría esto en la instrucción binaria que indica al computador que sume los números A y B.

Los lenguajes de programación de alto nivel ofrecen varias ventajas importantes. Primero, permiten al programador pensar en un lenguaje más natural, usando palabras inglesas y notación algebraica, dando lugar a programas con un aspecto mucho más parecido a un texto que a una tabla de símbolos crípticos (véase figura 1.3). Por otro lado, permiten que los lenguajes sean diseñados de acuerdo con su intención de uso. De este modo, el Fortran fue diseñado para computación científica, Cobol para el procesado de datos comerciales, Lisp para manipulación simbólica, etc. Hay también lenguajes de dominio específico, incluso para pequeños grupos de usuarios, como por ejemplo los interesados en simulación de fluidos.

La segunda ventaja de los lenguajes de programación es la mejora de la productividad del programador. Una de las pocas cuestiones con amplio consenso en el desarrollo de programas es que toma menos tiempo desarrollar programas cuando se escriben en lenguajes que requieren menos líneas para expresar una idea. La concisión es una ventaja clara de los lenguajes de alto nivel sobre el lenguaje ensamblador.

La ventaja final es que los lenguajes de programación permiten a los programas ser independientes del computador sobre el que se desarrollan, ya que los compiladores y ensambladores pueden traducir programas en lenguaje de alto nivel a las instrucciones binarias de cualquier máquina. Estas tres ventajas son tan decisivas que hoy en día se programa muy poco en lenguaje ensamblador.

1.3 Bajo la cubierta

Ahora que hemos mirado debajo de los programas para descubrir la programación subyacente, abramos la cubierta del computador para aprender sobre la circuitería que hay debajo. El hardware de cualquier computador lleva a cabo las mismas funciones básicas: introducción de datos, extracción de resultados, procesamiento de datos y almacenamiento de datos. El tema principal de este libro es explicar cómo se realizan estas funciones, y los capítulos siguientes tratan las diferentes partes de estas cuatro tareas.

Cuando llegamos a un punto importante en este libro, un punto tan importante que desearíamos que lo recordase para siempre, lo enfatizamos identificándolo con

un ítem de "Idea clave". Tenemos cerca de una docena de Ideas clave en este libro, siendo la primera los cinco componentes de un computador que realizan las tareas de entrada, salida, proceso y almacenamiento de datos.

IDEA clave

Los cinco componentes clásicos de un computador son entrada, salida, memoria, camino de datos y control, donde las dos últimas a veces están combinadas y se llaman el procesador. La figura 1.4 muestra la organización estándar de un computador. Esta organización es independiente de la tecnología del hardware: se puede colocar cada parte de cada computador, pasado y presente, en una de estas cinco categorías. Para ayudarle a tener todo esto en perspectiva, los cinco componentes de un computador se muestran en la primera página de los capítulos siguientes, con la parte de interés para ese capítulo resaltada.

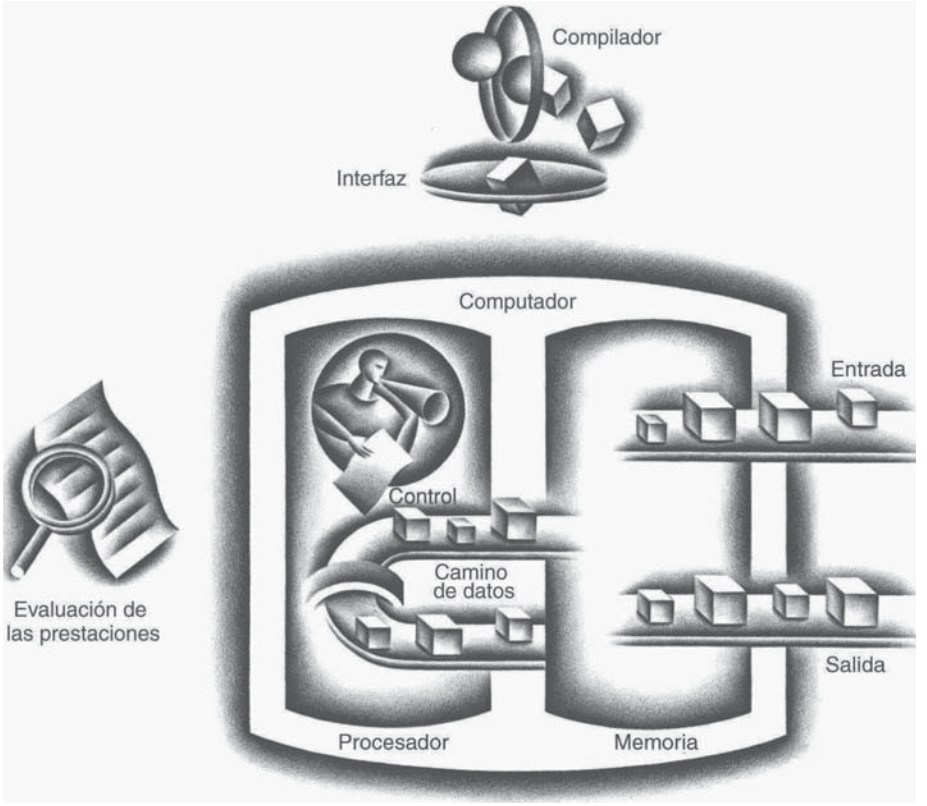


FIGURA 1.4 La organización de un computador, mostrando los cinco componentes clásicos. El procesador toma las instrucciones y datos de la memoria. La entrada escribe datos en la memoria y la salida lee datos de la memoria. El control envía señales que determinan las operaciones del camino de datos, la memoria, la entrada y la salida.

FIGURA 1.5 Un computador de sobremesa. La pantalla de cristal líquido (LCD) es el dispositivo de salida principal, y el teclado y el ratón son los dispositivos de entrada principales. En la parte derecha se puede ver un cable de Ethernet que conecta el portátil con la red y la web. El portátil tiene procesador, memoria y dispositivos de E/S. Este computador es un portátil Macbook Pro 15" conectado a un monitor externo.

La figura 1.5 muestra un computador de sobremesa típico, con teclado, ratón y pantalla. Esta fotografía revela dos de los componentes clave de un computador: **dispositivos de entrada**, como el teclado y el ratón, y **dispositivos de salida**, como la pantalla. Tal como sugiere su nombre, la entrada alimenta al computador y la salida es el resultado de la computación enviado al usuario. Algunos dispositivos, tales como las redes y discos, ofrecen al computador tanto entrada como salida.

El capítulo 6 describe los dispositivos de entrada/salida (E/S) con mayor detalle, pero ahora, como introducción, haremos un breve recorrido por la circuitería del computador, empezando por los dispositivos externos de E/S.

Dispositivo de entrada: mecanismo, como el teclado o el ratón, mediante el cual se introduce información al computador.

Dispositivos de salida: mecanismo que comunica el resultado de la computación a un usuario o a otro computador.

Se me ocurrió la idea del ratón mientras estaba en una conferencia de computadores. El conferenciante era tan aburrido que empecé a soñar despierto y di con la idea.

Doug Engelbart

En las pantallas de los computadores he hecho aterrizar un avión en la cubierta de un portaaviones en movimiento, he visto el impacto de una partícula nuclear con un átomo, he volantemplado un computador revelando todas sus tareas más internas.

Ivan Sutherland, "padre" de los gráficos por computador, cita de "Computer Software for Graphics"; *Scientific American*, 1984.

Anatomía de un ratón

Aunque muchos usuarios no se imaginarían ahora un computador sin ratón, la idea de un dispositivo para señalar, como es el ratón, fue mostrada por primera vez por Engelbart usando un prototipo de investigación en 1967. El *Alto,* que fue la inspiración para todas las estaciones de trabajo (*workstations*) y para los sistemas operativos Macintosh y Windows, incluía un ratón como dispositivo para señalar en 1973. En la década de 1990, todos los computadores de sobremesa incluían este dispositivo y se hicieron populares nuevas interfaces basadas en presentaciones gráficas y ratones.

El primer ratón era electromecánico y usaba una bola grande que cuando rodaba a lo largo de una superficie hacía que unos contadores *x* e *y* se incrementasen. La cantidad de incremento de cada contador decía cuánto se había movido el ratón.

La mayoría de los ratones electromecánicos fueron reemplazados por el novedoso ratón óptico, que en realidad es un procesador óptico en miniatura que incluye un diodo luminoso (*Light Emitting Diode*, LED) para proporcionar iluminación, una minúscula cámara en blanco y negro y un procesador óptico sencillo. El LED ilumina la superficie que hay debajo del ratón; la cámara toma 1500 fotografías de muestra por segundo de la zona iluminada. Las sucesivas fotografías se envían al procesador óptico que compara las imágenes y determina si el ratón se ha movido y en qué medida. La sustitución del ratón electromecánico por el ratón electro-óptico es una ilustración de un fenómeno común: los costes decrecientes y la mayor fiabilidad de la electrónica hace que una solución electrónica reemplace a una solución electromecánica más antigua. En la página 22 veremos otro ejemplo: la memoria flash.

A través del cristal de observación

Pantalla de panel plano, pantalla de cristal líquido: tecnología de pantalla que usa una capa fina de polímeros de líquidos que se pueden usar para transmitir o bloquear la luz según si se aplica una carga eléctrica o no.

Pantalla de panel plano, Pantalla de matriz activa: pantalla de cristal líquido que usa un transistor para controlar la transmisión de luz a cada píxel individual.

Píxel: el elemento individual más pequeño de la imagen. Las pantallas están formadas de millones de píxels organizados como una matriz.

El dispositivo de E/S más fascinante es probablemente la pantalla gráfica. Los computadores portátiles y los de mano, las calculadoras, los teléfonos móviles o celulares y casi todos los computadores de sobremesa usan una **pantalla de cristal líquido (*liquid crystal display*, LCD)** o de **panel plano**, para conseguir una pantalla delgada y de bajo consumo. La diferencia principal es que el píxel del LCD no es la fuente de luz; en su lugar controla la transmisión de la luz. Un LCD típico consiste en moléculas en forma de bastoncillos suspendidos en un líquido. Estas moléculas forman una hélice retorcida que refractan la luz que entra en la pantalla, habitualmente de una fuente de luz situada detrás de ella o, menos frecuentemente, de una luz reflejada. Las barras se alinean cuando se aplica una corriente y dejan de refractar la luz; puesto que el material del cristal líquido está entre dos pantallas polarizadas a 90 grados, la luz no puede pasar si no es refractada. Hoy en día, muchas pantallas de LCD usan una **matriz activa** que tiene un minúsculo transistor que actúa como interruptor en cada píxel para controlar con precisión la corriente y así formar imágenes más nítidas. La intensidad de los tres colores —rojo, verde, azul— de la imagen final se obtiene a partir de una máscara roja-verde-azul asociada a cada punto de la pantalla; en una matriz activa LCD de color, hay tres transistores en cada punto.

La imagen se compone de una matriz de elementos, o **píxels**,[1] que se pueden representar en una matriz de bits, llamada *mapa de bits (bit map)*. Dependiendo de la

1. Originariamente, contracción de "picture element", elemento de imagen. Dada la extensión de su uso, se ha optado por mantener el término. (N. del T.)

medida de la pantalla y de la resolución, la matriz varía en tamaño desde 640 × 480, hasta 2560 × 1600 píxeles en 2008. Una pantalla en color puede usar 8 bits para cada uno de los tres colores primarios (rojo, azul y verde), 24 bits por píxel en total, y permite ver millones de colores diferentes en la pantalla.

El soporte del hardware del computador para gráficos consiste principalmente en un búfer de refresco, o búfer de pantalla, para almacenar el mapa de bits. La imagen que se va a representar se almacena en el búfer de pantalla y el patrón de bits de cada píxel se leen hacia la pantalla gráfica a la velocidad de refresco. La figura 1.6 muestra un búfer de pantalla con 4 bits por píxel.

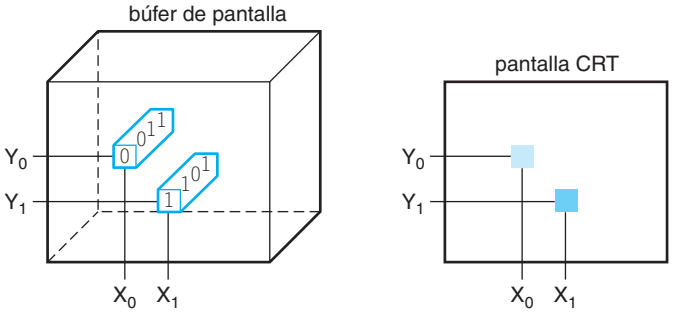


FIGURA 1.6 Cada coordenada en el búfer de pantalla de la izquierda determina el sombreado de la correspondiente coordenada de la pantalla CRT de la derecha. El píxel (X_0, Y_0) contiene el patrón de bits 0011, el cual es un tono de gris más claro en pantalla que el patrón de bits 1101 del píxel (X_1, Y_1).

El objetivo del mapa de bits es representar fielmente lo que está en la pantalla. Los problemas en sistemas gráficos aparecen debido a que el ojo humano es muy bueno detectando cambios sutiles.

Apertura de la caja

Si se abre la caja del computador, se ve una placa fascinante de plástico verde delgado, cubierta con docenas de pequeños rectángulos grises o negros. En la figura 1.7 se puede ver el contenido del computador portátil de la figura 1.5. En la parte superior de la foto se muestra la **placa base**. En la parte frontal hay dos bahías de disco, el disco duro a la derecha y el DVD a la izquierda. El hueco en el medio es para albergar la batería del portátil.

Los pequeños rectángulos de la placa base son los dispositivos que impulsan nuestra avanzada tecnología, los **circuitos integrados** o **chips**. La placa consta de tres partes: la parte que conecta con los dispositivos de E/S antes mencionados, la memoria y el procesador.

La **memoria** es el lugar donde se guardan los programas mientras se ejecutan; también contiene los datos requeridos por éstos. En la figura 1.8, la memoria se encuentra en las dos placas pequeñas, y cada pequeña placa de memoria contiene ocho circuitos integrados. La memoria de la figura 1.10 está construida con chips

Placa base: placa de plástico que contiene empaquetados de circuitos integrados o chips, incluyendo al procesador, cache, memoria y conectores para los dispositivos de E/S tales como redes o discos.

Circuito integrado: también llamado chip. Dispositivo que combina desde docenas a millones de transistores.

Memoria: área de almacenamiento en la cual se almacenan los programas cuando se están ejecutando y que contiene los datos que necesitan esos mismos programas.

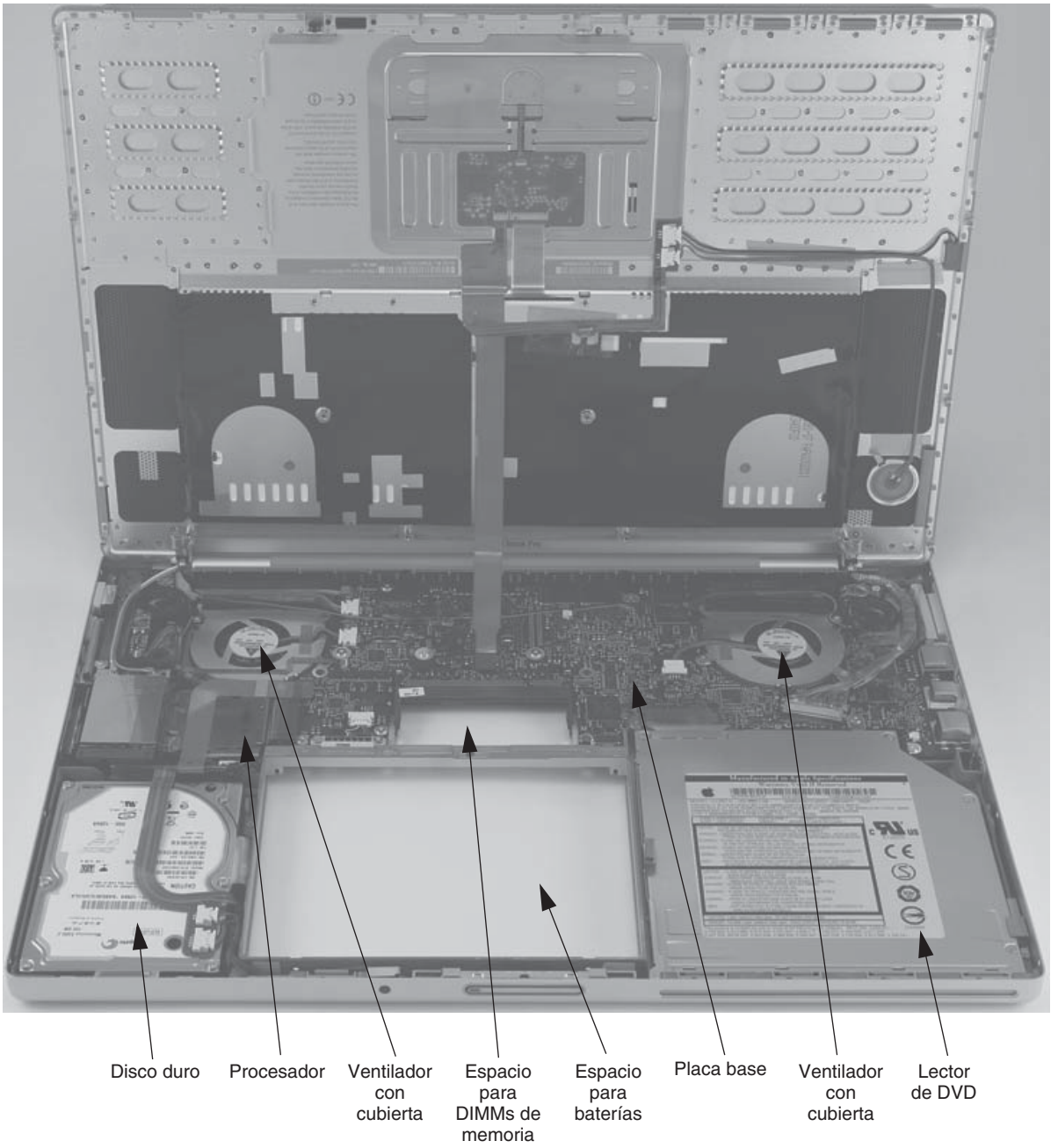


FIGURA 1.7 El portátil de la figura 1.5 por dentro. La caja delgada con la etiqueta blanca de la esquina izquierda es un disco duro SATA de 100GB, y la caja delgada metálica de la esquina derecha es un DVD. La batería se alojará en el hueco entre ellos. El hueco pequeño encima del hueco de la batería es para la memoria DIMM. La figura 1.8 muestra una visión más cercana de los DIMMs que se insertan en este portátil. En la parte superior de la batería y el DVD está la placa base, un circuito impreso que contiene la mayor parte de la electrónica del computador. Los dos círculos delgados de la mitad superior son dos ventiladores con sus cubiertas. El procesador es el rectángulo que puede verse justo debajo del ventilador de la izquierda. Foto por gentileza de OtherWorldComputing.com

DRAM. La DRAM es la **memoria dinámica de acceso aleatorio (*dynamic random access memory*)**. Se usan conjuntamente varias DRAMs para contener las instrucciones y los datos de los programas. En contraste con las memorias de acceso secuencial, como las cintas magnéticas, la parte RAM del término DRAM significa que los accesos a memoria toman el mismo tiempo independientemente de la posición de memoria que se lea.

Memoria de acceso aleatorio dinámica (*DRAM*): memoria construida como un circuito integrado, que provee acceso aleatorio a cualquier posición.

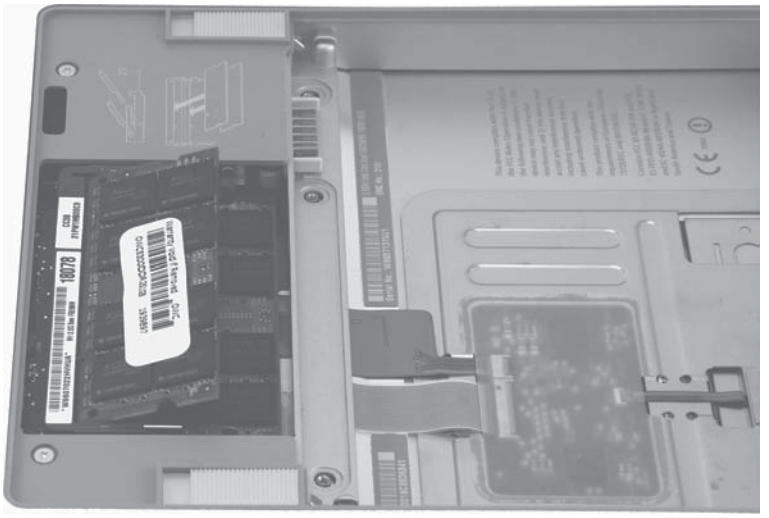


FIGURA 1.8 La memoria se aprecia con una vista cercana de la parte inferior del portátil. La memoria principal se aloja en una o más pequeñas tarjetas mostradas a la izquierda. El hueco para la batería se puede apreciar a la derecha. Los chips de DRAM se colocan en esas tarjetas (llamadas **DIMM**, *dual inline memory modules* o modulos de memoria en línea duales) y se enchufan a los conectores. Foto por gentileza de OtherWorldComputing.com

DIMM (módulo de memoria de dos líneas): pequeña tarjeta que contiene chips DRAM en ambas caras. Los SIMMs tienen DRAMs en una sola cara.

El *procesador* es la parte activa de la placa y se encarga de seguir las instrucciones de un programa al pie de la letra. Suma números, comprueba números, indica la activación de dispositivos de E/S, etc. El procesador es el cuadrado grande que se encuentra debajo del ventilador y está cubierto por un disipador térmico, en la parte izquierda de la figura 1.7. A veces, al procesador se le llama CPU, porque el término suena más "oficial": **unidad central de proceso (*central processing unit*, CPU)**.

Unidad central de proceso (*CPU*): también llamada procesador. Es la parte activa de un computador; contiene los caminos de datos y el control que suma y comprueba números, indica a los dispositivos de E/S que se activen, etc.

Adentrándonos aún más en el hardware, la figura 1.9 revela detalles del procesador. El procesador comprende dos componentes principales: el camino de datos *(datapath)* y el control, la fuerza y el cerebro del procesador, respectivamente. El **camino de datos** realiza las operaciones aritméticas. El **control** indica al camino de datos, a la memoria y a los dispositivos de E/S lo que deben hacer, de acuerdo con la voluntad de las instrucciones del programa. El capítulo 4 explica el camino de datos y el control para obtener un diseño con mejores prestaciones.

Camino de datos: componente del procesador que ejecuta las operaciones aritméticas.

Control: componente del procesador que gobierna el camino de datos, la memoria y los dispositivos de E/S según a las instrucciones del programa.

Introducirse en las profundidades de cualquier componente del hardware supone comprender lo que hay en la máquina. Dentro del procesador hay otro tipo de memoria: la memoria *cache*. La **memoria cache** es una memoria pequeña

Memoria cache: memoria rápida y pequeña que actúa como un *búfer* para otra memoria mayor y más lenta.

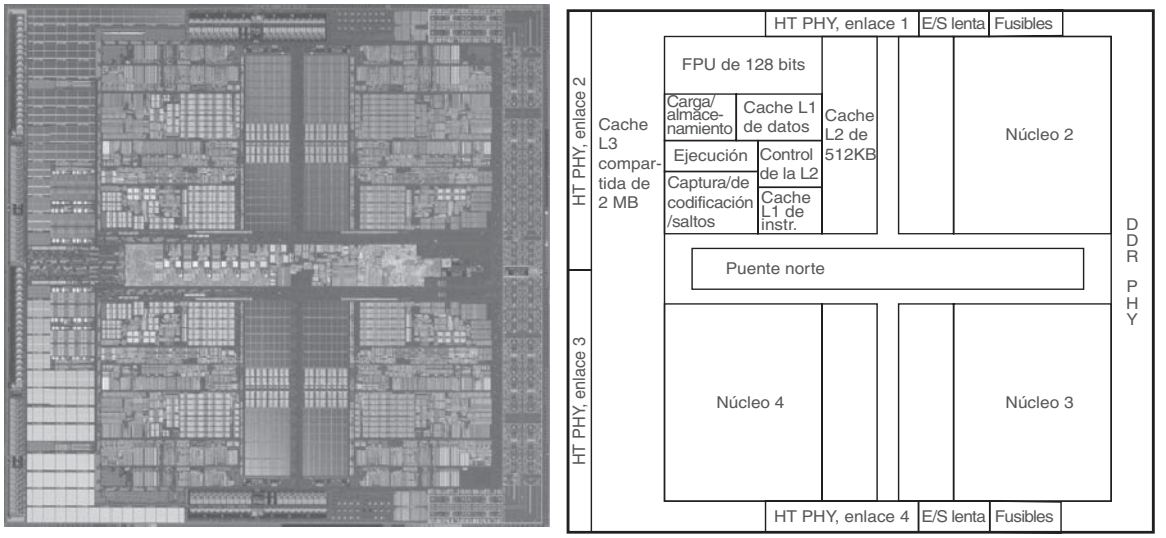


FIGURA 1.9 Interior del chip del microprocesador AMD Barcelona. La parte izquierda es una microfotografía de un chip procesador AMD Barcelona y la parte derecha muestra los principales bloques del procesador. Este chip tiene 4 procesadores o núcleos. El microprocesador del portátil de la figura 1.7 tiene dos núcleos, Intel Core 2 Duo.

Memoria de acceso aleatorio estática (SRAM): también es una memoria construida como un circuito integrado, pero es más rápida y menos densa que la DRAM.

Abstracción: modelo que oculta temporalmente los detalles de menor nivel de los sistemas de computadores para facilitar el diseño de sistemas sofisticados.

Arquitectura del repertorio de instrucciones (también llamada arquitectura): en una máquina, es una interfaz abstracta entre el hardware y el nivel más bajo del software que incluye toda la información necesaria para escribir un programa en lenguaje máquina que se ejecutará correctamente, e incluye

continúa...

y rápida que actúa como un búfer para la memoria DRAM. (Esta definición no técnica de *cache* es una buena forma de ocultar detalles que se verán posteriormente.) La memoria *cache* se construye usando una tecnología de memoria diferente, memoria de acceso aleatorio estático (***static random acces memory*, SRAM**). La SRAM es más rápida pero menos densa, y por lo tanto, más cara que la DRAM (véase capítulo 5).

Se observa fácilmente un aspecto común en las descripciones tanto del software como del hardware: penetrar en las profundidades revela más información, o a la inversa, los detalles de bajo nivel se ocultan para ofrecer un modelo más simple a los niveles más altos. El uso de estos niveles o **abstracciones** es una técnica fundamental para diseñar computadores muy complejos.

Una de las abstracciones más importantes es la interfaz entre el hardware y el nivel más bajo del software. Por su importancia recibe un nombre especial: **arquitectura del repertorio de instrucciones (*instruction set architecture*)**, o simplemente arquitectura, de una máquina. La arquitectura del repertorio de instrucciones incluye todo lo que los programadores puedan necesitar para construir un programa correcto en lenguaje máquina binario, incluidas las instrucciones, los dispositivos de E/S, etc. Típicamente, el sistema operativo se encarga de realizar los detalles de las operaciones de E/S, de asignación de memoria y de otras funciones de bajo nivel del sistema, de manera que los programadores de aplicaciones no necesitan preocuparse de

esos detalles. La combinación del repertorio de instrucciones básico y la interfaz del sistema operativo proporcionados para los programadores de aplicaciones se llama la **interfaz binaria de las aplicaciones (*application binary interface*, ABI)**.

Una *arquitectura del repertorio de instrucciones* permite a los diseñadores de computadores hablar de las funciones independientemente del hardware que las lleva a cabo. Por ejemplo, se puede hablar de las funciones de un reloj digital (mantener y mostrar la hora, establecer alarmas) independientemente de la circuitería del reloj (cristal de cuarzo, pantalla *LED,* botones de plástico). Los diseñadores de computadores hacen una distinción entre arquitectura e **implementación** de una arquitectura: una implementación es el hadware que obedece a una abstracción arquitectónica. Estas ideas nos llevan a otra Idea clave.

...continuación
las instrucciones, los registros, el acceso a memoria, la E/S, etc.

Interfaz binaria de aplicación (*ABI*): es la porción del repertorio de instrucciones correspondiente al usuario más la interfaz del sistema operativo que usan los programadores de aplicaciones. Define un estándar para la portabilidad binaria entre computadores.

Implementación: Hardware que cumple la abstracción de una arquitectura.

Tanto el hardware como el software están estructurados en niveles jerárquicos, cada uno de los cuales oculta detalles al nivel superior. Mediante este principio de *abstracción los* diseñadores de hardware y de software pueden enfrentarse a la complejidad de los computadores. Una interfaz clave entre los niveles de abstracción es la *arquitectura del repertorio de instrucciones:* la interfaz entre el hardware y el software de bajo nivel. Esta interfaz abstracta permite muchas *implementaciones* de costes y prestaciones diferentes para ejecutar programas idénticos.

IDEA clave

Un lugar seguro para los datos

Hasta ahora hemos visto cómo se pueden introducir datos, hacer cálculos con ellos y mostrarlos en pantalla. No obstante, si se fuera la corriente del computador, lo perderíamos todo porque su **memoria** interna es **volátil**; esto es, cuando se va la corriente, "olvida" su contenido. En cambio, un DVD no pierde la película grabada cuando se apaga el aparato, porque es una tecnología de **memoria no volátil**.

Para distinguir entre la memoria usada para almacenar programas mientras se ejecutan y esta memoria no volátil, usada para guardar programas entre ejecuciones, se utiliza el término **memoria principal** para la primera y **memoria secundaria** para la última. Las DRAMs predominan como memoria principal desde 1975,

Memoria volátil: almacenamiento, tal como la DRAM, que solo guarda los datos si está recibiendo alimentación eléctrica.

Memoria no volátil: forma de memoria que guarda los datos incluso en ausencia de alimentación eléctrica y que se usa para almacenar los programas entre ejecuciones. Los discos magnéticos son memoria no volátil, mientras que la memoria DRAM no lo es.

Memoria primaria (memoria principal): es la memoria volátil que se usa para almacenar los programas cuando se están ejecutando; típicamente se compone de DRAM en los computadores actuales.

Memoria secundaria: memoria no volátil usada para almacenar los

continúa...

...continuación
programas y datos entre ejecuciones; típicamente se compone de discos magnéticos en los computadores actuales.

Disco magnético (disco duro): forma de memoria secundaria no volátil compuesta de platos rotatorios recubiertos con un material de grabación magnético.

Memoria Flash: memoria semiconductora no volátil. Es más barata y más lenta que la DRAM, pero más cara y más rápida que los discos magnéticos.

pero los **discos magnéticos** predominan como memoria secundaria desde 1965. La **memoria Flash**, una memoria semiconductora no volátil, es utilizada en sustitución de discos móviles en dispositivos tales como teléfonos móviles y está reemplazando a gran velocidad a los discos en reproductores de música e incluso en portátiles.

Tal como muestra la figura 1.10, un disco duro magnético consiste en una colección de platos que giran a una velocidad de entre 5400 y 15 000 revoluciones por minuto. Los discos de metal están recubiertos de material magnético grabable en ambas caras, similar al material de las cintas de casete o de vídeo. Para leer y escribir información en un disco duro, sobre cada superficie hay un *brazo* móvil que en su extremo tiene una pequeña bobina electromagnética llamada *cabezal de lectura/escritura*. El disco entero está sellado permanentemente para controlar el ambiente de su interior y, además, para permitir que a los cabezales del disco estén mucho más cerca de la superficie del disco.

Hoy en día, los diámetros de los discos duros varían aproximadamente en un factor de 3, desde menos de 1 pulgada hasta 3.5 pulgadas, y su tamaño se ha ido reduciendo a lo largo de los años para caber en nuevos aparatos; las estaciones de trabajo servidoras, los computadores personales, los portátiles, los computadores de bolsillo y las cámaras digitales han inspirado nuevos formatos de discos. Tradicionalmente, los discos más grandes tienen mejores prestaciones, mientras que los más pequeños tienen un menor coste; sin embargo, el mejor

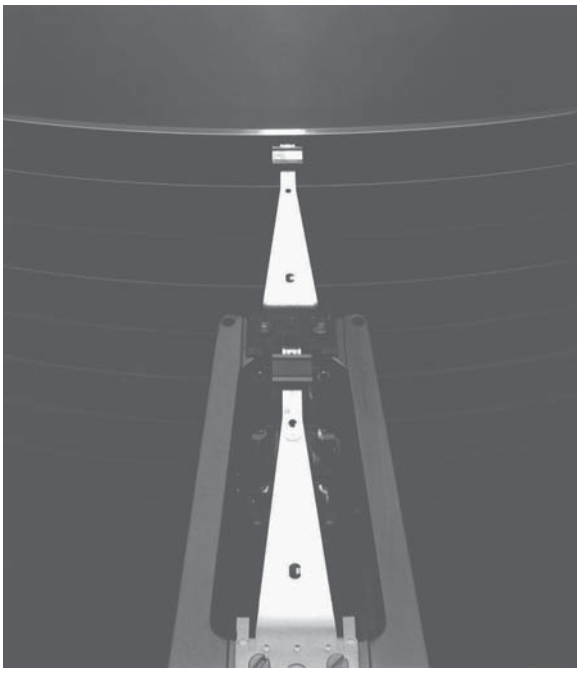


FIGURA 1.10 Disco donde se muestran 10 platos y los cabezales de escritura/lectura.

coste por **gigabyte** varía. Aunque la mayoría de los discos duros están dentro del computador (como en la figura 1.7), también pueden estar conectados mediante interfaces externas tales como USB.

Gigabyte: tradicionalmente 1 073 741 824 (2^{30}) bytes, aunque algunos sistemas de almacenamiento secundario y de comunicaciones lo han redefinido a 1 000 000 000 (10^9) bytes. De forma similar, un megabyte puede ser 2^{20} o 10^6 dependiendo del contexto.

El uso de componentes mecánicos hace que el tiempo de acceso de los discos magnéticos sea mucho más lento que el de las DRAMs. Típicamente, el tiempo de acceso de los discos es de unos 5 a 20 milisegundos, mientras que el de las DRAMs está entre los 50 y los 70 nanosegundos, por lo que éstas últimas son unas 100 000 veces más rápidas que los primeros. En contrapartida, el coste por megabyte de los discos es muy inferior al de las DRAMs, ya que los costes de producción de una determinada capacidad en disco son más bajos que para la misma capacidad en circuitos integrados. En 2008, el coste por megabyte de los discos era unas 30 a 100 veces más barato que el de las DRAMs.

Por lo tanto, hay tres diferencias principales entre discos magnéticos y memoria principal: los discos son no volátiles porque son magnéticos, tienen un tiempo de acceso más lento porque son mecánicos y son más baratos por megabyte porque tienen una gran capacidad de almacenaje con un coste moderado.

Ha habido muchos intentos de desarrollar una tecnología más barata que la DRAM y más cara que los discos para llenar el hueco entre ambos, pero la mayoría han fracasado. Los contendientes no fueron nunca capaces de hacer llegar el producto al mercado en el momento adecuado. Cuando un nuevo producto podía estar listo para comercializarse, DRAMs y discos habían continuado su evolución, los costes habían disminuido y el producto rival quedó obsoleto inmediatamente.

Sin embargo, la memoria Flash es un serio competidor. Esta memoria semiconductora es no volátil, al igual que los discos, tiene aproximadamente el mismo ancho de banda y la latencia es 100 a 1000 veces más rápida que la del disco. A pesar de que en 2008 el coste por gigabyte de una memoria Flash era de 6 a 10 veces superior al del disco, este tipo de memoria se ha vuelto muy popular en cámaras y reproductores de música portátiles porque viene con capacidades mucho menores, es más dura y es más eficiente en cuanto a consumo de potencia que los discos. Al contrario que en las DRAM y los discos, la memoria Flash se desgasta después de 100 000 a 1 000 000 escrituras. Así, el sistema de ficheros debe llevar cuenta del número de escrituras y debe tener una estrategia para evitar la pérdida de lo almacenado, como por ejemplo, copiar en otro soporte los datos más importantes. Las memorias Flash se describen con mayor detalle en el capítulo 6.

Aunque los discos duros no son extraíbles (de quita y pon), existen varias tecnologías de almacenamiento en uso que sí lo son, entre ellas las siguientes:

- Los discos ópticos, que incluyen tanto discos compactos (CDs) como discos de vídeo digital (DVDs), constituyen la forma más común de almacenamiento extraíble. El disco óptico Blu-Ray (BD) es el heredero natural del DVD.
- Las tarjetas de memoria extraíbles basadas en memoria FLASH se conectan mediante una conexión USB (*Universal Serial Bus*, Bus serie universal) y se usan para transferir archivos.
- Las cintas magnéticas sólo proporcionan un acceso en serie lento y se han usado para realizar copias de seguridad de discos, pero actualmente esta técnica es reemplazada habitualmente por la duplicación de discos.

La tecnología de los discos ópticos funciona de una manera totalmente diferente a la de los discos duros. En un CD, los datos se graban en forma de espiral, y los bits individuales se graban quemando pequeños hoyos —de aproximadamente 1 micra (10^{-6} metros) de diámetro— en la superficie del disco. El disco se lee radiando una luz láser en la superficie del CD y determinando si hay un hoyo o una superficie plana (reflectante) cuando se examina la luz reflejada. Los DVDs usan el mismo método para hacer rebotar un rayo láser sobre una serie de hoyos y superficies planas; pero además, el rayo láser se puede enfocar sobre múltiples capas y el tamaño de cada bit es mucho menor, lo cual incrementa significativamente la capacidad. En Blu-Ray se utiliza un láser con longitud de onda más corta que disminuye el tamaño de los bits y, por lo tanto, incrementa la capacidad.

Las grabadoras de CD y DVD de los computadores personales usan un láser para hacer los hoyos en la capa de grabación de la superficie del CD o del DVD. Este proceso de escritura es relativamente lento: desde unos minutos (para un CD completo) hasta cerca de 10 minutos (para un DVD completo). Por ello, para grabar cantidades grandes se usa una técnica de planchado diferente, que sólo cuesta unos pocos céntimos por cada CD o DVD.

Los CDs y DVDs regrabables tienen una superficie de grabación diferente, constituida por un material reflectante cristalino; los hoyos se forman de manera que no son reflectantes, al igual que en los CDs o DVDs de una sola escritura. Para borrar un CD o DVD regrabable, la superficie se calienta y enfría lentamente, permitiendo un proceso de templado que devuelve a la capa de grabación su estructura cristalina. Los discos regrabables son más caros que los de una sola escritura; para discos de solo lectura —usados para distribuir software, música o películas— tanto el coste del disco como el de la grabación son mucho menores.

Comunicación con otros computadores

Ya hemos explicado cómo se pueden introducir, calcular, mostrar y guardar datos, pero aún nos queda por ver un elemento presente en los computadores de hoy en día: las redes de computadores. Del mismo modo que el procesador de la figura 1.4 se conecta con la memoria y los dispositivos de E/S, las redes conectan computadores enteros y permiten que los usuarios aumenten su capacidad de computación mediante la comunicación. Las redes se han hecho tan populares que se han convertido en la columna vertebral de los sistemas informáticos actuales. Una máquina nueva sin una interfaz para red sería ridícula. Los computadores en red tienen algunas ventajas importantes:

- *Comunicación:* la información se intercambia entre computadores a gran velocidad.
- *Compartición de recursos:* en lugar de tener dispositivos de E/S para cada computador, los computadores de la red pueden compartir sus dispositivos.
- *Acceso no local:* conectando computadores separados a grandes distancias, los usuarios no necesitan estar cerca del computador que están usando.

Las redes varían en longitud y rendimiento, y el coste de comunicación aumenta en función de la velocidad de comunicación y de la distancia a la que debe viajar la información. Quizá la red más popular sea *Ethernet. Su* longitud está limitada a un kilómetro y transfiere hasta 10 gigabytes por segundo. Por su longitud y velocidad,

Ethernet es útil para conectar computadores de un mismo edificio, y de ahí que sea un buen ejemplo de lo que genéricamente se llaman **red de área local, o LAN (*local area network*)**. Las redes de área local se interconectan mediante conmutadores que pueden proporcionar también servicios de encaminamiento y de seguridad. Las **redes de área extensa, o WAN (*wide area network*)**, que cruzan continentes, son la espina dorsal de Internet, que soporta la *World Wide Web*; se basan típicamente en fibras ópticas y son gestionadas por empresas de telecomunicaciones.

Red de área local (LAN): red diseñada para transportar datos en un área confinada geográficamente, típicamente dentro de un mismo edificio.

Red de área extensa (WAN): red que se extiende a lo largo de cientos de kilómetros y que puede abarcar un continente.

Las redes han cambiado la fisonomía de la computación en los últimos 25 años, tanto por haberse extendido por todas partes como por su expectacular incremento en prestaciones. En la década de 1970, muy pocas personas tenían acceso al correo electrónico, e Internet y la Web no existían; así, el envío físico de cintas magnéticas era la principal manera de transferir grandes cantidades de datos entre dos sitios. En esa misma década, las redes de área local eran casi inexistentes, y las pocas redes de gran ámbito que existían tenían una capacidad limitada y eran de acceso restringido.

A medida que la tecnología de las redes mejoraba, se hizo mucho más barata y su capacidad aumentó. Por ejemplo, la primera tecnología de red de área local estándar desarrollada hace 25 años era una versión de Ethernet que tenía una capacidad máxima (también llamado ancho de banda) de 10 millones de bits por segundo, y típicamente era compartida por decenas, si no cientos, de computadores. Hoy, la tecnología de las redes de área local ofrecen una capacidad de entre 100 millones de bits por segundo a 10 gigabites por segundo, y es frecuentemente compartida por unos pocos computadores como mucho. La tecnología de las comunicaciones ópticas permitió un crecimiento similar en la capacidad de las redes de área extensa desde cientos de kilobits a gigabits, y desde cientos de computadores conectados a millones de computadores conectados entre sí formando una red mundial. El espectacular incremento en la utilización de la redes junto con el incremento en su capacidad han hecho que la tecnología de red sea central en la revolución de la información de los últimos 25 años.

Recientemente, otra innovación en las redes está dando nuevas formas a la manera en que los computadores se comunican. La tecnología inalámbrica se está utilizando ampliamente, y actualmente muchos computadores portátiles incorporan esta tecnología. La capacidad de hacer una transmisión por radio con la misma tecnología de semiconductores de bajo coste (CMOS) que se usa para la memoria y los microprocesadores ha posibilitado una mejora significativa en coste, lo que ha conducido a una explosión en su utilización. Las tecnologías inalámbricas, llamadas 802.11 según el estándar IEEE, permiten tasas de transmisión entre 1 y casi 100 millones de bits por segundo. La tecnología inalámbrica es bastante diferente de la redes basadas en cables, puesto que todos los usuarios de una misma área comparten las ondas radiofónicas.

Autoevaluación

- La memoria DRAM semiconductora y el almacenamiento en disco difieren significativamente. Describa la diferencia fundamental para cada uno de los siguientes aspectos: volatibilidad, tiempo de acceso y coste.

Tecnologías para construir procesadores y memorias

Los procesadores y la memoria han mejorado a un ritmo increíble porque los diseñadores de computadores han aprovechado la tecnología electrónica más avanzada para intentar ganar la carrera de diseñar un computador mejor. La figura 1.11

muestra las tecnologías que se han ido usando a lo largo del tiempo, con una estimación relativa de las prestaciones por unidad de coste de cada una de ellas. La sección 1.7 explora la tecnología que ha alimentado la industria de los computadores desde 1975 y que presumiblemente lo continuará haciendo en el futuro. Como esta tecnología determina lo que los computadores podrán hacer y la rapidez con que evolucionarán, creemos que todos los profesionales del sector deberían estar familiarizados con los fundamentos de los circuitos integrados.

Año	Tecnología usada en los computadores	Prestaciones relativas por unidad de coste
1951	Válvula de vacío	1
1965	Transistor	35
1975	Circuito integrado	900
1995	Circuito integrado a muy gran escala	2 400 000
2005	Circuito integrado a ultra gran escala	6 200 000 000

FIGURA 1.11 Prestaciones relativas por unidad de coste de las tecnologías usadas en los computadores a lo largo del tiempo. Fuente: Museo de los computadores, Boston, donde 2005 ha sido extrapolado por los autores.

Válvula de vacío: componente electrónico, predecesor del transistor, que consiste en tubo de vidrio hueco de entre 5 y 10 cm de longitud, en el cual se ha eliminado la mayor cantidad de aire posible, y que usa un haz de electrones para transferir datos.

Transistor: interruptor de encendido/apagado controlado por una señal eléctrica.

Circuito integrado a muy gran escala (VLSI): dispositivo que contiene cientos de miles a millones de transistores.

Un **transistor** es simplemente un interruptor de encendido/apagado, controlado eléctricamente. El *circuito integrado* combina desde docenas hasta cientos de ellos en un solo chip. Para describir el enorme incremento de integración desde cientos a millones de transistores, se utiliza el adjetivo *a muy gran escala,* que da lugar a la expresión **circuito a muy gran escala de integración,** o **circuito VLSI (*very large scale of integration*)**.

Esta velocidad de aumento de la integración ha sido notablemente estable. La figura 1.12 muestra el incremento de la capacidad de las DRAM desde 1977. La industria, prácticamente, ha cuadruplicado la capacidad cada 3 años, ¡un incremento total que excede las 16 000 veces en unos 20 años! Este incremento en el número de transistores en un circuito integrado se conoce popularmente como la ley de Moore, que establece que la capacidad en transistores se dobla cada 18 ó 24 meses. La ley de Moore es el resultado de una predicción del crecimiento en la capacidad de los circuitos integrados hecha por Gordon Moore, uno de los fundadores de Intel, durante los años sesenta del siglo pasado.

El mantenimiento de esta tasa de progreso durante al menos 40 años ha requerido increíbles innovaciones en las técnicas de manufacturación. En la sección 1.7 abordamos cómo se fabrican los circuitos integrados.

1.4 Prestaciones

Evaluar las prestaciones de un sistema puede ser un desafío. La envergadura y complejidad de los sistemas software modernos, junto con la amplia variedad de técnicas para mejorar las prestaciones empleadas por los diseñadores de hardware, han hecho que su evaluación sea mucho más difícil.

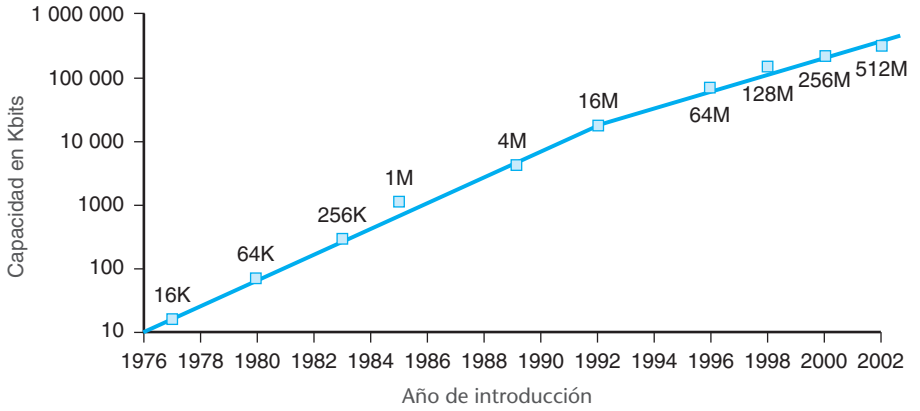


FIGURA 1.12 Crecimiento de la capacidad por chip DRAM a lo largo del tiempo. El eje *y* se mide en Kbits, donde K = 1024 (2^{10}). La industria DRAM ha cuadriplicado la capacidad casi cada 3 años, un incremento del 60% por año, durante 20 años. En años recientes, la tasa se ha frenado un poco y está más cerca de doblarse cada dos años o cuadruplicarse cada cuatro años.

Por supuesto, a la hora de elegir entre diferentes computadores, las prestaciones son casi siempre un atributo importante. Unas medidas precisas y una comparación entre diferentes máquinas son decisivas para los compradores y, por lo tanto, para los diseñadores. Los vendedores de computadores también conocen este hecho. A menudo, los vendedores desean que los compradores vean las mejores cualidades de su máquina, independientemente de si esas cualidades reflejan las necesidades de las aplicaciones de los compradores.

El resto de esta sección describe diferentes maneras de determinar las prestaciones. Después se describen las métricas para medir las prestaciones desde el punto de vista del usuario y del diseñador de computadores. También se analiza cómo estas métricas están relacionadas y se presenta la ecuación clásica sobre las prestaciones del procesador que será utilizada a lo largo del texto.

Definición de prestaciones

Cuando se dice que un computador tiene mejores prestaciones que otro, ¿qué se quiere decir? A pesar de que esta cuestión podría parecer simple, una analogía con los pasajeros de un avión muestra lo sutil que puede ser la cuestión de las prestaciones. La figura 1.13 muestra algunos aviones típicos de pasajeros, junto con su velocidad de crucero, autonomía y capacidad. Si se quisiera saber cuál de los aviones de esa tabla tiene las mejores prestaciones, primero se tendría que definir lo que se entiende por prestaciones. Por ejemplo, considerando diferentes medidas de las prestaciones, se observa que el avión con la mayor velocidad de crucero es el Concorde, el avión con la mayor autonomía es el DC-8, y el avión con la mayor capacidad es el 747.

Supongamos que se definen las prestaciones en términos de velocidad. Esto aún deja dos posibles definiciones. Se podría definir el avión más rápido como aquel que posea la velocidad de crucero más alta, es decir, el que lleve a un pasajero desde un punto a otro en el mínimo tiempo. Sin embargo, si se estuviera inte-

Avión	Capacidad	Autonomía	Velocidad de crucero (km/h)	Productividad pasajeros (pasajeros x km/h)
Boeing 777	375	4630	980	367 500
Boeing 747	470	4150	980	460 600
BAC/Sud Concorde	132	4000	2175	287 100
Douglas DC-8-50	146	8720	875	127 750

FIGURA 1.13 Capacidad, autonomía y velocidad de algunos aviones comerciales. La última columna muestra la razón a la que el avión transporta los pasajeros, que es la capacidad multiplicada por la velocidad de crucero (ignorando la autonomía y los tiempos de despegue y aterrizaje).

resado en transportar 450 pasajeros desde un punto a otro, el 747 sería claramente el más rápido, como muestra la última columna de la figura. Análogamente, las prestaciones de un computador se pueden definir de varias maneras diferentes.

Si se estuviera ejecutando un programa en dos estaciones de trabajo diferentes, se podría decir que la más rápida es la que acaba el trabajo primero. Si se estuviera dirigiendo un centro de computación que tiene dos grandes computadores de tiempo compartido, que ejecutan trabajos enviados por diferentes usuarios, se diría que el computador más rápido es aquel que ha completado más trabajos durante el día. Los usuarios individuales de computadores están más interesados en reducir el **tiempo de respuesta** —el tiempo entre el inicio y el final de una tarea— también referido como **tiempo de ejecución**. Los directivos de los centros de computación están habitualmente interesados en incrementar la productividad, que es la cantidad total de trabajo hecho en un cierto tiempo. Por lo tanto, en la mayoría de los casos, se necesitarán diferentes métricas de prestaciones, así como diferentes conjuntos de aplicaciones para probar las prestaciones de los computadores de sobremesa frente a los servidores y los computadores empotrados, que están más orientados al tiempo de respuesta, frente a los servidores, más orientados a la **productividad**.

Tiempo de respuesta (tiempo de ejecución): tiempo total requerido por un computador para completar una terea, incluidos los accesos a disco, los accesos a memoria, las operaciones E/S, sobrecarga del sistema operativo, tiempo de ejecución, etc.

Productividad, también llamada ancho de banda: Otra medida de las prestaciones, se define como el número de tareas que se completan por unidad de tiempo.

EJEMPLO

Productividad y tiempo de respuesta

Si en un sistema informático se realizan los siguientes cambios, ¿qué ocurre: aumenta la productividad, se reduce el tiempo de respuesta o ambos a la vez?

1. Se reemplaza el procesador de un computador por una versión más rápida.
2. Se añaden procesadores a un sistema que usa múltiples procesadores para diferentes tareas, por ejemplo, buscar en la World Wide Web.

RESPUESTA

Reducir el tiempo de respuesta de un sistema casi siempre mejora la productividad. Así, en el primer caso, ambos mejoran. En el segundo caso, ninguna de las tareas consigue que su trabajo se haga más rápido, por lo tanto sólo la productividad se verá incrementada. Sin embargo, si la demanda de procesadores en el segundo caso fuera casi tan alta como la productividad, el sistema podría verse forzado a poner en cola algunas peticiones. En este caso, el incremento de la productividad podría mejorar también el tiempo de respuesta, ya que eso reduciría el tiempo de espera en la cola. Por lo tanto, en muchos sistemas informáticos reales, cambiar el tiempo de ejecución o la productividad afecta a menudo al otro factor.

En nuestra exposición de las prestaciones de los computadores, en los primeros capítulos nos ocuparemos principalmente del tiempo de respuesta. Para maximizar las prestaciones, lo que se desea es minimizar el tiempo de respuesta o tiempo de ejecución de alguna tarea. Por lo tanto, las prestaciones y el tiempo de ejecución de un computador X se pueden relacionar de la siguiente manera:

$$\text{Prestaciones}_X = \frac{1}{\text{Tiempo de ejecución}_X}$$

Así, si las prestaciones de una máquina X son mayores que las prestaciones de una máquina Y, se tiene:

$$\text{Prestaciones}_X > \text{Prestaciones}_Y$$

$$\frac{1}{\text{Tiempo de ejecución}_X} > \frac{1}{\text{Tiempo de ejecución}_Y}$$

$$\text{Tiempo de ejecución}_Y > \text{Tiempo de ejecución}_X$$

Esto significa que el tiempo de ejecución de Y es mayor que el de X, si X es más rápido que Y.

Al tratar sobre el diseño de un computador, a menudo se desea relacionar cuantitativamante las prestaciones de dos máquinas diferentes. Usaremos la frase "X es *n* veces más rápida que Y" para indicar que:

$$\frac{\text{Prestaciones}_X}{\text{Prestaciones}_Y} = n$$

Si X es *n* veces más rápida que Y, entonces el tiempo de ejecución de Y es *n* veces mayor que el de X:

$$\frac{\text{Prestaciones}_X}{\text{Prestaciones}_Y} = \frac{\text{Tiempo de ejecución}_Y}{\text{Tiempo de ejecución}_X} = n$$

Prestaciones relativas

EJEMPLO

Si una máquina A ejecuta un programa en 10 segundos y una máquina B ejecuta el mismo programa en 15 segundos, ¿cuánto más rápida es A respecto de B?

RESPUESTA

Se sabe que A es *n* veces más rápida que B si

$$\frac{\text{Prestaciones}_A}{\text{Prestaciones}_B} = \frac{\text{Tiempo de ejecución}_B}{\text{Tiempo de ejecución}_A} = n$$

Así, la relación de las prestaciones es

$$\frac{15}{10} = 1.5$$

y, por lo tanto, A es 1.5 veces más rápida que B.

En el ejemplo anterior se podría decir también que la máquina B es 1.5 veces *más lenta* que la máquina A, ya que

$$\frac{\text{Prestaciones}_A}{\text{Prestaciones}_B} = 1.5$$

lo que significa que

$$\frac{\text{Prestaciones}_A}{1.5} = \text{Prestaciones}_B$$

Por simplicidad, usaremos normalmente el término *más rápido que* cuando intentemos comparar máquinas cuantitativamente. Ya que las prestaciones y el tiempo de ejecución son recíprocos, aumentar las prestaciones requiere reducir el tiempo de ejecución. Para evitar la confusión potencial entre los términos *aumentar* y *reducir,* diremos normalmente "mejorar las prestaciones" o "mejorar el tiempo de ejecución" cuando queramos decir "aumentar las prestaciones" o "reducir el tiempo de ejecución".

Medición de las prestaciones

El tiempo es la medida de las prestaciones de un computador: el computador que ejecuta la misma cantidad de trabajo en el menor tiempo es el más rápido. El *tiempo de ejecución* de un programa se mide en segundos. Pero el tiempo puede ser definido de maneras diferentes, dependiendo de lo que se cuente. La definición más sencilla de tiempo se llama *tiempo de reloj (wall clock time), tiempo de respuesta (response time)* o *tiempo transcurrido (elapsed time).* Estos términos se refieren al tiempo total que tarda una tarea en completarse, e incluye los accesos a disco, los accesos a memoria, las actividades de entrada/salida (E/S) y la sobrecarga introducida por el sistema operativo.

Tiempo de ejecución de CPU (tiempo de CPU): tiempo real que la CPU emplea en computar una tarea específica.

Tiempo de CPU del usuario: tiempo de CPU empleado en el propio programa.

Tiempo de CPU del sistema: tiempo que la CPU emplea en realizar tareas del sistema operativo para el programa.

Sin embargo, a menudo los computadores son de tiempo compartido, y un procesador podría trabajar en diferentes programas simultáneamente. En estos casos, el sistema podría intentar optimizar la productividad más que tratar de minimizar el tiempo de ejecución de un programa concreto. Por lo tanto, a menudo se querrá distinguir entre el tiempo transcurrido y el tiempo que un procesador está trabajando para nosotros. El **tiempo de ejecución de CPU**, o simplemente **tiempo de CPU**, el cual reconoce esta distinción, es el tiempo que la CPU dedica a ejecutar una tarea concreta y no incluye el tiempo perdido en las actividades de E/S o en la ejecución de otros programas. (Sin embargo, hay que recordar que el tiempo de respuesta que percibirá un usuario será el tiempo transcurrido para el programa, no el tiempo de CPU). Además, el tiempo de CPU puede ser dividido en el tiempo de CPU consumido por el programa, llamado **tiempo de CPU del usuario**, y el tiempo de CPU consumido por el sistema operativo, llamado **tiempo de CPU del sistema**. La diferenciación entre los tiempos de CPU del usuario y del sistema es difícil de realizar de una manera precisa,

ya que a menudo es complicado asignar responsabilidades a las actividades del sistema operativo para un programa de usuario más que a otro, y además los sistemas operativos presentan diferentes funcionalidades.

Por coherencia, mantendremos la distinción entre las prestaciones basadas en el tiempo transcurrido y las basadas en el tiempo de ejecución de CPU. Usaremos el término *prestaciones del sistema* para referirnos al tiempo transcurrido en un sistema sin carga, y usaremos las *prestaciones de la* CPU para referirnos al tiempo de CPU. Nos centraremos en las prestaciones de la CPU, aunque nuestras argumentaciones sobre la manera de exponer las prestaciones pueden ser aplicadas para el tiempo total de ejecución o para el tiempo de CPU.

Comprender las prestaciones de los programas

Diferentes aplicaciones son sensibles a diferentes aspectos de las prestaciones de un computador. Muchas aplicaciones, especialmente aquellas que se ejecutan en servidores, dependen en gran medida de las prestaciones de E/S, que a su vez depende tanto del hardware como del software, y el tiempo total transcurrido medido en tiempo de reloj (*wall clock time*) es la medida de interés. En algunos entornos de aplicación, el usuario puede preocuparse por la productividad, el tiempo de respuesta, o una compleja combinación de ambos (p. ej., la máxima productividad con un tiempo de respuesta en el peor caso). Para mejorar las prestaciones de un programa hay que tener una clara definición de cuáles son las métricas de las prestaciones importantes y entonces proceder a identificar problemas de prestaciones midiendo la ejecución del programa y buscando probables cuellos de botella. En los capítulos siguientes se describe cómo se buscan los cuellos de botella y se mejoran las prestaciones en varias partes del sistema.

Aunque como usuarios de computadores nos importa el tiempo, cuando se examinan los detalles de una máquina es conveniente considerar las prestaciones según otras medidas. En particular, es posible que los diseñadores de computadores quieran considerar una máquina utilizando una medida relacionada con la rapidez con que el hardware realiza determinadas funciones básicas. Casi todos los computadores tienen un reloj que funciona a una frecuencia concreta y determina el momento en que tienen lugar los sucesos en el hardware. Estos intervalos discretos de tiempo se llaman **ciclos de reloj** (o tics, tics de reloj, periodos de reloj o ciclos de reloj). Los diseñadores llaman a la longitud del **periodo de reloj** tanto el tiempo de un *ciclo de reloj* completo (p. ej. 0.25 nanosegundos, 0.25 ns, 250 picosegundos, o 250 ps) como la *frecuencia de reloj* (p. ej. 4 gigahercios o 4 GHz), que es el inverso del periodo de reloj. En la siguiente sección formalizaremos la relación entre los ciclos de reloj de los diseñadores de circuitos y los segundos de los usuarios de computadores.

Ciclo de reloj: también llamado tic, tic de reloj, periodo de reloj, ciclo de reloj. El tiempo de un periodo de reloj, usualmente del reloj del procesador, avanza a velocidad constante.

Periodo de reloj: longitud de cada ciclo de reloj.

Autoevaluación

1. Suponga que se sabe que una aplicación que utiliza un cliente local y un servidor remoto está limitada por las prestaciones de la red. Para los cambios siguientes, determine si sólo mejora la productividad, si mejoran tanto el tiempo de respuesta como la productividad, o si no mejora ninguna de las dos.
   a. Entre el cliente y el servidor se añade un canal de red extra que aumenta la productividad total de la red y reduce el retardo para obtener acceso a la red (ya que ahora hay dos canales).

b. Se mejora el software de red, de modo que se reduce el retardo de comunicación en la red, pero no se incrementa la productividad.
c. Se añade más memoria al computador.

2. Las prestaciones del computador C es 4 veces mejor que el del computador B, que ejecuta una aplicación dada en 28 segundos. ¿Cuánto tiempo emplea el computador C en ejecutar la aplicación?

Prestaciones de la CPU y sus factores

Frecuentemente, diseñadores y usuarios miden las prestaciones usando métricas diferentes. Si se pudieran relacionar estas métricas, se podría determinar el efecto de un cambio en el diseño sobre las prestaciones observadas por el usuario. Como nos estamos restringiendo a las prestaciones de la CPU, la medida base de prestaciones será el tiempo de ejecución de la CPU. Una fórmula sencilla que relaciona las métricas más básicas (ciclos de reloj y tiempo del ciclo de reloj) con el tiempo de CPU es la siguiente:

$$\text{Tiempo de ejecución de CPU para un programa} = \text{Ciclos de reloj de la CPU para el programa} \times \text{Tiempo del ciclo del reloj}$$

Alternativamente, ya que la frecuencia de reloj es la inversa del tiempo de ciclo,

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

Esta fórmula pone de manifiesto que el diseñador de hardware puede mejorar las prestaciones reduciendo la longitud del ciclo de reloj o el número de ciclos de reloj requeridos por un programa. Además, como se verá en otros capítulos, el diseñador a menudo se enfrenta a un compromiso entre el número de ciclos requerido por un programa y la longitud de cada ciclo. Muchas de las técnicas que reducen el número de ciclos incrementan también su longitud.

EJEMPLO

Mejora de las prestaciones

Nuestro programa favorito se ejecuta en 10 segundos en el computador A, que tiene un reloj de 2 GHz. Intentaremos ayudar al diseñador de computadores a que construya una máquina B que ejecute el programa en 6 segundos. El diseñador ha determinado que es posible un incremento sustancial en la frecuencia de reloj, pero que este incremento afectará al resto del diseño de la CPU, haciendo que la máquina B requiera 1.2 veces los ciclos de reloj que la máquina A necesitaba para ejecutar el programa. ¿Qué frecuencia de reloj debería ser el objetivo del diseñador?

RESPUESTA

Primero se calcularán los ciclos de reloj requeridos por el programa A:

$$\text{Tiempo de CPU}_A = \frac{\text{Ciclos de reloj de CPU}_A}{\text{Frecuencia de reloj}_A}$$

$$10 \text{ segundos} = \frac{\text{Ciclos del reloj de CPU}_A}{2 \times 10^9 \frac{\text{ciclos}}{\text{segundo}}}$$

$$\text{Ciclos de reloj de CPU}_A = 10 \text{ segundos} \times 2 \times 10^9 \frac{\text{ciclos}}{\text{segundo}} = 20 \times 10^9 \text{ciclos}$$

El tiempo de CPU para B se puede calcular utilizando la misma fórmula:

$$\text{Tiempo de CPU}_B = \frac{1.2 \times \text{Ciclos de reloj de CPU}_A}{\text{Frecuencia de reloj}_B}$$

$$6 \text{ segundos} = \frac{1.2 \times 20 \times 10^9 \text{ciclos}}{\text{Frecuencia de reloj}_B}$$

$$\text{Frecuencia de reloj}_B = \frac{1.2 \times 20 \times 10^9 \text{ciclos}}{6 \text{ segundos}} = \frac{4 \times 10^9 \text{ciclos}}{\text{segundo}} = 4 \text{ GHz}$$

Por lo tanto, la máquina B necesitará el doble de la frecuencia de reloj de la máquina A para ejecutar el programa en 6 segundos.

Prestaciones de las instrucciones

Las ecuaciones de los ejemplos previos no incluyen ninguna referencia al número de instrucciones ejecutadas por el programa. En el siguiente capítulo veremos qué aspecto tienen las instrucciones que forman un programa. Sin embargo, debido a que el compilador evidentemente genera las instrucciones que se deben ejecutar, y la máquina ha de ejecutarlas para que el programa funcione, el tiempo de ejecución debe depender del número de instrucciones del programa: una manera de pensar en el tiempo de ejecución es que éste es igual al número de instrucciones ejecutadas multiplicado por el tiempo medio por instrucción. Por lo tanto, el número de ciclos de reloj requerido por un programa puede ser representado como:

$$\text{Ciclos de reloj de CPU} = \text{Instrucciones de un programa} \times \frac{\text{Media de ciclos}}{\text{por instrucción}}$$

Ciclos de reloj por instrucción (CPI): número medio de ciclos de reloj por instrucción para un programa o fragmento de programa.

El término **ciclos de reloj por instrucción**, que es el número medio de ciclos de reloj *que* una instrucción necesita para ejecutarse, es a menudo abreviado como CPI *(clock cycles* per *instruction)*. Como instrucciones diferentes podrían

necesitar un número de ciclos diferente dependiendo de lo que hacen, el CPI es una media de todas las instrucciones ejecutadas por el programa. El CPI proporciona una manera de comparar dos realizaciones diferentes de la misma arquitectura del repertorio de instrucciones, ya que el número de instrucciones (o número total de instrucciones) requeridas por un programa será, obviamente, el mismo.

Utilización de la ecuación de las prestaciones

EJEMPLO

Supongamos que se tienen dos realizaciones de la misma arquitectura de repertorio de instrucciones. La máquina A tiene un tiempo de ciclo de reloj de 250 ps y un CPI de 2.0 para un programa concreto, mientras que la máquina B tiene un tiempo de ciclo de 500 ps y un CPI de 1.2 para el mismo programa. ¿Qué máquina es más rápida para este programa? ¿Cuánto más rápida es?

RESPUESTA

Sabemos que cada máquina ejecuta el mismo número de instrucciones para el programa, y a este número le llamaremos *I*. Primero, hallaremos el número de ciclos para cada máquina.

$$\text{Ciclos de reloj de CPU}_A = I \times 2.0$$

$$\text{Ciclos de reloj de CPU}_B = I \times 1.2$$

Ahora calcularemos el tiempo de CPU para cada máquina:

$$\begin{aligned}\text{Tiempo de CPU}_A &= \text{Ciclos de reloj de CPU}_A \times \text{Tiempo de ciclo}_A \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

De la misma forma, para B:

$$\text{Tiempo de CPU}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Claramente, la máquina A es más rápida. La diferencia en rapidez viene dada por la relación entre los tiempos de ejecución.

$$\frac{\text{Prestaciones CPU}_A}{\text{Prestaciones CPU}_B} = \frac{\text{Tiempo ejecución}_B}{\text{Tiempo ejecución}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

Se puede concluir que, para este programa, la máquina A es 1.2 veces más rápida que la máquina B.

La ecuación clásica de las prestaciones de la CPU

Ahora se puede escribir la ecuación básica de las prestaciones en términos del **número de instrucciones** (número de instrucciones ejecutadas por el programa), del CPI y del tiempo de ciclo:

Número de instrucciones: número de instrucciones ejecutadas por el programa.

$$\text{Tiempo de ejecución} = \text{Número de instrucciones} \times \text{CPI} \times \text{Tiempo de ciclo}$$

o bien, dado que la frecuencia es el inverso del tiempo de ciclo:

$$\text{Tiempo de ejecución} = \frac{\text{Número de instrucciones} \times \text{CPI}}{\text{Frecuencia de reloj}}$$

Estas fórmulas son especialmente útiles porque distinguen los tres factores claves que influyen en las prestaciones. Estas fórmulas se pueden utilizar para comparar dos realizaciones diferentes o para evaluar un diseño alternativo si se conoce el impacto en estos tres parámetros.

Comparación de segmentos de código

EJEMPLO

Un diseñador de compiladores está intentando decidir entre dos secuencias de código para una máquina en particular. Los diseñadores del hardware le han proporcionado los siguientes datos:

	CPI para instrucciones de esta clase		
	A	B	C
CPI	1	2	3

Para una declaración particular de un lenguaje de alto nivel, el diseñador del compilador está considerando dos secuencias de código que requieren el siguiente número de instrucciones:

Secuencia de código	Total de instrucciones por clase		
	A	B	C
1	2	1	2
2	4	1	1

¿Qué secuencia de código ejecuta el mayor número de instrucciones? ¿Cuál será la más rápida? ¿Cuál es el CPI para cada secuencia?

RESPUESTA

La secuencia 1 ejecuta 2 + 1 + 2 = 5 instrucciones. La secuencia 2 ejecuta 4 + 1 + 1 = 6 instrucciones. Por lo tanto la secuencia 1 ejecuta el menor número de instrucciones.

Se puede usar la ecuación para los ciclos de reloj de la CPU basada en el número de instrucciones y el CPI para encontrar el número total de ciclos para cada secuencia:

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

Y se obtiene

$$\text{Ciclos de reloj de la CPU}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ ciclos}$$

$$\text{Ciclos de reloj de la CPU}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ ciclos}$$

Por lo tanto, la segunda secuencia de código es más rápida, aun cuando ejecuta una instrucción más. Debido a que la secuencia de código 2 necesita menos ciclos totales, pero tiene más instrucciones, debe tener un CPI menor. Los valores del CPI pueden ser calculados así:

$$\text{CPI} = \frac{\text{Ciclos de reloj de la CPU}}{\text{Número de instrucciones}}$$

$$\text{CPI}_1 = \frac{\text{Ciclos de reloj de la CPU}_1}{\text{Número de instrucciones}_1} = \frac{10}{5} = 2$$

$$\text{CPI}_2 = \frac{\text{Ciclos de reloj de la CPU}_2}{\text{Número de instrucciones}_2} = \frac{9}{6} = 1.5$$

IDEA clave

La figura 1.14 muestra las medidas básicas a diferentes niveles del computador y lo que está siendo medido en cada caso. Se puede ver cómo se combinan estos factores para obtener el tiempo de ejecución medido en segundos

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

Se ha de tener siempre en cuenta que la única medida completa y fiable de las prestaciones de un computador es el tiempo. Por ejemplo, cambiar el repertorio de instrucciones para disminuir el número total de las mismas podría llevar a una organización con un ciclo de reloj mayor que contrarrestara la mejora en el número de instrucciones. Igualmente, ya que el CPI depende de la mezcla de instrucciones, el código que ejecuta el menor número de instrucciones podría no ser el más rápido.

Componentes de las prestaciones	Unidades de medida
Tiempo de ejecución de CPU de un programa	Segundos por programa
Número de instrucciones	Número de instrucciones ejecutadas por el programa
Ciclos por instrucción (CPI)	Número medio de ciclos por instrucción
Tiempo de ciclo del reloj	Segundos por ciclo de reloj

FIGURA 1.14 Componentes básicos de las prestaciones y cómo se mide cada uno de ellos.

¿Cómo podemos determinar el valor de estos factores en la ecuación de las prestaciones? El tiempo de CPU se puede medir ejecutando un programa, y el tiempo de ciclo normalmente forma parte de la documentación de una máquina. El número de instrucciones y el CPI pueden ser más difíciles de obtener. Evidentemente, si la frecuencia de reloj y el tiempo de ejecución de CPU son conocidos, sólo necesitaremos conocer o el número de instrucciones, o el CPI, para determinar el otro parámetro que falta.

El número de instrucciones se puede determinar utilizando herramientas que analizan la ejecución o usando un simulador de la arquitectura. Alternativamente, se pueden usar contadores hardware, los cuales forman parte de la mayoría de los procesadores con el objetivo de guardar una gran variedad de medidas, incluido el número de instrucciones ejecutadas por el programa, el CPI medio y, a menudo, las fuentes de pérdida de prestaciones. Ya que el número de instrucciones depende de la arquitectura, pero no de la implementación exacta, se puede medir el número de instrucciones sin conocer todos los detalles sobre la implementación. Sin embargo, el CPI depende de una amplia variedad de detalles del diseño de la máquina, que incluyen tanto el sistema de memoria como la estructura del procesador (como se verá en los capítulos 4 y 5), así como de la mezcla de tipos de instrucciones ejecutadas en una aplicación. Por lo tanto, el CPI varía según la aplicación, así como entre las diferentes realizaciones con el mismo repertorio de instrucciones.

El ejemplo anterior muestra el peligro de usar un solo factor (número de instrucciones) para evaluar las prestaciones. Cuando se comparan dos máquinas, se deben considerar los tres componentes, los cuales se combinan para formar el tiempo de ejecución. Si algunos de los factores son idénticos, como la frecuencia de reloj en el ejemplo anterior, las prestaciones pueden determinarse comparando los factores diferentes. Ya que la **mezcla de instrucciones** modifica el CPI, tanto éste como el número de instrucciones deben ser comparados, aunque las frecuencias de reloj sean idénticas. En varios ejercicios se le pedirá que estudie una serie de mejoras en el computador y el compilador que afectan a la frecuencia de reloj, el CPI y el número de instrucciones. En la sección 1.8 se examinará una medida de las prestaciones habitual que no incorpora todos los términos y que por lo tanto puede inducir a error.

Mezcla de instrucciones: medida de la frecuencia dinámica de las instrucciones a lo largo de uno o varios programas.

Comprender las prestaciones de los programas

Las prestaciones de un programa depende del algoritmo, del lenguaje, del compilador, de la arquitectura y del hardware real. La siguiente tabla resume de qué manera estos componentes afectan a los distintos factores de la ecuación de las prestaciones.

Componente hardware o software	¿A qué afecta?	¿Cómo?
Algoritmo	Número de instrucciones, posiblemente CPI	El algoritmo determina el número de instrucciones del programa fuente ejecutadas y por lo tanto el número de instrucciones del procesador ejecutadas. El algoritmo puede también afectar al CPI, favoreciendo instrucciones más lentas o más rápidas. Por ejemplo, si el algoritmo utiliza más operaciones en punto flotante, tenderá a tener un mayor CPI.
Lenguaje de programación	Número de instrucciones, CPI	El lenguaje de programación afecta al número de instrucciones, ya que las sentencias del lenguaje son traducidas a instrucciones del procesador, lo cual determina el número de instrucciones. Las características del lenguaje también pueden afectar al CPI; por ejemplo, un lenguaje con soporte para datos abstractos (p. ej. Java) requerirá llamadas indirectas, las cuales utilizarán instrucciones con un CPI mayor.
Compilador	Número de instrucciones, CPI	La eficiencia del compilador afecta tanto al número de instrucciones como al promedio de los ciclos por instrucción, ya que el compilador determina la traducción de las instrucciones del lenguaje fuente a instrucciones del computador. El papel del compilador puede ser muy complejo y afecta al CPI de formas complejas.
Arquitectura del repertorio de instrucciones	Número de instrucciones, frecuencia de reloj, CPI	La arquitectura del repertorio de instrucciones afecta a los tres aspectos de las prestaciones de la CPU, ya que afecta a las instrucciones necesarias para realizar una función, al coste en ciclos de cada instrucción, y a la frecuencia del reloj del procesador.

Extensión: Aunque se podría esperar que el valor mínimo para el CPI es 1, como veremos en el capítulo 4, algunos procesadores buscan y ejecutan varias instrucciones en cada ciclo de reloj; para reflejar este hecho, alguno diseñadores invierten el CPI para obtener el IPC, *instrucciones por ciclo*. Si un procesador ejecuta una media de 2 instrucciones por ciclo, el IPC es 2 y, por lo tanto, el CPI es 0.5.

Autoevaluación

Una aplicación dada escrita en Java se ejecuta durante 15 segundos en un procesador. Se dispone de un nuevo compilador Java que requiere 0.6 de las instrucciones que necesitaba el viejo compilador. Desafortunadamente, el CPI se incrementa en 1.1. ¿Cuánto tardará la aplicación en ejecutarse utilizando el nuevo compilador?

a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ s}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ s}$

c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ s}$

1.5 El muro de la potencia

La figura 1.15 ilustra el incremento en la frecuencia de reloj y el consumo de potencia de ocho generaciones de microprocesadores de Intel en los últimos 25 años. Tanto la frecuencia de reloj como el consumo de potencia crecieron de forma rápida durante décadas, y de forma más moderada recientemente. La razón de este crecimiento conjunto es que están correlacionados, y la razón de un crecimiento moderado más recientemente es que se han alcanzado los límites prácticos de disipación de potencia en los microprocesadores corrientes.

La tecnología dominante en la fabricación de circuitos integrados es la tecnología CMOS *(complementary metal oxide semiconductor)*. En CMOS, la fuente principal de disipación de potencia es la llamada potencia dinámica; es decir, potencia consumida en las transiciones. La disipación de potencia dinámica depende de la carga capacitiva de cada transistor, del voltaje aplicado y de la frecuencia de conmutación del transistor:

$$\text{Potencia} = \text{carga capacitiva} \times \text{voltaje}^2 \times \text{frecuencia de conmutación}$$

La frecuencia de conmutación depende de la frecuencia de la señal de reloj. La carga capacitiva por transistor es una función del número de transistores conectados a una salida (llamado *fanout*) y de la tecnología, que determina la capacitancia de las conexiones y los transistores.

¿Cómo es posible que la frecuencia de la señal de reloj crezca un factor 1000 mientras la potencia crece sólo un factor 30? La potencia puede reducirse disminuyendo el voltaje, algo que ha ocurrido con cada nueva generación de la tecnología, ya que depende de forma cuadrática del voltaje. Típicamente, el voltaje se ha reducido un

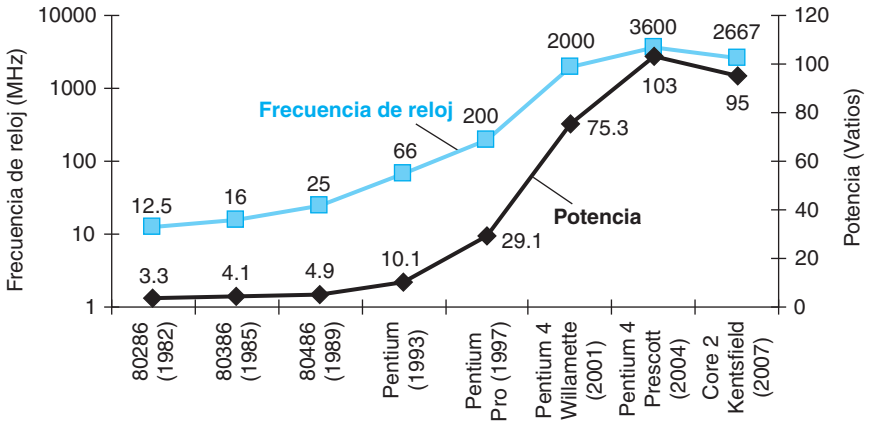


FIGURA 1.15 Variación de la frecuencia de reloj y la potencia en las últimas 8 generaciones en 25 años de microprocesadores x86. La arquitectura Pentium 4 dio un salto muy importante en la frecuencia de reloj y la potencia, pero menos importante en las prestaciones. La línea Pentium 4 se abandonó debido a los problemas térmicos del Prescott. La línea Core 2 vuelve a una segmentación más simple con frecuencias de reloj más bajas y varios procesadores por chip.

15% con cada nueva generación. En 20 años, el voltaje ha disminuido desde 5V a 1V; este el motivo que explica que la potencia sólo se haya incrementado 30 veces.

EJEMPLO

Potencia relativa

Supongamos que desarrollamos un nuevo procesador más sencillo, cuya carga capacitiva es el 85% de la carga de un procesador anterior más complejo. Supongamos, además, que el voltaje es ajustable de modo que puede reducirse en un 15% respecto al procesador anterior, lo que ocasiona una disminución del 15% en la frecuencia de la señal de reloj. ¿Cuál es el impacto en la potencia dinámica?

RESPUESTA

$$\frac{\text{Potencia}_{\text{nuevo}}}{\text{Potencia}_{\text{antiguo}}} = \frac{(\text{carga capacitiva} \times 0.85) \times (\text{voltaje} \times 0.85)^2 \times (\text{frecuencia} \times 0.85)}{\text{carga capacitiva} \times \text{voltaje}^2 \times \text{fremcuencia}}$$

Así, la relación de potencias es

$$0.85^4 = 0.52$$

Por lo tanto, el nuevo procesador utiliza aproximadamente la mitad de potencia que el procesador antiguo.

El problema actualmente es que disminuciones adicionales en el voltaje hacen que el transistor tenga demasiadas fugas, como grifos de agua que no cierran perfectamente. Hoy en día, aproximadamente el 40% del consumo de potencia es debido a la fuga de corriente. Si las pérdidas de los transistores continuasen creciendo, el proceso completo se volvería difícil de controlar.

En un intento de afrontar el problema de la potencia, los diseñadores ya han adosado grandes dispositivos para incrementar la refrigeración, y las partes del chip que no se utilizan en un ciclo de reloj determinado se apagan. Aunque hay otras técnicas para refrigerar los chips y, en consecuencia, aumentar la potencia, por ejemplo a 300 W, estas son demasiado caras para poder ser incluidas en los computadores de sobremesa.

Desde que los diseñadores de computadores se encontraron con el muro de la potencia, han necesitado una nueva forma de afrontarlo. Han elegido una forma de diseñar los microprocesadores diferente de la que se utilizó en los primeros 30 años.

Extensión: Aunque la potencia dinámica es la principal fuente de disipación de potencia en CMOS, hay una disipación estática debida a las fugas de corriente que se producen incluso cuando el transistor está inactivo. Como se ha mencionado anteriormente, en 2008, la fuga de corriente es la responsable del 40% de la disipación de potencia. De este modo, un aumento en el número de transistores aumenta la potencia disipada, incluso si los transistores estuviesen siempre inactivos. Se están empleando diversas técnicas e innovaciones tecnológicas para controlar la fuga de corriente, pero es difícil, conseguir disminuciones adicionales del voltaje.

1.6 El gran cambio: el paso de monoprocesadores a multiprocesadores

Hasta ahora, la mayor parte del software ha sido como la música escrita para un solista; con la actual generación de microprocesadores, estamos teniendo una pequeña experiencia con duetos y cuartetos y con otros grupos de pocos intérpretes; pero conseguir un trabajo para una gran orquesta y un coro es un reto diferente.

Brian Hayes, *Computing in a Parallel Universe*, 2007.

El muro de potencia ha forzado un cambio dramático en el diseño de microprocesadores. La figura 1.16 muestra la mejora en el tiempo de respuesta de programas para microprocesadores de computadores de sobremesa en los últimos años. Desde 2002, el factor de aumento ha disminuido desde un factor 1.5 por año hasta un factor menor que 1.2 por año.

En lugar de continuar con la reducción del tiempo de respuesta de un único programa ejecutándose en un único procesador, desde 2006 todas las compañías de computadores personales y servidores están incorporando microprocesadores con varios procesadores por chip, con los cuales el beneficio es, a menudo, más patente en la productividad que en el tiempo de respuesta. Para eliminar la confusión entre las palabras procesador y microprocesador, los fabricantes utilizan el término "núcleo *(core)*" para referirse a un procesador, y a estos microprocesadores se les denomina habitualmente microprocesadores multinúcleo *(multicore)*. Así, un microprocesador *quadcore*, es un chip que contiene cuatro procesadores o cuatro núcleos.

La figura 1.17 muestra el número de procesadores (núcleos), potencia y frecuencias de reloj de varios microprocesadores recientes. La previsión de muchos fabricantes es doblar el número de núcleos por microprocesador en cada generación de tecnología de semiconductores, es decir, cada dos años (véase capítulo 7).

En el pasado, los programadores contaban con las innovaciones en el hardware, la arquitectura y los compiladores para doblar las prestaciones de sus programas cada 18 meses sin tener que cambiar ninguna línea de código. En la actualidad, para obtener mejoras significativas en tiempos de respuesta, los programadores deben reescribir sus programas para extraer las ventajas de disponer de múltiples procesadores. Además, para alcanzar los beneficios históricos de ejecuciones más rápidas en nuevos microprocesadores, los programadores tendrán que continuar mejorando las prestaciones de sus códigos a medida que el número de núcleos de duplica.

Para enfatizar la manera en que hardware y software cooperan, a lo largo del libro usamos la sección especial *interfaz hardware/software*, la primera de las cuales se incluye a continuación. Estos elementos resumen visiones importantes de esta interfaz crítica.

Interfaz hardware software

El paralelismo siempre ha sido un elemento crítico para las prestaciones en computación, pero a menudo ha estado oculto. El capítulo 4 aborda la segmentación, una técnica elegante que permite ejecutar más rápidamente los programas mediante el solapamiento de la ejecución de las instrucciones. Este es un ejemplo del *paralelismo a nivel de instrucciones*, donde la naturaleza paralela del hardware está tan oculta que el programador y el compilador creen que el hardware ejecuta las instrucciones secuencialmente.

Conseguir que los programadores estén pendientes del hardware paralelo y reescriban sus programas explícitamente para ser paralelos había sido la "tercera vía" de la arquitectura de computadores, para aquellas compañías que el pasado dependieron de este cambio de conducta fallido (véase sección 7.14 en el CD). Desde esta perspectiva histórica, resulta asombroso que la industria de las Tecnologías de la

Información (TI) al completo haya apostado porque los programadores finalmente se habituarán a una programación explícitamente paralela.

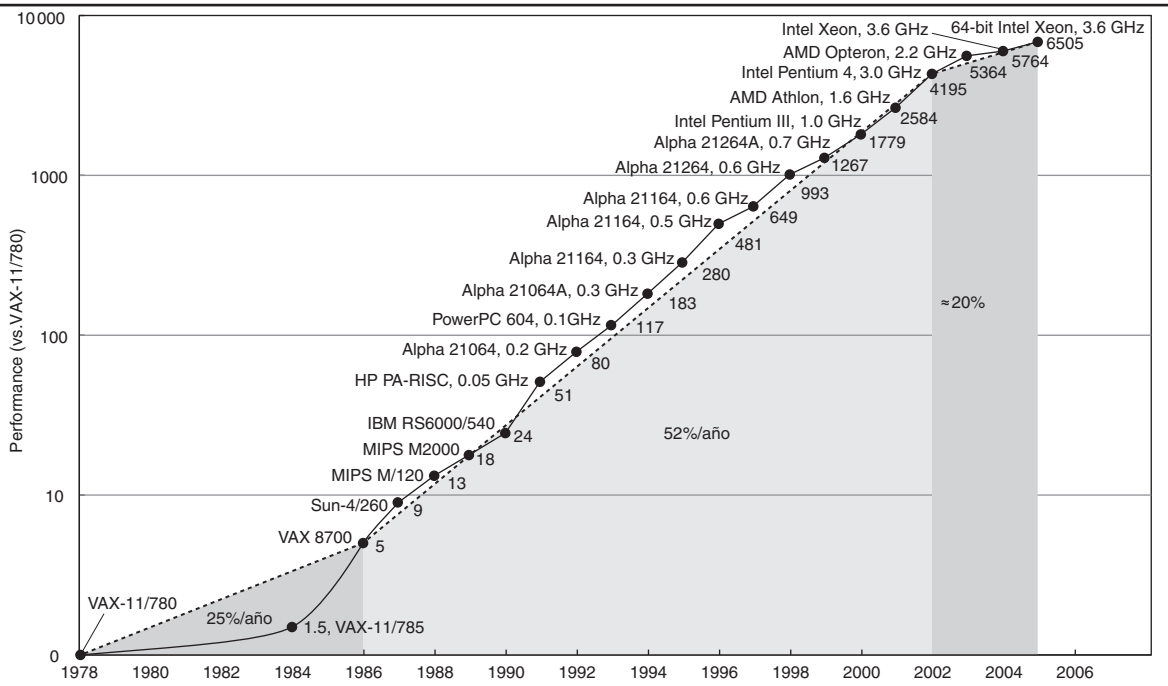


FIGURA 1.16 Aumento de las prestaciones de los procesadores desde la década de 1980. Esta gráfica muestra las prestaciones relativas al VAX 11/780 obtenidas con los programas de prueba SPECint (véase sección 1.8). Antes de mediados de la década de 1980 el aumento de las prestaciones se debía principalmente a la tecnología y de media suponía un 25% cada año. A partir de entonces hay un incremento del 52% durante varios años, atribuible a avances y nuevas ideas en la arquitectura y la organización. En 2002, este aumento había supuesto multiplicar por 7 las prestaciones. Las prestaciones de aplicaciones y cálculos orientados a punto flotante era todavía mayor. Desde 2002, el aumento de las prestaciones de un monoprocesador se redujo a un 20% anual debido a los límites en la disipación de potencia, el paralelismo a nivel de instrucciones disponible y la latencia de la memoria.

Producto	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Núcleos por chip	4	4	2	8
Frecuencia de reloj	2.5 GHz	~ 2.5 GHz ?	4.7 GHz	1.4 GHz
Potencia microprocesador	120 W	~ 100 W ?	~ 100 W ?	94 W

FIGURA 1.17 Número de núcleos por chip, frecuencia de reloj y potencia en varios microprocesadores multinúcleo de 2008.

¿Por qué ha sido tan difícil conseguir que los programadores escribiesen programas explícitamente paralelos? La primera razón es que la programación paralela es por definición programación de prestaciones, que presenta una mayor dificultad. No solamente obliga a que el programa sea correcto, resuelva un problema importante, y proporcione una interfaz útil a los usuarios o a otros progra-

madores que lo utilicen, el programa debe ser también rápido. Por el contrario, si no se necesitan prestaciones, es más sencillo escribir un programa secuencial.

La segunda razón es que rápido para un hardware paralelo significa que el programador debe dividir un aplicación de forma que todos los procesadores tengan que hacer, a grandes rasgos, la misma cantidad de trabajo en el mismo tiempo, y que la sobrecarga debido a la planificación y la coordinación no dilapide los beneficios potenciales del paralelismo.

Podemos utilizar la siguiente analogía. Supongamos la tarea de escribir un artículo para un periódico. Ocho periodistas trabajando en el mismo artículo podrían, potencialmente, escribir el artículo ocho veces más rápido. Para alcanzar este aumento de velocidad, sería necesario dividir la tarea para que cada periodista tuviese algo que hacer en todo momento. Por lo tanto, deberíamos planificar las subtareas. Si algo estuviese mal planificado y un periodista necesitase más tiempo que los otros siete, podrían perderse los beneficios de disponer de ocho escritores. Así, para obtener la aceleración deseada se debe *equilibrar la carga* equitativamente. Otro peligro sería que los periodistas tuviesen que perder demasiado tiempo hablando entre sí para escribir sus secciones. Por otra parte, se podría fracasar en alcanzar ese objetivo si una parte del artículo, por ejemplo las conclusiones, no pudiesen escribirse hasta que todas las demás partes estuviese terminadas. Así, se debe poner especial atención en reducir la *sobrecarga de las comunicaciones y la sincronización*. Tanto para esta analogía como para la programación paralela, los retos son la planificación, el equilibrio de la carga, el tiempo de sincronización y la sobrecarga de las comunicaciones entre colaboradores. Como se puede adivinar, el reto es todavía mayor con más periodistas para un artículo y más procesadores para programación paralela.

Para analizar la influencia de este gran cambio en la industria, cada uno de los siguientes cinco capítulos de esta edición del libro tienen una sección sobre las implicaciones de la revolución paralela sobre los conceptos de ese capítulo.

- *Capítulo 2, sección 2.11: Paralelismo e instrucciones: Sincronización*. Habitualmente, es necesario coordinar las tareas paralelas independientes en un cierto instante de tiempo, por ejemplo para indicar cuando han terminado su trabajo. En este capítulo se explican las instrucciones de sincronización de tareas utilizadas en los procesadores multinúcleo.
- *Capítulo 3, sección 3.6: Paralelismo y aritmética del computador: Asociatividad*. A menudo los programadores paralelos toman como punto de partida un programa secuencial ya operativo. La pregunta natural para determinar si versión paralela funciona correctamente es "¿se obtiene el mismo resultado?". Si la respuesta es no, la conclusión lógica es que la versión nueva tiene algún error. Esta lógica está asumiendo que la aritmética del computador es asociativa: el resultado de la suma de un millón de números es el mismo independientemente del orden en que se sumen. En este capítulo se explica que este argumento es válido para números enteros, pero no para número punto flotante.
- *Capítulo 4, sección 4.10: Paralelismo y paralelismo a nivel de instrucciones avanzado.* Dada la dificultad de la programación paralela explícita, desde comienzos de la década de 1990 se ha hecho un gran esfuerzo en el desarrollo de hardware y compiladores que aprovechen el paralelismo implícito. En este capítulo se describen

algunas de estas técnicas agresivas, incluida la búsqueda y ejecución simultánea de varias instrucciones, la predicción en los resultados de las decisiones y la ejecución especulativa de instrucciones.

- *Capítulo 5, sección 5.8: Paralelismo y jerarquía de memoria: Coherencia de la cache.* Una forma para reducir el coste de las comunicaciones es hacer que todos los procesadores usen el mismo espacio de direcciones, para que cualquier procesador pueda leer y escribir cualquier dato. Dado que todos los procesadores actuales usan caches para mantener una copia temporal de los datos en una memoria rápida cercana al procesador, es fácil darse cuenta de que la programación paralela sería incluso más dificultosa si las caches asociadas a cada procesador tuviesen valores incoherentes de los datos compartidos. En este capítulo se describen los mecanismos que permiten mantener valores coherentes de los datos en todas las caches.
- *Capítulo 6, sección 6.9: Paralelismo y E/S: Conjuntos redundantes de discos baratos.* Si nos olvidamos de la entrada y salida en esta revolución paralela, la consecuencia imprevista de la programación paralela podría ser que el programa paralelo desperdicia la mayor parte de su tiempo esperando la E/S. En este capítulo se describen los RAID, una técnica para mejorar las prestaciones en los accesos a los dispositivos de almacenamiento. RAID pone de manifiesto otro beneficio potencial del paralelismo: si se disponen de muchas copias de un recurso, el sistema puede continuar prestando el servicio a pesar de un fallo en alguno de los recursos. De esta forma, RAID puede mejorar tanto las prestaciones de E/S como la disponibilidad.

Además de estas secciones, hay un capítulo entero dedicado al procesamiento paralelo. El capítulo 7 profundiza en los retos de la programación paralela; presenta dos alternativas de comunicación a través de memoria compartida y envío explícito de mensajes; describe un modelo restringido de paralelismo fácil de programar; discute la dificultad de evaluación de los procesadores paralelos; introduce un nuevo y simple modelo de prestaciones para los microprocesadores multinúcleo y finalmente describe y evalúa cuatro ejemplos de microprocesadores multinúcleo usando este modelo.

El apéndice A es nuevo en esta edición del libro. En él se describe un componente hardware, incluido en los computadores personales, que cada vez es más popular: la unidad de procesamiento gráfico (GPU). Desarrolladas inicialmente para acelerar las aplicaciones con gráficos, las GPU se están convirtiendo en plataformas de programación. Como es de esperar en estos tiempos, las GPU son altamente paralelas. En el apéndice A se describe la GPU de NVIDIA y algunas partes de su entorno de programación paralela.

Pensé (que el computador) sería una idea aplicable universalmente, tal y como lo es un libro. Pero no pensé que se desarrollaría tan rápido como lo ha hecho, porque no imaginé que fuésemos capaces de poner tantos componentes en un chip tal y como finalmente se ha conseguido. El transistor llegó de manera inesperada. Todo sucedió mucho más rápido de lo que esperábamos.

J. Persper Eckert, coinventor del ENIAC, hablando en 1991.

1.7 Casos reales: fabricación y evaluación del AMD Opteron x4

Cada capítulo tiene una sección llamada "Casos reales" que enlaza los conceptos del libro con el computador que uno puede usar a diario. Estas secciones informan sobre la tecnología usada en los compuadores modernos. En este primer "Caso real", estudiamos cómo se fabrican los circuitos integrados y cómo se miden las prestaciones y la potencia, tomando el AMD Opteron X4 como ejemplo.

Empecemos por el principio. La fabricación de un chip empieza con el **silicio**, sustancia que se encuentra en la arena. Debido a que el silicio no es un buen conductor, se le llama **semiconductor**. Con un proceso químico especial, es posible añadir al silicio materiales que permiten transformar áreas muy pequeñas en uno de los tres dispositivos siguientes:

Silicio: elemento natural que es semiconductor.

Semiconductor: sustancia que no conduce bien la electricidad.

- Conductores excelentes de la electricidad (utilizando cables microscópicos de cobre o aluminio).
- Excelentes aislantes de la electricidad (como el plástico o vidrio).
- Áreas que pueden conducir o ser aislantes en determinadas condiciones (como un interruptor).

Los transistores están en la última categoría. Así, un circuito VLSI no es más que miles de millones de conductores, aislantes e interruptores fabricados en un único bloque muy pequeño.

El proceso de fabricación de circuitos integrados es crucial para el coste de los chips y, por lo tanto, importante para los diseñadores de computadores. La figura 1.18 muestra este proceso, que comienza con un **lingote de cristal de silicio**, cuyo aspecto es el de una salchicha gigante. Hoy en día, estas barras miden de unos 15 a 30 cm de diámetro y entre 30 y 60 cm de longitud. Las barras se rebanan muy finamente en **obleas** de menos de 0.25 cm de grosor. A estas obleas se les aplica un proceso de varias etapas en las cuales sobre cada una de ellas se estampan los patrones de productos quí-

Lingote de cristal de silicio: barra compuesta de cristal de silicio que tiene entre 15 y 30 cm de diámetro y entre 30 y 60 cm de longitud.

Oblea: rebanada de un lingote de silicio, de grosor no mayor que 0.25 cm, que se usa para fabricar chips.

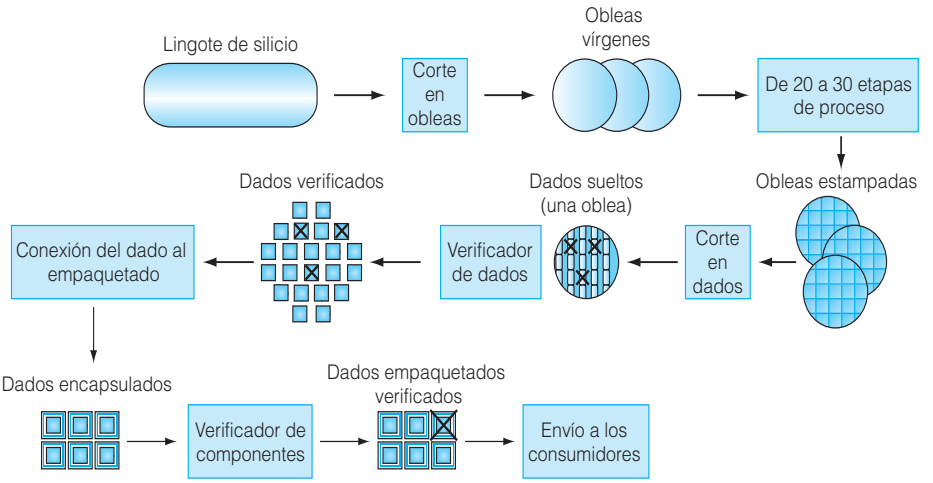


FIGURA 1.18 Proceso de fabricación de chips. Después de cortarse del lingote de silicio, las obleas vírgenes pasan por un proceso que tiene de 20 a 40 etapas para crear obleas estampadas (véase figura 1.9). Estas obleas estampadas se comprueban con un comprobador de obleas y se hace un mapa de las partes correctas. A continuación, las obleas se cortan en dados (véase figura 1.9). En esta figura, una oblea produce 20 dados, de los cuales 17 pasaron con éxito la comprobación. (X significa que el dado es defectuoso.) El factor de producción de dados correctos en este caso es de 17/20, o un 85%. Estos dados correctos se unen a los empaquetados y se comprueban una vez más antes de enviar los chips empaquetados a los clientes. En esta comprobación final se encontró un empaquetado defectuoso.

micos que crean los transistores, conductores y aislantes mencionados anteriormente. Los circuitos integrados actuales contienen una sola capa de transistores, pero podrían tener de dos a ocho niveles de conductor metal, separados por capas de aislante.

Defecto: imperfección microscópica en una oblea o en uno de los pasos de estampación que provoca que falle el dado que lo contiene.

Dado: sección rectangular individual que se corta de una oblea, más informalmente conocido como chip.

Factor de producción: porcentaje de dados correctos del total de dados de la oblea.

Una sola imperfección en la propia oblea o en cualquiera de las docenas de etapas de patronaje puede causar un mal funcionamiento en esa área de la oblea. Estos **defectos**, así se llaman, hacen prácticamente imposible fabricar una oblea perfecta. Para resolver este problema, se han usado varias estrategias, pero la más sencilla es poner muchos componentes independientes sobre una única oblea. La oblea estampada se corta en esos componentes, llamados **dados (*die/dices*)**, más informalmente conocidos como **chips**. La figura 1.19 es una fotografía de una oblea que contiene microprocesadores Pentium 4 antes de que se hayan cortado; antes, en la figura 1.9 de la página 21, mostrábamos un dado individual de Pentium y sus principales componentes.

El corte en dados permite descartar solamente aquellos que tengan defectos, en lugar de la oblea entera. Este concepto se cuantifica como **factor de producción (*yield factor*)** del proceso, que se define como el porcentaje de dados buenos del total de dados de una oblea.

El coste de un circuito integrado se eleva muy deprisa a medida que aumenta el tamaño del dado, tanto por el menor factor de producción como por el menor número de dados (de mayor tamaño) que caben en una oblea. Para reducir el coste, los dados grandes se "encogen" mediante un proceso de nueva generación que incorpora transistores y conductores de menor tamaño. Esto mejora el factor de producción y el número de dados por oblea.

Una vez que se dispone de los dados buenos, se conectan a las patas *(pins)* de entrada/salida del encapsulado *(package)* mediante un proceso de conexión *(bonding)*. Los componentes encapsulados *(packaged)* se verifican por última vez, ya que también se pueden producir errores durante este proceso y, finalmente, se envían a los consumidores.

Otra restricción de diseño de importancia creciente es la alimentación eléctrica. La alimentación es un reto. Esto es así por dos razones. En primer lugar, se debe llevar y distribuir electricidad a todo el chip; los microprocesadores modernos usan cientos de patas para las conexiones eléctricas y a tierra. Asimismo, se usan varios niveles de interconexión solamente para proporcionar electricidad y conexiones a tierra a determinadas a porciones del chip. En segundo lugar, la energía se disipa como calor que debe ser eliminado. ¡Un AMD Opteron X4 modelo 2352 a 2.0 GHz consume 120 vatios, que deben ser extraídos de un chip que tiene poco más de 1 cm^2 de superficie!

Extensión: El coste de un circuito integrado se puede expresar con tres ecuaciones simples:

$$\text{coste por dado} = \frac{\text{coste por oblea}}{\text{dado por oblea} \times \text{factor de producción}}$$

$$\text{dados por oblea} = \frac{\text{área de la oblea}}{\text{área del dado}}$$

$$\text{factor de producción} = \frac{1}{(1 + (\text{defectos por área} \times \text{área del dado}/2))^2}$$

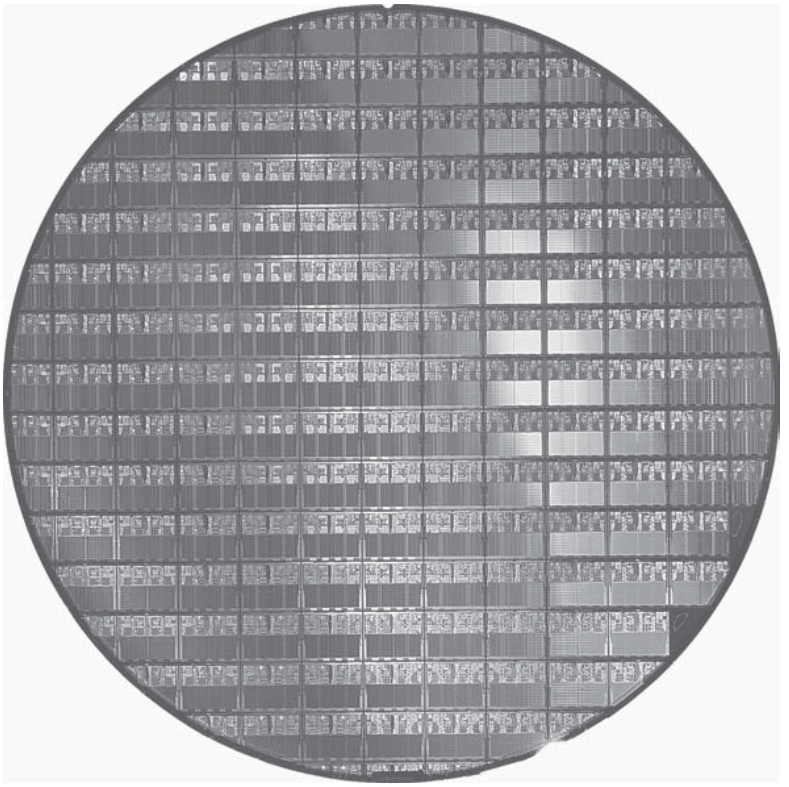


FIGURA 1.19 Un oblea de 12 pulgadas (300 mm) que contiene procesadores AMD Opteron X2, el predecesor del Opteron X4 (cortesía de AMD). Con un factor de producción del 100%, hay 117 dados en cada oblea. Las docenas de chips parcialmente redondeados que están en el borde la oblea no son útiles; se incluyen porque así es más fácil crear las máscaras para estampar el silicio. Estos dados se han fabricado con una tecnología de 90 nanómetros, lo que el significa que el tamaño del transistor más pequeño es aproximadamente de 90 nanómetros, aunque típicamente son algo más pequeños que el tamaño característico real, el cual se refiere al tamaño de los transistores cuando se dibujan y no al tamaño real fabricado.

La primera ecuación se obtiene de forma directa y sencilla. La segunda es una aproximación, puesto que no resta el área del borde de la oblea circular que no puede aprovecharse para dados (véase figura 1.19). La última ecuación se basa en observaciones empíricas del factor de producción en fábricas de circuitos integrados, y el exponente está relacionado con el número de pasos críticos del proceso de fabricación.

De este modo, generalmente los costes son no lineales con el área de dado, dependiendo del número de defectos y del tamaño del dado y de la oblea.

Evaluación de la CPU con programas de prueba SPEC

Carga de trabajo (*workload*): conjunto de programas que se están ejecutando en un computador que son o bien un conjunto real de aplicaciones de usuario, o bien un conjunto construido artificialmente a partir de fragmentos de programas reales. Una carga de trabajo típica especifica tanto los programas como su frecuencia relativa.

Programa de prueba (*benchmark*): programa utilizado para comparar las prestaciones de un computador.

Un usuario que está ejecutando los mismos programas día tras día sería un candidato perfecto para evaluar un nuevo computador; y el conjunto de programas utilizados formarían la **carga de trabajo**. Para llevar a cabo la evaluación de dos sistemas nuevos, el usuario simplemente tendría que comparar la carga de trabajo en los dos computadores. Sin embargo, la mayoría de los usuarios no se encuentran en esta situación. Por el contrario, tienen que depender de otros métodos de medida de las prestaciones de un computador candidato, esperando que estos métodos reflejen como se va a comportar el computador con la carga de trabajo del usuario. Con esta alternativa, se utiliza un conjunto de **programas de prueba**, especialmente seleccionados, para la evaluación de las prestaciones. Los programas de prueba forman una carga de trabajo que debe predecir las prestaciones con una carga de trabajo real.

SPEC (System Performance Evaluation Cooperative) es una organización fundada y financiada por un numeroso conjunto de vendedores de computadores para la creación de conjuntos estándar de programas de prueba para los computadores modernos. En 1989, originalmente SPEC creó un conjunto de programas de prueba enfocados hacia la evaluación de las prestaciones de procesadores (llamado ahora SPEC89), que evolucionó a lo largo de cinco generaciones. La última es SPEC CPU2006, y consiste en un conjunto de 12 programas de prueba de enteros (CINT2006) y 17 programas de prueba de punto flotante (CFP2006). Los programas de prueba de enteros varían desde parte de un compilador de C a un programa de ajedrez o a un código de simulación de un computador cuántico. Los programas de prueba de punto flotante incluyen códigos de mallas estructurados para el modelado de elementos finitos, códigos de métodos de partículas de dinámica molecular y álgebra lineal dispersa para dinámica de fluidos.

La figura 1.20 describe los programas de prueba de enteros y los tiempos de ejecución en el Opteron X4, y muestra los factores que explican el tiempo de ejecución: número de instrucciones, CPI y tiempo de ciclo del reloj. Observe que el CPI varía en un factor 13.

Para simplificar la comercialización de computadores, SPEC decidió resumir los 12 programas de prueba de enteros en un único número. En primer lugar, se normalizan los tiempos de ejecución medidos dividiendo el tiempo de ejecución en un procesador de referencia entre el tiempo de ejecución del computador que se está evaluando; obteniéndose así la *razón SPEC (SPECratio)*, que tiene la ventaja de que cuanto mayor es el número mayores son las prestaciones (es decir, la razón SPEC es inversamente proporcional al tiempo de ejecución). Haciendo la media geométrica de las razones SPEC se obtiene una medida resumen de los programas de prueba CINT2006 o CF2006.

Extensión: Cuando se comparan dos computadores utilizando razones SPEC, es conveniente utilizar la media geométrica porque nos da la misma respuesta relativa sin importar qué computador se utiliza como referencia para normalizar los tiempos. Por el contrario, si utilizásemos la media aritmética de los tiempos de ejecución, los resultados variarían dependiendo del computador utilizado como referencia.

La fórmula de la media geométrica es la siguiente

$$\sqrt[n]{\prod_{i=1}^{n} \text{Relaciones de tiempos de ejecución}_i}$$

Descripción	Nombre	Nº instr. × 10^9	CPI	Tiempo de ciclo del reloj (seg. × 10^9)	Tiempo de ejecución (segundos)	Tiempo de referencia (segundos)	Razón SPEC
Procesamiento de secuencias de caracteres	perl	2118	0.75	0.4	637	9770	15.3
Compresión *block-sorting*	bzip2	2389	0.85	0.4	817	9650	11.8
Compilador C de GNU	gcc	1050	1.72	0.4	724	8050	11.1
Optimización combinatorial	mc	336	10.00	0.4	1345	9120	6.8
Juego *Go* (IA)	go	1658	1.09	0.4	721	10 490	14.6
Búsqueda en secuencia de genes	hmmer	2783	0.80	0.4	890	9330	10.5
Juego de ajedrez (AI)	sjeng	2176	0.96	0.4	837	12 100	14.5
Simulación ordenador cuántico	libquantum	1623	1.61	0.4	1047	20 720	19.8
Compresión de vídeo	h264avc	3102	0.80	0.4	993	22 130	22.3
Librería de simulación de eventos discretos	omnetpp	587	2.94	0.4	690	6250	9.1
Juego *path finding*	astar	1082	1.79	0.4	773	7020	9.1
Análisis sintáctico de XML	xalancbmk	1058	2.70	0.4	1143	6900	6.0
Media geométrica							11.7

FIGURA 1.20 Programas de prueba SPECINT2006 en un AMD Opteron X4 modelo 2356 (Barcelona). Como se ha explicado en las ecuaciones de la página 35, el tiempo de ejecución es el producto de tres factores de la tabla; número de instrucciones, ciclos por instrucción (CPI) y tiempo de ciclo del reloj. Razón SPEC es el tiempo de referencia, proporcionado por SPEC, dividido entre el tiempo de ejecución medido. El número ofrecido como resultado del SPECINT2006 es la media geométrica de todas las razones SPEC. La figura 5.40 en la página 542 muestra que el CPI de mcf, libquantum, omnetpp y xalancbmk es relativamente alto debido a las elevadas relaciones de fallos de cache.

siendo *relaciones de tiempos de ejecución*$_i$ el tiempo de ejecución normalizado al del computador de referencia para el programa i de un total de n en la carga de trabajo y

$$\prod_{i=1}^{n} a_i \text{ significa el producto } a_1 \times a_2 \times \ldots \times a_n$$

Evaluación de la potencia con programas de prueba SPEC

En la actualidad, SPEC ofrece una docena de conjuntos de programas de prueba destinados al chequeo de una amplia variedad de ambientes de computación, usando aplicaciones reales y con una especificación estricta de las reglas de ejecución y los requerimientos de información. El más reciente es SPECpower, que proporciona datos sobre consumo de potencia de servidores con diferentes niveles de cargas de trabajo, divididos en tramos de incrementos del 10%, en un periodo de tiempo dado. La figura 1.21 muestra los resultados de un servidor con el procesador Barcelona.

El comienzo de SPECpower fueron los programas de prueba SPEC para aplicaciones de negocios codificadas en Java (SPECJBB2005), que evaluaban el procesador, la memoria principal y la cache, así como la máquina virtual de Java, el compilador, recolector de basura (*garbage collector*) y partes del sistema operativo. La medida de las prestaciones es la productividad y las unidades son operaciones de negocio por segundo. Una vez más, para facilitar la comercialización de los

Carga objetivo	Prestaciones (ssj_ops)	Potencia media (vatios)
100%	231 867	295
90%	211 282	286
80%	185 803	275
70%	163 427	266
60%	140 160	256
50%	118 324	246
40%	92 035	233
30%	70 500	222
20%	47 126	206
10%	23 066	180
0%	0	141
Suma global	1 283 590	2605
$\sum$ ssj_ops/$\sum$ potencia		493

FIGURA 1.21 SPECpower_ssj2008 en un AMD Opteron x4 2356 (Barcelona) con soporte para dos procesadores, 16 GB de memoria RAM dinámica DDR2-667 y un disco de 500 GB.

computadores, SPEC reduce estas medidas a un único número, llamado *ssj_ops global por vatio*. La fórmula para esta métrica de resumen es

$$\text{ssj\_ops global por vatio} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) / \left(\sum_{i=0}^{10} \text{potencia}_i\right)$$

siendo ssj_ops_i las prestaciones en cada incremento del 10% de la carga y potencia_i el consumo de potencia en cada nivel.

Autoevaluación

Un factor clave para determinar el coste de un circuito integrado es el volumen de fabricación. ¿Cuáles de las siguientes razones pueden hacer que el coste de un chip disminuya con el volumen de fabricación?

1. Con volúmenes elevados, el proceso de fabricación puede adaptarse a un diseño determinado, incrementando las prestaciones.
2. Es más fácil diseñar una parte que va a ser fabricada en grandes volúmenes que una parte que va a ser fabricada en volúmenes pequeños.
3. Las máscaras necesarias para la fabricación de los chips son caras, por lo tanto el coste por chip es menor para mayores volúmenes de fabricación.
4. Los costes de desarrollo de ingeniería son elevados y fuertemente dependientes del volumen; así, el coste de desarrollo por dado es menor para grandes volúmenes de fabricación.
5. Las partes que van a ser fabricada en grandes volúmenes usualmente tienen dados más pequeños que las partes que van a ser fabricada en volúmenes pequeños y, por lo tanto, tienen un mayor rendimiento por oblea.

1.8 Falacias y errores habituales

La ciencia debe empezar con mitos, y la crítica de los mitos.

Sir Karl Popper, *The Philosophy of Science*, 1957

El objetivo de una sección sobre falacias y errores más comunes, que podremos encontrar en cada capítulo, es explicar algunos errores de concepto comúnmente aceptados. A estos errores conceptuales los llamaremos *falacias*. Cada vez que se discuta una falacia, se intentará dar un contraejemplo. También discutiremos los errores más comunes. A menudo estos errores son generalizaciones de principios que son ciertos en un contexto limitado. El objetivo de estas secciones es ayudar a que estos errores no se reproduzcan en los computadores que el lector pueda diseñar o utilizar. Las falacias y errores más comunes relacionados con el coste/prestaciones han atrapado a muchos diseñadores de computadores, incluyéndonos a nosotros. Por lo tanto, en esta sección no faltarán ejemplos relevantes. Empezamos con un error que comenten muchos diseñadores y revela importantes relaciones en el diseño de computadores.

Error: Esperar que la mejora de un aspecto de un computador incremente las prestaciones globales en una cantidad proporcional a dicha mejora.

Este error ha sido cometido por diseñadores de hardware y diseñadores de software. Para ilustrarlo, proponemos un problema simple de diseño. Supongamos que un programa se ejecuta en 100 segundos en un computador dado, de los cuales 80 segundos se dedican a varias multiplicaciones. ¿Cuánto debo mejorar la velocidad de la multiplicación si quiero que mi programa se ejecute cinco veces más rápido?

El tiempo de ejecución del programa después de hacer las mejoras viene dado por la siguiente ecuación simple, conocida como **ley de Amdahl**:

Ley de Amdhal: regla que establece que el aumento posible de las prestaciones con una mejora determinada está limitado por la cantidad en que se usa la mejora. Esta es una versión cuantitativa de la ley de rendimiento decreciente en economía.

$$\text{tiempo de ejecución después de las mejoras} = \frac{\text{tiempo de ejecución por la mejora}}{\text{cantidad de mejora}} + \text{tiempo de ejecución no afectado}$$

Para este ejemplo:

$$\text{tiempo de ejecución por la mejora} = \frac{80 \text{ segundos}}{n} + (100 - 80 \text{ segundos})$$

como queremos que la ejecución sea cinco veces más rápida, el nuevo tiempo de ejecución debe ser 20 segundos, entonces

$$20 \text{ segundos} = \frac{80 \text{ segundos}}{n} + 20 \text{ segundos}$$

$$0 = \frac{80 \text{ segundos}}{n}$$

Es decir, no es posible mejorar la multiplicación para alcanzar un aumento en las prestaciones que implique que la ejecución es cinco veces más rápida, siempre y cuando la multiplicación suponga el 80% de la carga de trabajo.

El aumento posible de las prestaciones con una mejora determinada está limitado por la cantidad en que se usa la mejora. Este concepto también conduce a lo que llamamos ley de rendimiento decreciente en economía y en la vida diaria.

Podemos utilizar la ley de Amdahl para estimar el aumento de las prestaciones cuando se conoce el tiempo consumido por una función y su aceleración potencial. La ley de Amdahl, junto con las ecuaciones de prestaciones de la CPU, es una herramienta útil para la evaluación de mejoras potenciales. En los ejercicios se profundizará en la utilización de la ley de Amdahl.

Un corolario de la ley de Amdahl es de uso común en el diseño de hardware: *Hacer el caso más frecuente rápido.* Esta idea simple nos recuerda que en muchos casos un evento puede ocurrir mucho más frecuentemente que otro. La ley de Amdahl nos dice que la oportunidad para conseguir una mejora se ve afectada por el tiempo que este suceso consume. Por lo tanto, hacer rápido el caso más frecuente tenderá a que la mejora de las prestaciones sea mayor que optimizando el caso menos frecuente. Irónicamente, el caso más frecuente es a menudo más simple que el caso menos frecuente y así es a menudo más fácil de mejorar.

La ley de Amdahl se usa también para justificar los límites prácticos en el número de procesadores paralelos. Esto se examinará en la sección de falacias y errores más comunes del capítulo 7.

Falacia: Computadores con una utilización baja consumen poca potencia.

La eficiencia energética es importante con una baja utilización porque la carga de trabajo de los servidores puede variar. El uso de CPU para los servidores en Google, por ejemplo, está comprendida entre el 10% y el 50% la mayor parte del tiempo, y es del 100% menos del 1% del tiempo. La figura 1.22 muestra la potencia consumida por varios servidores con los mejores resultados del SPECpower con una carga del 100%, del 50%, 10% e inactivos. Incluso cuando los servidores están ocupados al 10%, la potencia consumida es dos tercios del consumo pico.

Dado que la carga de trabajo de los servidores varía pero consumen una parte elevada de la potencia pico, Luiz Barroso y Urs Hözle [2007] dicen que se debería rediseñar el hardware para alcanzar una "computación energeticamente proporcional". Si lo servidores utilizasen, por ejemplo, el 10% de la potencia pico cuando tienen una carga de trabajo del 10%, se reduciría el consumo de electricidad de un centro de datos y se comportarían como buenos ciudadanos en una época de grandes preocupaciones por la emisiones de CO_2.

Fabricante del servidor	Micro-procesador	Número de núcleos/ zócalos	Fre-cuencia de reloj	Presta-ciones pico (ssj_ops)	Poten-cia a 100% de carga	Poten-cia a 50% de carga	Carga 50%/ potencia 100%	Poten-cia a 10% de carga	Carga 10%/ potencia 100%	Potencia inactivo	Potencia inactivo/ potencia 100%
HP	Xeon E5440	8/2	3.0 GHz	308 022	269 W	227 W	84%	174 W	65%	160 W	59%
Dell	Xeon E5440	8/2	2.8 GHz	305 413	276 W	230 W	83%	173 W	63%	157 W	57%
Fujitsu Seimens	Xeon X3220	4/1	2.4 GHz	143 742	132 W	110 W	83%	85 W	65%	80 W	60%

FIGURA 1.22 Resultados de SPECPower para tres servidores con el mejor ssj_ops por vatio en el último cuarto de 2007. El ssj_ops por vatio de los tres servidores es 698, 682 y 667, respectivamente. La memoria de los dos primeros servidores es 16 GB y del último es 8 GB.

Error: Usar un subconjunto de las ecuaciones de prestaciones como una métrica de las prestaciones.

Se ha mostrado anteriormente la falacia resultante de predecir las prestaciones basándose únicamente en la frecuencia de la señal de reloj, o en el número de instrucciones, o en el CPI. Otro error muy corriente se produce cuando se utilizan sólo dos de los tres factores para comparar las prestaciones. Aunque esto puede ser válido en un contexto limitado, es fácil cometer errores. De hecho, casi todas las alternativas para usar el tiempo como una métrica de las prestaciones han dado lugar a conclusiones engañosas, resultados distorsionados o interpretaciones incorrectas.

Una alternativa al tiempo de ejecución son los **MIPS (millones de instrucciones por segundo)**. Para un programa dado,

$$\text{MIPS} = \frac{\text{número de instrucciones}}{\text{tiempo de ejecución} \times 10^6}$$

Millones de instrucciones por segundo (MIPS): medida de la velocidad de ejecución de un programa basada en el número de instrucciones. MIPS está definido como el número de instrucciones dividido por el producto del tiempo de ejecución por 10^6.

MIPS es una especificación de las prestaciones inversamente proporcional al tiempo de ejecución: los computadores más rápidos tienen valores mayores para MIPS. Por lo tanto, la ventaja de MIPS es que es una métrica fácil de entender e intuitiva, ya que computadores más rápidos tienen MIPS más elevados.

Sin embargo, cuando se utiliza como una medida para comparar computadores aparecen tres problemas. Primero, MIPS especifica la razón de ejecución de instrucciones pero no tiene en cuenta lo que hace cada instrucción. No es posible comparar computadores con diferentes conjuntos de instrucciones usando MIPS, porque el número de instrucciones será diferente. Segundo, MIPS varía entre diferentes programas ejecutándose en el mismo computador; así, un computador no puede tener un único valor de MIPS. Por ejemplo, sustituyendo el tiempo de ejecución, obtenemos las relación entre MIPS, frecuencia del reloj y CPI:

$$\text{MIPS} = \frac{\text{número de instrucciones}}{\dfrac{\text{número de instrucciones} \times \text{CPI}}{\text{frecuencia de reloj}} \times 10^6} = \frac{\text{frecuencia de reloj}}{\text{CPI} \times 10^6}$$

Tal como se mostró anteriormente, el CPI varía en un factor 13 con los programas del SPEC2006 en un Opteron X4, por lo tanto MIPS varía de la misma forma. Por último, y más importante, las variaciones de MIPS pueden ser independientes de las prestaciones.

Autoevaluación

Considere las siguientes medidas de las prestaciones de un programa:

Medida	Computador A	Computador B
Número de instrucciones	10 billones	8 billones
Frecuencia del reloj	4 Ghz	4 Ghz
CPI	1.0	1.1

a. ¿Qué computador tiene un MIPS más elevado?
b. ¿Qué computador es más rápido?

Mientras el ENIAC está equipado con 18 000 válvulas de vacío y pesa 30 toneladas, los computadores del futuro pueden tener 1000 válvulas y quizás pesen sólo 1 tonelada y media.

Popular Mechanics, Marzo de 1949

1.9 Conclusiones finales

Aunque es difícil predecir exactamente la relación coste/prestaciones que tendrán los computadores en el futuro, afirmar que será mucho mejor que la actual es una apuesta segura. Para participar en estos avances, los diseñadores de computadores y programadores deben entender una gran variedad de cuestiones.

Los diseñadores de hardware y los de software construyen sus sistemas en niveles jerárquicos, donde cada nivel oculta sus detalles a los niveles superiores. Este principio de abstracción es fundamental para comprender los computadores de hoy en día, pero no significa que los diseñadores se puedan limitar al conocimiento de una sola tecnología. El ejemplo de abstracción más importante es quizá la interfaz entre el hardware y el software de más bajo nivel, llamada *arquitectura del repertorio de instrucciones.* Mantener la arquitectura del repertorio de instrucciones constante permite muchas implementaciones de esa arquitectura (diferentes en coste y prestaciones) capaces de ejecutar los mismos programas. En el lado negativo, la arquitectura puede imposibilitar la introducción de innovaciones que requieran cambios en esa interfaz.

Hay un método fiable para determinar las prestaciones utilizando como métrica el tiempo de ejecución de programas reales. Este tiempo de ejecución está relacionado con otras métricas importantes mediante la siguiente ecuación:

$$\frac{\text{segundos}}{\text{programa}} = \frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos de reloj}}{\text{instrucción}} \times \frac{\text{segundos}}{\text{ciclos de reloj}}$$

Esta ecuación y los factores que en intervienen en ella serán utilizados en muchas ocasiones en el libro. Es importante tener presente que estos factores, de forma individual, no determinan las prestaciones, sólo el producto de ellos, que es igual al tiempo de ejecución, es una medida fiable de las prestaciones.

IDEA clave

La única métrica totalmente válida para evaluar las prestaciones es el tiempo de ejecución. Se han propuesto otras métricas pero no han dado la talla. A veces, estas métricas tenían ya fallos desde el mismo concepto, por no reflejar el tiempo de ejecución. En otras ocasiones, métricas que eran válidas en un contexto limitado se utilizaron en un contexto más amplio sin las aclaraciones necesarias para poder aplicarlas sin errores.

La tecnología clave de los procesadores modernos es el silicio. Para entender la tecnología de los circuitos integrados es igual de importante comprender las tasas de cambio tecnológico esperadas. Mientras el silicio alimenta el rápido avance del hardware, nuevas ideas sobre la organización de los computadores han mejorado la relación precio/prestaciones. Dos de las ideas clave son la explotación del paralelismo en

el procesador, típicamente vía la utilización de varios procesadores, y la explotación de la localidad de los accesos a la jerarquía de memoria, típicamente vía *caches*.

El área de silicio ha sido reemplazado por el consumo de potencia como recurso más crítico del diseño de microprocesadores. Aumentar las prestaciones manteniendo la potencia ha forzado que la industria del hardware cambie a los microprocesadores multinúcleo; en consecuencia, la industria del software se ha visto forzada ha moverse hacia la programación de hardware paralelo.

Los diseños de computadores siempre se han medido por su coste y sus prestaciones, y por otros factores también importantes, como la potencia, fiabilidad, propiedad y escalabilidad. Aunque en este capítulo nos hemos centrado en el coste, las prestaciones y la potencia, los mejores diseños mantendrán un equilibrio adecuado entre todos estos factores, para un segmento de mercado concreto.

Mapa de carreteras de este libro

En el nivel más bajo de estas abstracciones están los cinco elementos clásicos del computador: camino de datos, control, memoria, entrada y salida (véase la figura 1.5). Estos cinco componentes también sirven como marco para el resto de los capítulos de este libro:

- *Camino de datos:* capítulos 3, 4, 7 y apéndice A
- *Control:* capítulos 4, 7 y apéndice A
- *Memoria:* capítulo 5
- *Entrada:* capítulo 6
- *Salida:* capítulo 6

Como se ha dicho anteriormente, el capítulo 4 describe como el procesador explota el paralelismo implícito, el capítulo 7 describe el corazón de la revolución paralela, los microprocesadores multinúcleo con paralelismo explícito, y el apéndice A describe los altamente paralelos procesadores gráficos. El capítulo 5 describe como la jerarquía de memoria explota la localidad. El capítulo 2 describe los repertorios de instrucciones —la interfaz entre los compiladores y la máquina— y enfatiza el papel de los compiladores y lenguajes de programación en el uso de las características del repertorio de instrucciones. El apéndice B proporciona una referencia para el repertorio de instrucciones del capítulo 2. El capítulo 3 describe como los computadores manipulan datos aritméticos. El **apéndice C**, en el CD, introduce los conceptos básicos del diseño lógico.

Un campo activo de la ciencia es como un inmenso hormiguero. El individuo casi desaparece en una masa de intelectos que tropiezan unos con otros, que transportan información de un sitio a otro y la hacen circular a la velocidad de la luz.

Lewis Thomas, "Natural Science"; en *The Lives of a Cell*, 1974.

1.10 Perspectiva histórica y lecturas recomendadas

En el CD que acompaña a este libro se puede encontrar una sección dedicada a la perspectiva histórica relacionada con cada capítulo. Podemos exponer el desarrollo de una idea a través de una serie de máquinas o describir algún proyecto importante y dar referencias para aquellos lectores interesados en profundizar más.

La perspectiva histórica para este capítulo proporciona el trasfondo de algunas de las ideas clave presentadas en él. Su propósito es mostrar la historia humana que hay detrás de los avances tecnológicos y poner los descubrimientos en su contexto histórico. Gracias a la comprensión del pasado se pueden entender mejor las líneas maestras que guiarán el futuro. Cada sección de perspectiva histórica del CD acaba con sugerencias para lecturas adicionales, las cuales están recogidas separadamente en el CD bajo la sección "**Lecturas adicionales**". El resto de esta **sección 1.10** está en el CD.

1.11 Ejercicios

Contribución de Javier Bruguera, Universidad de Santiago de Compostela

La mayor parte de los ejercicios de esta edición se han diseñado de modo que constan de una descripción cualitativa apoyada en una tabla que proporciona diferentes parámetros cuantitativos alternativos. Estos parámetros se necesitan para resolver las cuestiones que se plantean en el ejercicio. Las cuestiones individuales pueden resolverse utilizando alguno o todos los parámetros; el número de parámetros que se necesitan para cualquiera de los ejercicios es decisión suya. Por ejemplo, es posible encontrarnos con que se dice "completar la cuestión 4.1.1 utilizando los parámetros de la fila A de la tabla". Alternativamente, los profesores pueden adaptar estos ejercicios para crear nuevas soluciones reemplazando los parámetros dados por otros valores.

El número de ejercicios cuantitativos varía de un capítulo a otro en función de los aspectos tratados en ese capítulo. Cuando esta aproximación cuantitativa no encaja perfectamente, se proponen ejercicios más convencionales.

El tiempo relativo necesario para resolver estos ejercicios se indica entre corchetes detrás del número del ejercicio. De media, un ejercicio catalogado como [10] necesitará el doble de tiempo que uno catalogado como [5]. Por otra parte, la sección que debería haberse leído antes de intentar resolver un ejercicio se indica mediante los signos <>; por ejemplo, <1.3> significa que deberías haber leído la sección 1.3, para resolver este ejercicio.

Ejercicio 1.1

En la lista que se muestra a continuación, encuentre la palabra o frase que mejor se adapte a la descripción de las cuestiones siguientes. Indique la respuesta utilizando los números que aparecen a la izquierda de las palabras. Cada palabra sólo se utiliza una vez.

1.1.1 [2]<1.1> Computador utilizado para ejecutar programas muy complejos y al que se accede habitualmente a través de una red.

1	Mundos virtuales	14	Sistema operativo
2	Computadores personales	15	Compilador
3	Servidores	16	Bit
4	Servidores de gama baja	17	Instrucciones
5	Supercomputadores	18	Lenguaje ensamblador
6	Terabyte	19	Lenguaje máquina
7	Petabyte	20	C
8	Centro de datos	21	Ensamblador
9	Computadores empotrados	22	Lenguaje de alto nivel
10	Procesadores multinúcleo	23	Software del sistema
11	VHDL	24	Software de aplicación
12	RAM	25	Cobol
13	CPU	26	Fortran

1.1.2 [2]<1.1> 10^{15} o 2^{50} bytes.

1.1.3 [2]<1.1> Computador formado por cientos o miles de procesadores con varios terabytes de memoria.

1.1.4 [2] <1.1> Aplicaciones que actualmente son ciencia ficción pero que probablemente serán reales en un futuro próximo.

1.1.5 [2]<1.1> Un tipo de memoria llamado memoria de acceso aleatorio *(random access memory)*.

1.1.6 [2]<1.1> La parte del computador llamada unidad central de procesamiento *(central processor unit)*.

1.1.7 [2]<1.1> Miles de procesadores formando un gran clúster.

1.1.8 [2]<1.1> Un microprocesador compuesto de varios procesadores en el mismo chip.

1.1.9 [2]<1.1> Computador personal sin monitor ni teclado al que se accede habitualmente a través de una red.

1.1.10 [2]<1.1> Sistema diseñado para ejecutar una aplicación concreta o un conjunto de aplicaciones software.

1.1.11 [2]<1.1> Lenguaje utilizado para la descripción de componentes hardware.

1.1.12 [2]<1.1> Computador que proporciona buenas prestaciones a un único usuario y con bajo coste.

1.1.13 [2]<1.2> Programa que traduce sentencias de un lenguaje de alto nivel a lenguaje ensamblador.

1.1.14 [2]<1.2> Programa traduce instrucciones simbólicas a instrucciones binarias.

1.1.15 [2]<1.2> Lenguaje de alto nivel para aplicaciones del ámbito de los negocios y la economía.

1.1.16 [2]<1.2> Lenguaje binario que es capaz de entender un procesador.

1.1.17 [2]<1.2> Órdenes que entiende e interpreta el procesador.

1.1.18 [2]<1.2> Lenguaje de alto nivel para aplicaciones científicas.

1.1.19 [2]<1.2> Representación simbólica de las instrucciones máquina.

1.1.20 [2]<1.2> Interfaz entre los programas de usuario y el hardware, que proporciona varios servicios y funciones de supervisión.

1.1.21 [2]<1.2> Software o programas desarrollados por los usuarios.

1.1.22 [2]<1.2> Dígito binario (valor 0 o 1)

1.1.23 [2]<1.2> Capa software entre el software de aplicación y el hardware, que incluye el sistema operativo y los compiladores.

1.1.24 [2]<1.2> Lenguaje de alto nivel utilizado para escribir software de sistema y de aplicaciones.

1.1.25 [2]<1.2> Lenguaje compuesto de palabras y expresiones algebraicas que deben ser traducidas a lenguaje ensamblador antes de ejecutarse en un computador.

1.1.26 [2]<1.2> 10^{12} o 2^{40} bytes.

Ejercicio 1.2

1.2.1 [10]<1.3> En un monitor de color en el que se utilizan 8 bits por píxel para cada color primario (rojo, verde, azul) y con una resolución de 1280 × 800 píxeles, ¿cuál es el tamaño mínimo (en bytes) del búfer de pantalla necesario para almacenar un cuadro?

1.2.2 [5]<1.3> Un computador con una memoria principal de 2GB, ¿cuántos cuadros puede almacenar si consideramos que no almacena ninguna otra información?

1.2.3 [5]<1.3> Si un computador conectado a una red ethernet de 1 gigabit tiene que enviar un fichero de 256 Kbytes, ¿cuánto tiempo tardará?

1.2.4 [5]<1.3> Teniendo en cuenta que una memoria cache es 10 veces más rápida que una memoria DRAM, que la memoria DRAM es 100 000 veces más rápida que un disco magnético y que la memoria flash es 1000 veces más rápida que el disco, determine cuánto tiempo se necesita para un leer un fichero de una DRAM, un disco y una memoria flash si se necesitan 2 microsegundos para leerlo de la memoria cache.

Ejercicio 1.3

Suponga que se dispone de tres procesadores diferentes, P1, P2 y P3, con las frecuencias de reloj y CPI mostrados en la tabla y que ejecutan el mismo conjunto de instrucciones.

Procesador	Frecuencia de reloj	CPI
P1	2 GHz	1.5
P2	15 GHz	1
P3	3 GHz	2.5

1.3.1 [5]<1.4> ¿Qué procesador tiene mejores prestaciones?

1.3.2 [5]<1.4> Si cada procesador ejecuta un programa que dura 10 segundos, calcule el número de ciclos y el número de instrucciones para cada uno.

1.3.3 [10]<1.4> Se quiere reducir el tiempo de ejecución en un 30%, pero esto sólo se consigue a costa de un incremento del 20% en el CPI. ¿Qué frecuencia de reloj es necesaria para alcanzar esa reducción en el tiempo de ejecución?

Para los siguientes problemas utilice la información de la tabla que se muestra a continuación

Procesador	Frecuencia de reloj	Nº instrucciones	Tiempo
P1	2 GHz	2×10^{10}	7 s
P2	1.5 GHz	30×10^{9}	10 s
P3	3 GHz	90×10^{9}	9 s

1.3.4 [10]<1.4> Calcule el IPC (instrucciones por ciclo) de cada procesador.

1.3.5 [10]<1.4> Calcule la frecuencia de reloj de P2 necesaria para que el tiempo de ejecución sea el de P1

1.3.6 [10]<1.4> Calcule cuántas instrucciones debería ejecutarse en P2 para que el tiempo de ejecución sea el de P3.

Ejercicio 1.4

Considere dos implementaciones diferentes de un mismo repertorio de instrucciones, con cuatro tipos de instrucciones, A, B, C y D. La frecuencia de reloj y el CPI de cada implementación se muestran en la siguiente tabla.

	Frec. reloj	CPI tipo A	CPI tipo B	CPI tipo C	CPI tipo D
P1	1.5 GHz	1	2	3	4
P2	2 GHz	2	2	2	2

1.4.1 [10]<1.4> Dado un programa con 10^6 instrucciones repartidas de la siguiente forma: 10% de tipo A, 20% de tipo B, 50% de tipo C y 20% de tipo D, ¿qué implementación es más rápida?

1.4.2 [5]<1.4> ¿Cuál es el CPI global de cada implementación?

1.4.3 [5]<1.4> Determine el número de ciclos de reloj en ambos casos.

La siguiente tabla muestra el número y tipo de instrucciones de un programa

	Aritméticas	Almacenamiento	Carga	Saltos	Total
P1	500	50	100	50	700

1.4.4 [5]<1.4> Suponiendo que las instrucciones aritméticas necesitan 1 ciclo para ejecutarse, las de carga y almacenamiento 5 ciclos y los saltos 2 ciclos, ¿cuál es el tiempo de ejecución en un procesador a 2 GHz?

1.4.5 [5]<1.4> Determine el CPI del programa.

1.4.6 [10]<1.4> Si el número de instrucciones de carga se reduce a la mitad, ¿cuál es la aceleración y el nuevo CPI?

Ejercicio 1.5

Considere dos implementaciones diferentes, P1 y P2, de un mismo repertorio de instrucciones, con cinco tipos de instrucciones, A, B, C, D y E. La frecuencia de reloj y el CPI de cada implementación se muestran en la siguiente tabla.

		Frec. reloj	CPI tipo A	CPI tipo B	CPI tipo C	CPI tipo D	CPI tipo E
a.	P1	1.0 GHz	1	2	3	4	3
	P2	1.5 GHz	2	2	2	4	4
b.	P1	1.0 GHz	1	1	2	3	2
	P2	1.5 GHz	1	2	3	4	3

1.5.1 [5]<1.4> Se define la prestación pico como la velocidad más elevada a la que un computador puede ejecutar cualquier secuencia de instrucciones. ¿Cuáles son las prestaciones pico de P1 y P2 expresadas en instrucciones por segundo?

1.5.2 [5]<1.4> Supongamos que las instrucciones de un programa se reparten equitativamente entre todos los tipos de instrucciones, excepto para las instrucciones de tipo A, que representan el doble de las instrucciones de cualquiera de los otros tipos. ¿Qué computador es más rápido? ¿Cuál es el aumento de velocidad?

1.5.3 [5]<1.4> Suponga que las instrucciones de un programa se reparten equitativamente entre todos los tipos de instrucciones, excepto para las instrucciones de tipo E, que representan el doble de las instrucciones de cualquiera de los otros tipos. ¿Qué computador es más rápido? ¿Cuál es el aumento de velocidad?

La siguiente tabla muestra el número de instrucciones de varios tipos de instrucciones en dos programas diferentes. Utilizando estos datos, explorar las variaciones de las prestaciones que se producen al aplicar diferentes cambios a un procesador MIPS.

		Nº de instrucciones				
		Cálculo	Carga	Almacenamiento	Salto	Total
a.	Programa 1	1000	400	100	50	1550
b.	Programa 2	1500	300	100	100	2000

1.5.4 [5]<1.4> Suponiendo que las instrucciones de cálculo necesitan 1 ciclo, las de carga y almacenamiento 10 ciclos y los saltos 3 ciclos, determine el tiempo de ejecución de cada programa en un procesador MIPS a 3 GHz.

1.5.5 [5]<1.4> Suponiendo que las instrucciones de cálculo necesitan 1 ciclo, las de carga y almacenamiento 2 ciclos y los saltos 3 ciclos, determine el tiempo de ejecución de cada programa en un procesador MIPS a 3 GHz.

1.5.6 [5]<1.4> Suponiendo que las instrucciones de cálculo necesitan 1 ciclo, las de carga y almacenamiento 10 ciclos y los saltos 3 ciclos, ¿cuál es la aceleración de un programa si el número de instrucciones de cálculo se reduce a la mitad?

Ejercicio 1.6

Los compiladores tienen un fuerte impacto sobre las prestaciones de una aplicación que se ejecuta en un determinado procesador. Este ejercicio analiza el impacto de los compiladores sobre el tiempo de ejecución.

	Compilador A		Compilador B	
	Nº de instrucciones	Tiempo de ejecución	Nº de instrucciones	Tiempo de ejecución
a.	1.00E+09	1 s	1.20E+09	1.4 s
b.	1.00E+09	0.8 s	1.20E+09	0.7 s

1.6.1 [5]<1.4> Se utilizan dos compiladores diferentes para un mismo programa. La tabla anterior muestra los tiempos de ejecución para los dos programas. Determine el CPI medio para cada programa considerando que el ciclo de la señal de reloj del procesador es 1 ns.

1.6.2 [5]<1.4> Suponga el CPI medio obtenido en 1.6.1, pero que los programas compilados se ejecutan en dos procesadores diferentes. Si los tiempos de ejecución en los dos procesadores son iguales, determine la relación entre las frecuencias de reloj del procesador ejecutando el programa compilado con el compilador A y del procesador ejecutando el programa compilado con el compilador B.

1.6.3 [5]<1.4> Se desarrollada un nuevo compilador que solamente genera 600 millones de instrucciones y tiene un CPI medio de 1.1. ¿Cuál es la aceleración obtenida al utilizar este nuevo compilador con respecto a los compiladores A y B de 1.6.1?

Considere dos implementaciones diferentes, P1 y P2, del mismo repertorio de instrucciones con 5 tipos de instrucciones (A, B, C, D y E). La frecuencia de reloj de P1 es 4 GHz y la de P2, 6 GHz. El número medio de ciclos para cada tipo de instrucción se indica en la siguiente tabla.

	Tipo	CPI en P1	CPI en P2
a.	A	1	2
	B	2	2
	C	3	2
	D	4	4
	E	5	4

	Tipo	CPI en P1	CPI en P2
b.	A	1	2
	B	1	2
	C	1	2
	D	4	4
	E	5	4

1.6.4 [5]<1.4> Se definen las prestaciones pico como la velocidad más elevada a la que un computador puede ejecutar cualquier secuencia de instrucciones. ¿Cuáles son las prestaciones pico de P1 y P2 expresadas en instrucciones por segundo?

1.6.5 [5]<1.4> Suponga que las instrucciones de un programa se reparten equitativamente entre todos los tipos de instrucciones, excepto para las instrucciones de tipo A, que representan el doble de las instrucciones de cualquiera de los otros tipos. ¿Cuál es la aceleración de P2 respecto a P1?

1.6.6 [5]<1.4> ¿A qué frecuencia las prestaciones de P2 se igualan a las de P1 para el reparto de instrucciones dado en 1.6.5?

Ejercicio 1.7

La siguiente tabla muestra el aumento de la frecuencia del reloj y potencia disipada para ocho generaciones de procesadores Intel en los últimos 28 años.

Procesador	Frecuencia de reloj	Potencia
80286 (1982)	12.5 MHz	3.3 W
80386 (1985)	16 MHz	4.1 W
80486 (1989)	25 MHz	4.9 W
Pentium (1993)	66 MHz	10.1 W
Pentium Pro (1997)	200 MHz	29.1 W
Pentium 4 Willamette (2001)	2 GHz	75.3 W
Pentium 4 Prescott (2004)	3.6 GHz	103 W
Core 2 Ketsfield (2007)	2667 GHz	95 W

1.7.1 [5]<1.5> Determine la media geométrica de las relaciones entre generaciones consecutivas, tanto para la frecuencia como para la potencia. (La media geométrica se define en la sección 1.7).

1.7.2 [5]<1.5> Indique cuál es el mayor incremento relativo en frecuencia y potencia entre generaciones.

1.7.3 [5]<1.5> ¿Cuál es el aumento de la frecuencia y la potencia de la última generación respecto la primera?

En los siguientes ejercicios se utilizan los valores de voltaje indicados en la siguiente tabla.

Procesador	Voltaje
80286 (1982)	5
80386 (1985)	5
80486 (1989)	5
Pentium (1993)	5
Pentium Pro (1997)	3.3
Pentium 4 Willamette (2001)	1.75
Pentium 4 Prescott (2004)	1.25
Core 2 Ketsfield (2007)	1.1

1.7.4 [5]<1.5> Determine las cargas capacitivas medias, suponiendo que el consumo estático de potencia es despreciable.

1.7.5 [5]<1.5> Determine el mayor incremento relativo en voltaje entre generaciones.

1.7.6 [5]<1.5> Determine la media geométrica de las relaciones de voltaje entre generaciones consecutivas desde el Pentium.

Ejercicio 1.8.

Suponga que se han desarrollado varias versiones de un procesador con las siguientes características

	Voltaje	Frecuencia de reloj
versión 1	5 V	0.5 GHz
versión 2	3.3 V	1 GHz

1.8.1 [5]<1.5> ¿Cuánto se ha reducido la carga capacitiva entre versiones si la potencia dinámica se ha reducido un 10%?

1.8.2 [5]<1.5> ¿Cuánto se ha reducido la potencia dinámica si la carga capacitiva no ha cambiado?

1.8.3 [5]<1.5> Suponiendo que la carga capacitiva de la versión 2 es el 80% de la carga capacitiva de la versión 1, determine el voltaje de la versión 2 si su potencia dinámica es tal que se ha reducido un 40% con respecto a la de la versión 1.

Suponga que las tendencias actuales en la industria son que, con cada nueva generación del proceso de fabricación, las características de un procesador escalan según los datos en la siguiente tabla.

Capacitancia	Voltaje	Frecuencia de reloj	Área
1	$2^{-1/4}$	$2^{1/2}$	$2^{-1/2}$

1.8.4 [5]<1.5> Determine el factor de escalado de la potencia dinámica.

1.8.5 [5]<1.5> Determine el factor de escalado de la capacitancia por unidad de área.

1.8.6 [5]<1.5> Utilizando los datos del ejercicio 1.7, determine el voltaje y la frecuencia de reloj del procesador Core 2 para la nueva generación del proceso de fabricación.

Ejercicio 1.9

Aunque la potencia dinámica es la fuente principal de disipación de potencia en tecnología CMOS, la corriente de fuga ocasiona una disipación de potencia estática, $V \times I_{\text{fuga}}$. Cuanto más pequeñas son las dimensiones del chip, más significativa es la potencia estática. Utilice los datos de la siguiente tabla sobre disipación estática y dinámica de potencia en varias generaciones de procesadores para resolver los siguientes problemas.

	Tecnología	Potencia dinámica (W)	Potencia estática (W)	Voltaje (V)
a.	250 nm	49	1	3.3
b.	90 nm	75	45	1.1

1.9.1 [5]<1.5> Determine que porcentaje de la potencia disipada corresponde a potencia estática.

1.9.2 [5]<1.5> La potencia estática depende de la corriente de fuga, $P_{\text{est}} = V \times I_{\text{fuga}}$. Determine la corriente de fuga para cada tecnología.

1.9.3 [5]<1.5> Determine la relación entre potencia estática y potencia dinámica para cada tecnología.

Considere ahora la disipación de potencia dinámica en diferentes versiones de un procesador para los tres voltajes dados en la siguiente tabla.

	1.2 V	1.0 V	0.8 V
a.	80 W	70 W	40 W
b.	65 W	55 W	30 W

1.9.4 [5]<1.5> Determine la potencia estática a 0.8 V de cada versión considerando que la relación entre potencia estática y dinámica es 0.6.

1.9.5 [5]<1.5> Determine la corriente de fuga a 0.8 V para cada versión.

1.9.6 [5]<1.5> Calcule la mayor corriente de fuga a 1.0 V y 1.2 V considerando que la relación entre potencia estática y dinámica es 1.7.

Ejercicio 1.10

La siguiente tabla muestra los distintos tipos de instrucciones de una aplicación dada que se ejecutan en 1, 2, 4 u 8 procesadores. Utilizando esta información se explorará la aceleración de las aplicaciones en procesadores paralelos.

	Procesadores	Nº instrucciones por procesador			CPI		
		Aritméticas	Carga/almacen.	Salto	Aritméticas	Carga/almacen.	Salto
a.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	4	2
	4	640	320	64	1	4	2
	8	320	160	32	1	4	2

	Procesadores	Nº instrucciones por procesador			CPI		
		Aritméticas	Carga/almacen.	Salto	Aritméticas	Carga/almacen.	Salto
a.	1	2560	1280	256	1	4	2
	2	1350	800	128	1	6	2
	4	800	600	64	1	9	2
	8	600	500	32	1	13	2

1.10.1 [5]<1.4, 1.6> En la tabla anterior se muestra el número de instrucciones por procesador necesarias para la ejecución de un programa en un multiprocesador con 1, 2 4 u 8 procesadores. ¿Cuántas instrucciones se han ejecutado en cada procesador? ¿Cuántas instrucciones adicionales se han ejecutado en cada configuración del multiprocesador?

1.10.2 [5]<1.4, 1.6> Con los valores de CPI de la tabla, determine el tiempo total de ejecución para este programa con 1, 2, 4 y 8 procesadores, suponiendo que la frecuencia del reloj es 2 GHz.

1.10.3 [5]<1.4, 1.6> Si el CPI de las instrucciones aritméticas se duplicase, ¿cuál sería el impacto en los tiempos de ejecución con 1, 2, 4 y 8 procesadores?

La siguiente tabla muestra el número de instrucciones por núcleo en un procesador multinúcleo, junto con el CPI medio de la ejecución de un programa en 1, 2, 4 y 8 núcleos. Utilizando esta información exploraremos la aceleración de una aplicación en procesadores multinúcleo.

	Núcleos por procesador	Instrucciones por núcleo	CPI medio
a.	1	1.00E+10	1.2
	2	5.00E+09	1.3
	4	2.50E+09	1.5
	8	1.25E+09	1.8

	Núcleos por procesador	Instrucciones por núcleo	CPI medio
b.	1	1.00E+10	1.2
	2	5.00E+09	1.2
	4	2.50E+09	1.2
	8	1.25E+09	1.2

1.10.4 [5]<1.4, 1.6> Suponiendo que la frecuencia de la señal de reloj es 3 GHz, ¿cuál es el tiempo de ejecución del programa utilizando 1, 2, 4 y 8 núcleos?

1.10.5 [5]<1.5, 1.6> Suponga que el consumo de potencia de un núcleo está dado por la siguiente ecuación

$$\text{Potencia} = \frac{5.0\ \text{mA}}{\text{MHz}}\ \text{Voltaje}^2$$

siendo el voltaje

$$\text{Voltaje} = \frac{1}{5}\ \text{Frecuencia} + 0.4$$

con la frecuencia medida en GHz. Es decir, a 5 GHz, el voltaje será 1.4 V. Determine el consumo de potencia del programa cuando se ejecuta en 1, 2, 4 y 8 núcleos si cada núcleo opera a 3 GHz. Asimismo, determine el consumo de potencia del programa cuando se ejecuta en 1, 2, 4 y 8 núcleos si cada núcleo está operando a 500 MHz.

1.10.6 [5]<1.5, 1.6> Determine la energía necesaria para ejecutar el programa en 1, 2, 4 y 8 núcleos suponiendo que la frecuencia de reloj de cada núcleo es 3 GHz y 500 MHz. Utilice las ecuaciones de consumo de potencia de 1.10.5.

Ejercicio 1.11

La siguiente tabla muestra los datos de fabricación para varios procesadores.

	Diámetro de la oblea	Dados por oblea	Defectos por unidad de área	Coste por oblea
a.	15 cm	90	0.018 defectos/cm^2	10
b.	25 cm	140	0.024 defectos/cm^2	20

1.11.1 [10]<1.7> Determine el factor de producción.

1.11.2 [10]<1.7> Determine el coste por dado.

1.11.3 [10]<1.7> Si el número de dados por oblea se incrementa en un 10% y los defectos por unidad de área en un 15% , determine el área del dado y el factor de producción.

Suponga que el factor de producción varía con la evolución de la tecnología de fabricación de dispositivos electrónicos según la siguiente tabla,

	T1	T2	T3	T4
factor de producción	0.85	0.89	0.92	0.95

1.11.4 [10] <1.7> Determine los defectos por unidad de área de cada tecnología para un área de dado de 200 mm^2.

1.11.5 [5] <1.7> Represente gráficamente la variación del factor de producción y del número de defectos por unidad de área.

Ejercicio 1.12

La siguiente tabla muestra algunos resultados de la ejecución de los programas de prueba SPEC2006 en el procesador AMD Barcelona.

	Nombre	Nº intr. × 10^9	Tiempo de ejecución (s)	Tiempo de referencia (s)
a.	perl	2118	500	9770
b.	mcf	336	1200	9120

1.12.1 [5] <1.7> Determine el CPI si el ciclo del reloj es 0.333 ns.

1.12.2 [5] <1.7> Determine la razón SPEC.

1.12.3 [5] <1.7> Determine la media geométrica para estos dos programas de prueba.

1.12.4 La siguiente tabla muestra algunos resultados adicionales.

	Nombre	CPI	Frecuencia de reloj	Razón SPEC
a.	sjeng	0.96	4 GHz	14.5
b.	omnetpp	2.94	4 GHz	9.1

1.12.5 [5] <1.7> Calcule el aumento del tiempo de CPU cuando el número de instrucciones de los programas de prueba aumenta un 10% sin afectar al CPI.

1.12.6 [5] <1.7> Calcule el aumento del tiempo de CPU cuando el número de instrucciones de los programas de prueba aumenta un 10% y el CPI un 5%.

1.12.7 [5] <1.7> Determine la razón SPEC con los datos de 1.12.5.

Ejercicio 1.13

Suponga que está desarrollando una nueva versión del procesador AMD Barcelona con un reloj de 4 GHz. Ha incorporado algunas instrucciones adicionales al repertorio de instrucciones de modo que el número de instrucciones de los programas de prueba del ejercicio 1.12 se ha reducido un 15%. Los tiempos de ejecución obtenidos se muestran en la siguiente tabla.

	Nombre	Tiempo de ejecución (s)	Tiempo de referencia (s)	Razón SPEC
a.	perl	450	9770	21.7
b.	mcf	1150	9120	7.9

1.13.1 [10] <1.8> Calcule el nuevo CPI.

1.13.2 [10] <1.8> En general, los CPI obtenidos son mayores que los obtenidos en ejercicios anteriores, debido a las frecuencias de reloj consideradas, 3 GHz y 4 GHz. Compruebe si el incremento del CPI es similar al incremento de la frecuencia. Si son diferentes, explique la causa.

1.13.3 [5] <1.8> ¿Cuál ha sido la reducción en el CPI?

La siguiente tabla muestra más resultados.

	Nombre	Tiempo de ejecución (s)	CPI	Frecuencia de reloj
a.	sjeng	820	0.96	3 GHz
b.	omnetpp	580	2.94	3 GHz

1.13.4 [10] <1.8> Si el tiempo de ejecución se reduce un 10% adicional sin afectar al CPI, determine el número de instrucciones para una frecuencia de 4GHz.

1.13.5 [10] <1.8> Determine la frecuencia de reloj para obtener una reducción adicional del 10% en el tiempo de ejecución manteniendo sin cambios el número de instrucciones y el CPI.

1.13.6 [10] <1.8> Determine la frecuencia de reloj si el CPI se reduce un 15% y el tiempo de CPU un 20%., manteniendo constante el número de instrucciones.

Ejercicio 1.14

En la sección 1.8 se pone como ejemplo de errores habituales la utilización de un subconjunto de las ecuaciones de prestaciones como métrica de las prestaciones. Para ilustrar este error, considere los datos de ejecución de una secuencia de 10^6 instrucciones en diferentes procesadores.

Procesador	Frecuencia de reloj	CPI
P1	4 GHz	1.25
P2	3 GHz	0.75

1.14.1 [5] <1.8> Una falacia habitual es considerar que el computador con la frecuencia más elevada es el que tiene mejores prestaciones. Compruebe si esto es cierto para P1y P2.

1.14.2 [10] <1.8> Otra falacia es creer que el procesador que ejecuta un mayor número de instrucciones necesita más tiempo de CPU. Considerando que el procesador P1 está ejecutando una secuencia de 10^6 instrucciones y que el CPI de P1 y P2 no cambia, calcule el número de instrucciones que P2 puede ejecutar en el mismo tiempo que P1 necesita para ejecutar 10^6 instrucciones

1.14.3 [10] <1.8> Otra falacia habitual se produce al utilizar MIPS (Millones de instrucciones por segundo) para comparar las prestaciones de dos procesadores, y considerar que el procesador con un MIPS más elevado es el que tiene las mejores prestaciones. Compruebe si esto es cierto para P1y P2.

Otra medida habitual de prestaciones es MFLOPS (*million floating-point operations per second*), que se define como

$$\text{MFLOPS} = \text{Nº operaciones PF} / \text{tiempo de ejecución} \times 10^6$$

La utilización de esta métrica presenta unos problemas parecidos a los de la utilización de MIPS. Consideremos los programas de la tabla siguiente en un procesador con frecuencia de reloj de 3 GHz.

	Nº instr.	Instr. L/S	Instr. PF	Instr. salto	CPI(L/S)	CPI(PF)	CPI(salto)
a.	10^6	50%	40%	10%	0.75	1	1.5
b.	3×10^6	40%	40%	20%	1.25	0.70	1.25

1.14.4 [10] <1.8> Calcule los MFLOPS de los programas.

1.14.5 [10] <1.8> Calcule los MIPS de los programas.

1.14.6 [10] <1.8> Calcule las prestaciones de los programas y compárelas con los valores de MIPS y MFLOPS.

Ejercicio 1.15

Otro error habitual citado en la sección 1.8 es pretender mejorar las prestaciones globales de un computador mejorando únicamente un aspecto concreto. Esto puede ser cierto en algunas ocasiones, pero no siempre. Considere un computador que ejecuta programas con los tiempos mostrados en la tabla.

	Instr. PF	Instr. ENT	Instr. L/S	Instr. salto	Tiempo total
a.	35 ns	85 ns	50 ns	30 ns	200 ns
b.	50 ns	80 ns	50 ns	30 ns	210 ns

1.15.1 [5] <1.8> Calcule cuánto se reduce el tiempo total si el tiempo de las operaciones punto flotante (PF) se reduce un 20%.

1.15.2 [5] <1.8> Calcule cuánto se reduce el tiempo de las operaciones enteras (ENT) si el tiempo total se reduce un 20%.

1.15.3 [5] <1.8> Determine si el tiempo total se puede reducir un 20% reduciendo únicamente el tiempo de las instrucciones de salto.

La siguiente tabla muestra los distintos tipos de instrucciones por procesador de una aplicación dada que se ejecuta en números diferentes de procesadores.

	Nº proc.	Instr. PF	Instr. ENT	Instr. L/S	Instr. salto	CPI (PF)	CPI (ENT)	CPI (L/S)	CPI (salto)
a.	1	560×10^6	2000×10^6	1280×10^6	256×10^6	1	1	4	2
b.	8	80×10^6	240×10^6	160×10^6	32×10^6	1	1	4	2

Suponga que la frecuencia de reloj de cada procesador es 2 GHz.

1.15.4 [10] <1.8> ¿Cuánto hay que mejorar el CPI de las instrucciones PF si se quiere que el programa se ejecute el doble de rápido?

1.15.5 10] <1.8> ¿Cuánto hay que mejorar el CPI de las instrucciones L/S si se quiere que el programa se ejecute el doble de rápido?

1.15.6 [5] <1.8> ¿Cuánto se mejora el tiempo de ejecución si el CPI de las instrucciones PF y ENT se reduce un 40% y el CPI de las instrucciones L/S y salto se reduce un 30%?

Ejercicio 1.16

Otro error común está relacionado con la ejecución de programas en un sistema multiprocesador, cuando se intenta mejorar las prestaciones mejorando únicamente el tiempo de ejecución de parte de las rutinas. La siguiente tabla indica el tiempo de ejecución de cinco rutinas de un programa cuando se ejecutan en varios procesadores.

	Nº proc.	Rutina A (ms)	Rutina B (ms)	Rutina C (ms)	Rutina D (ms)	Rutina E (ms)
a.	2	20	80	10	70	5
b.	16	4	14	2	12	2

1.16.1 [10] <1.8> Obtenga el tiempo de ejecución total y cuánto se reduce si el tiempo de las rutinas A, C y E se reduce un 15%.

1.16.2 [10] <1.8> ¿Cuánto se mejora el tiempo total si el tiempo de la rutina B se reduce un 10%?

1.16.3 [10] <1.8> ¿Cuánto se mejora el tiempo total si el tiempo de la rutina D se reduce un 10%?

El tiempo de ejecución en un sistema multiprocesador puede dividirse en tiempo de cálculo de la rutinas y tiempo de comunicaciones empleado en el envío de datos de un procesador a otro. Considere los tiempos de cálculo y de comunicación de la tabla. En este caso, el tiempo de comunicaciones es significativo.

Nº proc.	Rutina A (ms)	Rutina B (ms)	Rutina C (ms)	Rutina D (ms)	Rutina E (ms)	Comun. (ms)
2	20	78	9	65	4	11
4	12	44	4	34	2	13
8	1	23	3	19	3	17
16	4	13	1	10	2	22
32	2	5	1	5	1	23
64	1	3	0.5	1	1	26

1.16.4 [10] <1.8> Determine las relación entre los tiempos de cálculo y los tiempos de comunicación cada vez que se dobla el número de procesadores.

1.16.5 [5] <1.8> Utilizando la media geométrica de las relaciones, extrapole el tiempo de cálculo y el tiempo de comunicación para un sistema con 128 procesadores.

1.16.6 [10] <1.8> Determine el tiempo de cálculo y el tiempo de comunicación para un sistema con un procesador.

Respuestas a las autoevaluaciones

1.1, página 9. Cuestiones de discusión: Son válidas varias respuestas.
1.3, página 25. Memoria en disco: no volátil, tiempo de acceso elevado (milisegundos) y coste entre 0.2 y 2.0 dólares/GB. Memoria semiconductor: volátil, tiempo de acceso reducido (naosegundos) y coste entre 20 y 75 dólares/GB
1.4, página 31. 1.a ambos, b: latencia, c: ninguno, 2.7 segundos
1.4, página 38: b
1.7, página 50. 1, 3 y 4 son razones válidas. Respuesta 5 es generalmente cierta porque el elvado volumen de fabricación puede hacer que la inversión extra para reducir el tamaño del dado, por ejemplo un 10%, sea una buena decisión económica, pero no tiene por qué ser cierto.
1.8, página 53. a. El computador con mayor MIPS es el A. b. El computador más rápido es el B.

2

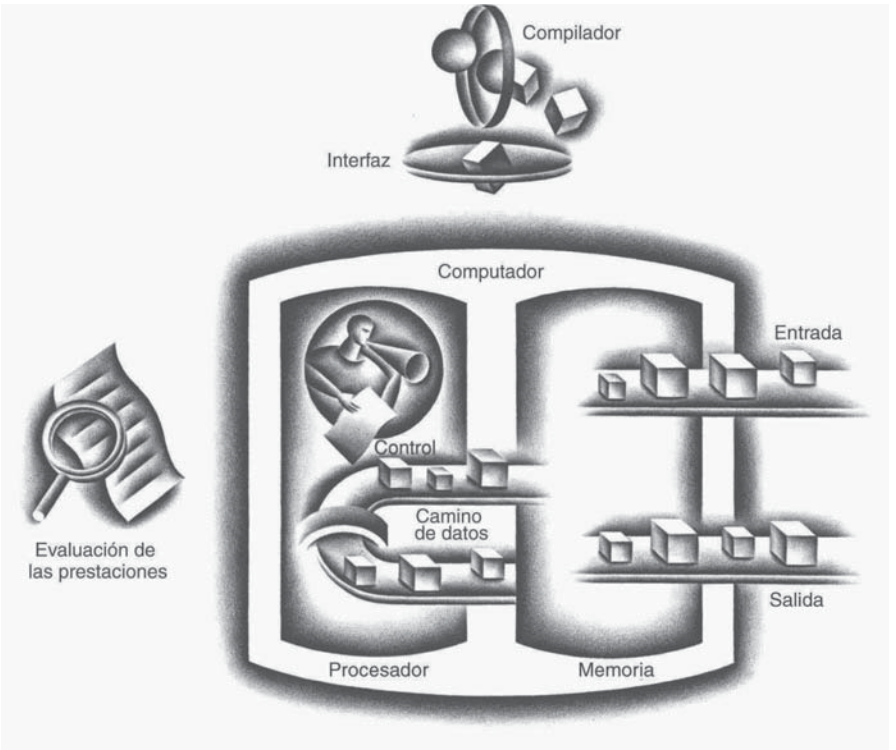
Instrucciones: el lenguaje del computador

Yo hablo castellano con Dios, italiano con las mujeres, francés con los hombres y alemán con mi caballo.

Carlos V, rey de Francia
1337–1380

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco componentes clásicos de un computador



2.1 Introducción

Repertorio de instrucciones: vocabulario de comandos entendidos por una arquitectura concreta.

Para dominar el hardware de un computador debemos hablar su lenguaje. Las palabras del lenguaje del computador se denominan *instrucciones*, y su vocabulario se denomina **repertorio de instrucciones**. En este capítulo veremos el repertorio de instrucciones de un computador real, en la forma escrita por los humanos y en la forma leída por el computador. Introduciremos instrucciones desde el nivel alto hasta el nivel bajo (*top-down*): comenzando desde una notación que parece un lenguaje de programación limitado, la refinaremos paso a paso hasta que veamos el lenguaje real de un computador real. El capítulo 3 continúa en el mismo orden descendente, y muestra la representación de números enteros y en punto flotante y el hardware que opera con ellos.

Podríamos pensar que los lenguajes de los computadores son tan diversos como los distintos idiomas de los humanos, pero en realidad son bastante similares, más parecidos a dialectos regionales que a lenguas independientes. Por tanto, una vez que se aprende uno, es fácil aprender otros. Esta similitud ocurre porque todos los computadores utilizan tecnologías hardware similares basadas en principios similares y porque hay un conjunto de operaciones básicas que todos los computadores deben proporcionar. Por otra parte, los diseñadores de computadores comparten una meta común: encontrar un lenguaje que haga fácil construir el hardware y el compilador, a la vez que se maximicen las prestaciones y se minimice el coste y la potencia consumida. Esta meta ha sido una constante a lo largo del tiempo. La siguiente cita fue escrita antes de que se pudiera comprar un computador, y es tan verdad hoy como lo fue en 1947:

> *Es fácil ver mediante métodos lógico-formales que existen ciertos [repertorios de instrucciones] que son adecuados en abstracto para controlar y ejecutar cualquier secuencia de operaciones… Las consideraciones realmente decisivas, desde el punto de vista actual, al seleccionar un [repertorio de instrucciones] son más bien de naturaleza práctica: la simplicidad del equipo pedido por el [repertorio de instrucciones] y la claridad de su aplicación a los problemas realmente importantes, junto con la velocidad en el manejo de esos problemas.*
>
> Burks, Goldstine y von Neumann, 1947

La "simplicidad del equipo" es una consideración tan valiosa para los computadores de los años 2000 como lo fue para los de los años 50. La finalidad de este capítulo es enseñar un repertorio de instrucciones que siga esta premisa: mostrar cómo se representa en el hardware y la relación entre el lenguaje de programación de alto nivel y el lenguaje del computador más primitivo. En nuestros ejemplos utilizaremos el lenguaje de programación C. La **sección 2.15** en el CD muestra cómo éstos cambiarían para un lenguaje orientado a objetos tipo Java.

Al aprender a representar instrucciones también descubriremos el secreto de la computación: el **concepto de programa almacenado**. Además, ejercitaremos nuestra habilidad con un "idioma extranjero" escribiendo programas en el lenguaje del computador y ejecutándolos en el simulador que proporcionamos con este libro. También veremos el impacto de los lenguajes de programación y la optimización del compilador en las prestaciones. Concluiremos con un vistazo a la evolución histórica del repertorio de instrucciones y con un repaso de otros dialectos del computador.

Concepto de programa almacenado: la idea de que instrucciones y datos de diferentes tipos se pueden almacenar en memoria como números nos lleva al concepto de computador de *programa almacenado*.

El repertorio de instrucciones elegido proviene de MIPS, un repertorio de instrucciones típico diseñado a partir de 1980. Más adelante, daremos una visión rápida de otros dos repertorios de instrucciones muy populares. ARM es bastante similar a MIPS y en 2008 se han vendido más de 3000 millones de procesadores ARM en dispositivos empotrados. El otro ejemplo, Intel x86, está en el interior de la mayoría de los 330 millones de PCs fabricados en 2008.

Mostraremos el repertorio de instrucciones MIPS por partes, una cada vez, relacionándola con la estructura del computador. Este tutorial *top-down* relaciona paso a paso los componentes con sus explicaciones, haciendo el lenguaje ensamblador más digerible. La figura 2.1 da una visión general del repertorio de Instrucciones de este capítulo.

2.2 Operaciones del hardware del computador

"Ciertamente, deben existir instrucciones para realizar las operaciones aritméticas fundamentales."

Burks, Goldstine y von Neumann, 1947

Todo computador debe ser capaz de realizar cálculos aritméticos. La notación del lenguaje ensamblador MIPS:

```
add a, b, c
```

indica a un computador que sume las variables `b` y `c` y que coloque el resultado en `a`.

Esta notación es rígida en el sentido de que cada instrucción aritmética MIPS lleva a cabo sólo una operación y debe tener siempre exactamente tres variables. Por ejemplo, Suponga que queremos colocar la suma de las variables `b`, `c`, `d` y `e` en la variable `a` (en esta sección se está siendo deliberadamente vago sobre lo que es una "variable" ; en la sección siguiente lo explicaremos en detalle).

La siguiente secuencia de instrucciones suma las cuatro variables:

```
add a, b, c     # la suma de b y c se coloca en a
add a, a, d     # la suma de b, c y d está ahora en a
add a, a, e     # la suma de b, c, d y e está ahora en a
```

Por tanto, se necesitan tres instrucciones para realizar la suma de cuatro variables.

Las palabras a la derecha del símbolo sostenido (#) en cada una de las líneas anteriores son *comentarios* para el lector y son ignorados por el computador. Observe que, a diferencia de otros lenguajes de programación, cada línea de este lenguaje puede contener como mucho una instrucción. Otra diferencia con el lenguaje C es que los comentarios siempre terminan al final de una línea.

Operandos MIPS

Nombre	Ejemplo	Comentarios
32 registros	`$s0–$s7`, `$t0–$t9`, `$zero`, `$a0–$a3`, `$v0–$v1`, `$gp`, `$fp`, `$sp`, `$ra`, `$at`	Localizaciones rápidas para los datos. En MIPS, los datos deben estar en los registros para realizar operaciones aritméticas. El registro MIPS `$zero` es siempre igual a 0. El registro `$at` está reservado por el ensamblador para manejar constantes grandes.
2^{30} palabras de memoria	Memory[0], Memory[4], . . . , Memory[4294967292]	Accesibles solamente por instrucciones de transferencia de datos. MIPS utiliza direcciones de byte, de modo que las direcciones de palabras consecutivas se diferencian en 4. La memoria guarda las estructuras de datos, las tablas y los registros desbordados (guardados).

Lenguaje ensamblador MIPS

Categoría	Instrucción	Ejemplo	Significado	Comentarios
Aritmética	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Tres operandos; datos en registros
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Tres operandos; datos en registros
	add immediate	`addi $s1,$s2,100`	`$s1` = `$s2` + 100	Usado para sumar constantes
Transferencia de dato	load word	`lw $s1,100($s2)`	`$s1` = Memory[`$s2` + 100]	Palabra de memoria a registro
	store word	`sw $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Palabra de registro a memória
	load half	`lh $s1,100($s2)`	`$s1` = Memory[`$s2` + 100]	Media palabra de memoria a registro
	store half	`sh $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Media palabra de registro a memoria
	load byte	`lb $s1,100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte de memoria a registro
	store byte	`sb $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte de registro a memoria
	load upper immed.	`lui $s1,100`	`$s1` = 100 * 2^{16}	Cargar constante en los 16 bits de mayor peso
Lógica	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Tres registros operandos; AND bit-a-bit
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Tres registros operandos; OR bit-a-bit
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` |`$s3`)	Tres registros operandos; NOR bit-a-bit
	and immediate	`andi $s1,$s2,100`	`$s1` = `$s2` & 100	AND Bit-a-bit registro con constante
	or immediate	`ori $s1,$s2,100`	`$s1` = `$s2` | 100	OR Bit-a-bit registro con constante
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Desplazamiento a la izquierda por constante
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Desplazamiento a la derecha por constante
Salto condicional	branch on equal	`beq $s1,$s2,L`	if (`$s1` == `$s2`) go to `L` PC + 4 + 100	Comprueba igualdad y bifurca relativo al PC
	branch on not equal	`bne $s1,$s2,L`	if (`$s1` != `$s2`) go to `L` PC + 4 + 100	Comprueba si no igual y bifurca relativo al PC
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else $s1 = 0	Compara si es menor que; usado con beq, bne
	set on less than immediate	`slt $s1,$s2,100`	if (`$s2` < 100) `$s1` = 1; else `$s1` = 0	Compara si es menor que una constante
Salto incondicional	jump	`j 2500`	go to 10000	Salto a la dirección destino
	jump register	`jr $ra`	go to `$ra`	Para retorno de procedimiento
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	Para llamada a procedimiento

FIGURA 2.1 Lenguaje ensamblador MIPS tratado en este capítulo. Esta información se encuentra también en la Tarjeta de Datos de Referencia de MIPS, que se incluye con este libro.

El número natural de operandos para una operación como la suma es tres: los dos números que se suman juntos y el lugar donde se coloca el resultado de la suma. El hecho de que cada instrucción requiera tener exactamente tres operandos, ni más ni menos, está de acuerdo con la filosofía de mantener el hardware sencillo: el hardware para un número de operandos variable es más complejo que el hardware para un número fijo. Esta situación ilustra el primero de los cuatro principios fundamentales del diseño de hardware:

Principio de diseño 1: la simplicidad favorece la regularidad.

Ahora podemos mostrar, en los dos ejemplos siguientes, la relación que existe entre los programas escritos en lenguaje de programación de alto nivel y los programas escritos en esta notación más primitiva.

Compilación de dos sentencias de asignación C en MIPS

EJEMPLO

Este segmento de un programa en C contiene cinco variables: `a`, `b`, `c`, `d` y `e`. Puesto que Java se desarrolló a partir de C, este ejemplo y los siguientes son válidos para ambos lenguajes de programación de alto nivel.

```
a = b + c;
d = a - e;
```

La traducción de C a instrucciones del lenguaje ensamblador MIPS la realiza el *compilador*. Mostrar el código MIPS producido por un compilador.

RESPUESTA

Una instrucción MIPS trabaja con dos operandos fuente y coloca el resultado en un operando destino. Por tanto, las dos sentencias simples anteriores se compilan directamente en estas dos instrucciones del lenguaje ensamblador MIPS:

```
add a, b, c
sub d, a, e
```

Compilación de una sentencia C compleja en MIPS

EJEMPLO

Una sentencia un tanto compleja contiene cinco variables: `f`, `g`, `h`, `i` y `j`: donde:

```
f = (g + h) - (i + j);
```

¿Qué debería producir un compilador de C?

RESPUESTA

El compilador debe fragmentar esta sentencia en varias instrucciones de ensamblador, puesto que se realiza sólo una operación por cada instrucción MIPS. La primera instrucción MIPS calcula la suma de `g` y `h`. Debemos colocar el resultado en algún lugar, por tanto el compilador crea una variable temporal llamada `t0`:

```
add t0,g,h # la variable temporal t0 contiene g + h
```

Aunque la siguiente operación en C es una resta, necesitamos calcular la suma de `i` y `j` antes de poder restar. Por tanto, la segunda instrucción coloca la suma de `i` y `j` en otra variable temporal creada por el compilador llamada `t1`:

```
add t1,i,j # la variable temporal t1 contiene i + j
```

Finalmente, la instrucción de restar resta la segunda suma de la primera y coloca la diferencia en la variable `f`, completando el código compilado:

```
sub f,t0,t1 # f obtiene t0 - t1, que es (g + h)-(i + j)
```

Autoevaluación Para una función dada, ¿qué lenguaje de programación lleva probablemente más líneas de código? Poner las tres representaciones siguientes en orden:

1. Java
2. C
3. Lenguaje ensamblador MIPS

Extensión: Para incrementar la portabilidad, Java fue originalmente concebido como dependiente de un software intérprete. El repertorio de instrucciones de este intérprete se denomina Java bytecodes (véase **sección 2.15** en el CD), y es completamente distinto del repertorio de instrucciones MIPS. Para conseguir unas prestaciones cercanas al programa C equivalente, los sistemas Java típicos actuales compilan los Java bytecodes en un repertorio de instrucciones nativo parecido a MIPS. Debido a que esta compilación se hace normalmente mucho más tarde que en los programas C, tales compiladores Java son frecuentemente llamados compiladores *Just-In-Time* (JIT). La sección 2.12 muestra cómo los compiladores JIT se utilizan más tarde que los compiladores C en el proceso de arranque del programa, y la sección 2.13 muestra las consecuencias en las prestaciones de compilación frente a las de interpretación de los programas Java.

2.3 Operandos del hardware del computador

A diferencia de los programas en lenguajes de alto nivel, los operandos de las instrucciones aritméticas tienen restricciones; debemos usar un número limitado de posiciones especiales construidas directamente en hardware llamados *registros*.

Los registros son las piezas para la construcción de un computador: son elementos básicos usados en el diseño hardware que son también visibles al programador cuando el computador está terminado. El tamaño de un registro en la arquitectura MIPS es de 32 bits; estos grupos de 32 bits son tan frecuentes que se les ha dado el nombre de **palabra**.

Palabra: unidad natural de acceso en un computador; normalmente un grupo de 32 bits; corresponde al tamaño de un registro en la arquitectura MIPS.

Una diferencia importante entre las variables de un lenguaje de programación y los registros es su número limitado, usualmente 32, en los computadores actuales como MIPS. (Véase la **sección 2.20** en el CD para conocer la historia del número de registros.) Así, continuando de arriba hacia abajo (*top-down*) con el desarrollo de la representación simbólica del lenguaje MIPS, en esta sección hemos añadido la restricción de que cada uno de los tres operandos de las instrucciones aritméticas MIPS debe elegirse a partir de uno de los 32 registros de 32 bits.

La razón del límite de 32 registros se puede encontrar en el segundo de nuestros cuatro principios fundamentales del diseño de hardware:

Principio de diseño 2: Cuanto más pequeño, más rápido.

Un número muy grande de registros puede aumentar la duración del ciclo de reloj, simplemente porque se necesitan señales electrónicas más largas cuando deben viajar más lejos.

Directrices tales como "cuanto más pequeño, más rápido" no son absolutas; 31 registros pueden no ser más rápidos que 32. Aun así, la verdad que hay detrás de tales observaciones obliga a los diseñadores de computadores a tomarlas en serio. En este caso, el diseñador debe equilibrar el ansia de programar con más registros con el deseo de mantener el ciclo de reloj rápido. Otra razón para no usar más de 32 registros es el número de bits que ocuparía en el formato de instrucción, como muestra la sección 2.5.

El capítulo 4 muestra el importante papel de los registros en la construcción del hardware. Como veremos en este capítulo, el uso eficaz de los registros es clave para las prestaciones de los programas.

Aunque podríamos escribir instrucciones simplemente usando números para los registros, de 0 a 31, el convenio para MIPS es utilizar nombres de dos caracteres precedidos por un signo de dólar para representar un registro. En la sección 2.8 explicaremos las razones que hay detrás de esta nomenclatura. Por ahora, utilizaremos `$s0`, `$s1`, ... para los registros que corresponden a variables en los programas en C y en Java, y `$t0`, `$t1`, ... para los registros temporales usados para compilar en instrucciones MIPS.

EJEMPLO

Compilación de una sentencia de asignación en C usando registros

Es trabajo del compilador asociar las variables del programa a los registros. Tomemos, por ejemplo, la sentencia de asignación de nuestro ejemplo anterior:

```
f = (g + h) - (i + j);
```

Las variables `f`, `g`, `h`, `i` y `j` se asignan a los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4`, respectivamente. ¿Cuál es el código MIPS compilado?

RESPUESTA

El programa compilado es muy similar al ejemplo anterior, excepto que ahora reemplazamos las variables por los nombres de registros mencionados anteriormente más los dos registros temporales `$t0` y `$t1`, que corresponden a las variables temporales

```
add $t0,$s1,$s2 # el registro $t0 contiene g + h
add $t1,$s3,$s4 # el registro $t1 contiene i + j
sub $s0,$t0,$t1 # f se carga con $t0 - $t1, que es
                # (g + h)-(i + j)
```

Operandos en memoria

Los lenguajes de programación tienen variables simples que contienen elementos con datos únicos, como en los ejemplos anteriores; pero también tienen estructuras de datos más complejas, como tablas (*arrays*) y estructuras. Estas estructuras complejas pueden contener muchos más datos que el número de registros que hay en un computador. ¿Cómo puede un computador representar y acceder a estructuras tan grandes?

Recordemos los cinco componentes de un computador introducidos en el capítulo 1 y representados en la página 75. El procesador puede mantener solamente una cantidad pequeña de datos en los registros, pero la memoria del computador contiene millones de datos. Por tanto, las estructuras de datos (tablas y estructuras) se guardan en memoria.

Según lo explicado anteriormente, las operaciones aritméticas se producen sólo entre registros en las instrucciones MIPS. Así, MIPS debe incluir instrucciones que transfieran datos entre la memoria y los registros. Tales instrucciones son llamadas **instrucciones de transferencia de datos**. Para acceder a una palabra en memoria, la instrucción debe proporcionar la **dirección** de memoria. La memoria es simplemente una gran tabla unidimensional, y la dirección actúa como índice de esa tabla y empieza por 0. Por ejemplo, en la figura 2.2 la dirección del tercer elemento de datos es 2, y el valor de Memoria[2] es 10:

Instrucciones de transferencia de datos: comandos que mueven datos entre memoria y registros.

Dirección: valor usado para señalar la posición de un elemento de datos específico dentro de una memoria.

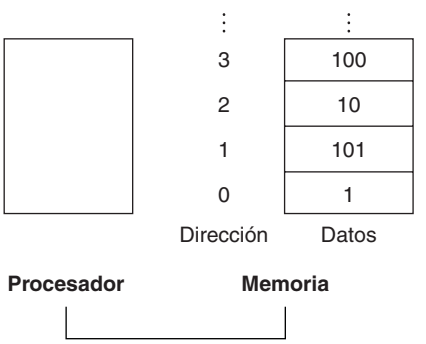


FIGURA 2.2 Direcciones de memoria y contenidos de memoria en esas posiciones. Esto es una simplificación del direccionamiento MIPS. La figura 2.3 muestra el direccionamiento MIPS real para direcciones de palabras secuenciales en memoria.

La instrucción de transferencia de datos que copia datos de la memoria a un registro se llama tradicionalmente *load* (carga). El formato de la instrucción de carga (*load*) es el nombre de la operación seguido por el registro que se cargará, una constante y el registro usado para acceder a la memoria. La dirección de memoria se forma sumando la parte constante de la instrucción y el contenido del segundo registro. El nombre MIPS real para esta instrucción es *lw*, contracción de *load word* (cargar palabra).

Compilación de una sentencia de asignación cuando un operando está en memoria

EJEMPLO

Suponga que *A* es una tabla de 100 palabras y que, como antes, el compilador ha asociado las variables `g` y `h` a los registros `$s1` y `$s2`. Suponga también que la dirección de comienzo o *dirección base* de la tabla está en `$s3`. Compilar esta sentencia de asignación en C:

```
g = h + A[8];
```

RESPUESTA

Aunque hay una sola operación en esta sentencia de asignación, uno de los operandos está en memoria, así que debemos hacer primero la transferencia de `A[8]` a un registro. La dirección de este elemento de la tabla es la suma de la base de la tabla *A*, que se encuentra en el registro `$s3`, más el número para seleccionar el elemento 8. El dato debe colocarse en un registro temporal para usarlo en la instrucción siguiente. De acuerdo con la figura 2.2, la primera instrucción compilada es:

```
lw $t0,8($s3)# el registro temporal $t0 toma el valor
             # de A[8]
```

(En la página siguiente haremos un pequeño ajuste a esta instrucción, pero por ahora utilizaremos esta versión simplificada.) La instrucción siguiente puede operar con el valor `$t0` (que contiene `A[8]`), puesto que está en un registro. La instrucción debe sumar `h` (contenido en `$s2`) con `A[8]` (`$t0`) y colocar el resultado en el registro que corresponde a `g` (asociado con `$s1`):

```
add  $s1,$s2,$t0 # g = h + A[8]
```

La constante en una instrucción de transferencia de datos se llama *desplazamiento*, y el registro añadido para formar la dirección se llama *registro base*.

Interfaz hardware software

Además de asociar variables a los registros, el compilador asigna las estructuras de datos, tales como tablas y estructuras, a posiciones de memoria. Con esto el compilador puede situar la dirección de comienzo adecuada en las instrucciones de transferencia de datos.

Puesto que los *bytes* (8 bits) se utilizan en muchos programas, la mayoría de las arquitecturas pueden direccionar bytes individuales. Por tanto, la dirección de una palabra coincide con la dirección de uno de sus 4 bytes. Así pues, las direcciones de palabras consecutivas difieren en 4 unidades. Por ejemplo, la figura 2.3 muestra las direcciones MIPS reales para la figura 2.2; la dirección del byte de la tercera palabra es 8.

Restricción de la alineación: requisito de que los datos se alineen en memoria en límites naturales.

En MIPS, las palabras deben comenzar en direcciones múltiplos de 4. Este requisito se llama **restricción de la alineación**, y muchas arquitecturas la tienen. (El capítulo 4 explica por qué la alineación facilita transferencias de datos más rápidas).

Los computadores se dividen en aquellos que utilizan la dirección del byte del extremo izquierdo como la dirección de la palabra —el extremo mayor o *big end*— y aquellos que utilizan el byte de más a la derecha —el extremo menor o *little end*—. MIPS usa el *extremo mayor* (es un *Big Endian*). (El apéndice B, muestra las dos opciones de la numeración de los bytes en una palabra).

El direccionamiento de byte también afecta al índice de la tabla (*array*). Para conseguir la dirección apropiada del byte en el código anterior, *el desplazamiento que se añadirá al registro base* `$s3` *debe ser 4 × 8, ó 32*, de modo que la dirección cargada sea `A[8]` y no `A[8/4]`. (Véase la trampa de la sección 2.18, en la página 175.)

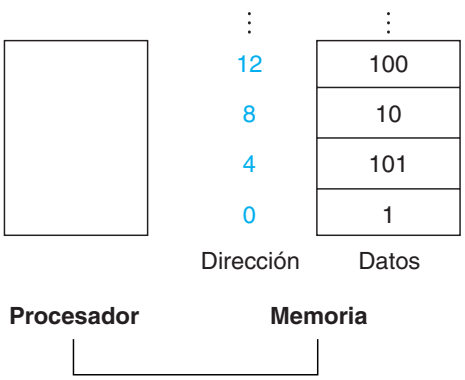


FIGURA 2.3 Direcciones de memoria reales MIPS y contenido de la memoria para dichas palabras. Las direcciones que han cambiado están resaltadas para comparar con las de la figura 2.2. Puesto que MIPS direcciona cada byte, las direcciones de las palabras son múltiplos de cuatro, ya que hay 4 bytes en cada palabra.

La instrucción complementaria de *load* (cargar) se ha llamado tradicionalmente *store* (almacenar), y copia los datos de un registro en la memoria. El formato de *store* es similar al de *load*: el nombre de la operación, seguido por el registro a almacenar, el desplazamiento para seleccionar el elemento de la tabla y finalmente el registro base. De nuevo, la dirección de MIPS se especifica en parte por una constante y en parte por el contenido de un registro. El nombre MIPS real es *sw*, contracción de *store word* (almacenar palabra).

Compilación usando *load* y *store*

EJEMPLO

Suponga que la variable `h` está asociada al registro `$s2` y la dirección base de la tabla `A` está en `$s3`. ¿Cuál es el código ensamblador MIPS para la siguiente sentencia C de asignación?

```
A[12] = h + A[8];
```

RESPUESTA

Aunque hay una sola operación en esta declaración de C, dos de los operandos están en memoria, así que necesitamos aún más instrucciones MIPS. Las dos primeras instrucciones son las mismas que en el ejemplo anterior, excepto que esta vez utilizamos el desplazamiento apropiado para el direccionamiento de byte en la instrucción de *load word* para seleccionar `A[8]`, y la instrucción *add* coloca la suma en `$t0`:

```
lw  $t0,32($s3)# el registro temporal $t0 toma el
               # valor de A[8]

add $t0,$s2,$t0 # el registro temporal $t0 toma el
                # valor de h + A[8]
```

La instrucción final almacena la suma en A[12], utilizando 48 (4 × 12) como desplazamiento y $s3 como registro base.

```
sw  $t0,48($s3) # almacena h + A[8] en A[12]
```

Cargar palabra y almacenar palabra son las instrucciones que permiten copiar palabras entre memoria y registros en la arquitectura MIPS. Otros computadores tienen otras instrucciones además de cargar y almacenar para transferir datos. Una de estas arquitecturas es la Intel x86, que se describe en la sección 2.17.

Interfaz hardware software

Muchos programas tienen más variables que registros tienen los computadores. Por tanto, el compilador intenta mantener las variables más frecuentemente usadas en registros y coloca el resto en memoria, y utiliza *load* y *store* para mover variables entre los registros y la memoria. El proceso de poner las variables usadas con menos frecuencia (o aquellas que necesitaremos más tarde) en memoria se llama *spilling* (desbordar los registros).

El principio hardware que relaciona tamaño y velocidad sugiere que la memoria debe ser más lenta que los registros, puesto que los registros son de menor tamaño. Éste es, efectivamente, el caso: los accesos a los datos son más rápidos si los datos están en los registros en vez de en la memoria.

Además, el dato es más útil cuando está en un registro. Una instrucción aritmética MIPS puede leer dos registros, operar con ellos y escribir el resultado. Una instrucción de transferencia de datos MIPS lee o escribe solamente un operando, sin operar con él.

Así, los registros MIPS necesitan menos tiempo de acceso *y tienen* mayor productividad que la memoria (una rara combinación), al hacer que sea más rápido acceder a los datos de los registros y que éstos sean más fáciles de usar. Para alcanzar las mayores prestaciones los compiladores deben utilizar eficientemente los registros.

Operandos constantes o inmediatos

Muchas veces un programa utilizará una constante en una operación —por ejemplo, incrementando un índice para señalar el elemento siguiente de una tabla—. De hecho, más de la mitad de las instrucciones aritméticas MIPS tienen una constante como operando cuando ejecutan los programas de prueba (*benchmark*) SPEC2006.

Si usamos solamente las instrucciones que hemos visto hasta ahora, tendríamos que cargar alguna constante de la memoria para poder utilizar una (las constantes habrían sido puestas en memoria cuando el programa fue cargado). Por ejemplo, para sumar la constante 4 al registro `$s3` podríamos utilizar el código:

```
lw  $t0, AddrConstant4($s1)# $t0 = constante 4
add $s3,$s3,$t0            # $s3 = $s3 + $t0 ($t0 == 4)
```

suponiendo que `AddrConstant4` es la dirección de memoria de la constante 4.

Un alternativa que evita la instrucción de carga es ofrecer versiones de las instrucciones aritméticas en las cuales un operando es una constante. Esta instrucción de suma rápida con un operando constante se llama *suma inmediata (add immediate)* o `addi`. Para sumar 4 al registro `$s3` escribimos:

```
addi    $s3,$s3,4   # $s3 = $s3 + 4
```

Las instrucciones inmediatas ilustran el tercer principio del diseño de hardware, citado anteriormente en los errores y trampas del capítulo 1:

Principio de diseño 3: Hacer rápido lo común (lo más frecuente).

Los operandos-constante son muy frecuentes, y al incluir las constantes en la operación aritmética, las operaciones son mucho más rápidas y requieren menos energía que si las constantes fuesen cargadas desde memoria.

El valor constante cero juega otro papel: ofrecer variaciones útiles para simplificar el repertorio de instrucciones. Por ejemplo, la operación mover es simplemente una suma con un operando igual a cero. Por lo tanto, MIPS tiene un registro, `$zero`, con el valor cero fijado por hardware. (Como es de esperar, es el registro número 0.)

Autoevaluación

Dada la importancia de los registros, ¿cuál es el coeficiente de incremento en el número de registros en un chip con relación al tiempo?

1. Muy rápido: aumentan tan rápido como la ley de Moore, que predice que se dobla el número de transistores en un chip cada 18 meses.
2. Muy lento: puesto que los programas generalmente se distribuyen en el lenguaje del computador, existe cierta inercia en la arquitectura del repertorio de instrucciones, y por eso el número de registros aumenta sólo tan rápido como los nuevos repertorios de instrucciones llegan a ser viables.

Extensión: Aunque los registros MIPS en este libro son de 32 bits de ancho hay una versión del repertorio de instrucciones de MIPS con 32 registros de 64 bits de ancho. Para distinguirlos claramente son llamados oficialmente MIPS-32 y MIPS-64. En este capítulo utilizamos un subconjunto de MIPS-32. El **apéndice E** en el CD muestra las diferencias entre MIPS-32 y MIPS-64.

El modo de direccionamiento MIPS, registro base más desplazamiento, es una combinación excelente para tratar tanto las estructuras como las tablas, ya que el registro puede señalar el inicio de la estructura y el desplazamiento puede seleccionar el elemento deseado. Veremos un ejemplo de esto en la sección 2.13.

El registro, en las instrucciones de transferencia de datos, fue inventado originalmente para mantener un índice de una tabla con el desplazamiento usado para la dirección inicial de la tabla. Por eso el registro base se llama también *registro índice*. Las memorias actuales son mucho más grandes y el modelo de programación para la asignación de los datos es más sofisticado, así que la dirección base de la tabla normalmente se pasa a un registro, puesto que no cabría en el desplazamiento, tal y como veremos.

La sección 2.4 explica que, puesto que MIPS soporta constantes negativas, no hay necesidad de resta inmediata en lenguaje MIPS.

2.4 Números con signo y sin signo

En primer lugar daremos un vistazo a como se representan los número en el computador. Los seres humanos pensamos en base 10, pero los números pueden representarse en cualquier base. Por ejemplo, 123 en base 10 = 1111011 en base 2.

Los números se almacenan en el computador como una serie de señales eléctricas, alta o baja, y por lo tanto se consideran números en base 2. (Del mismo modo que los números en base 10 se llaman números *decimales*, los números en base 2 se llaman números *binarios*.)

Dígito binario o bit: uno de los dos números en base 2, 0 ó 1, que son los componentes de la información.

Así, un dígito de un número binario es el "átomo" de la computación, puesto que toda la información se compone de **dígitos binarios** o *bits*. Este bloque de cons-

trucción fundamental puede tomar dos valores, que pueden verse desde diferentes perspectivas: alto o bajo, activo (on) o inactivo (off), verdadero o falso, 1 o 0.

En cualquier base, el valor del *i*-ésimo digito *d* es

$$d \times \text{base}^{i}$$

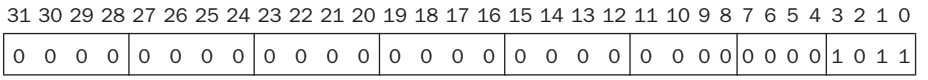
donde *i* empieza en 0 y se incrementa de derecha a izquierda. Esto conduce a una manera obvia de numerar los bits de una palabra: simplemente se usa la potencia de la base para ese bit. Indicamos los números decimales con el subíndice diez y los números binarios con el subíndice dos. Por ejemplo,

$$1011_{\text{dos}}$$

representa

$$\begin{aligned}&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{diez}}\\ &= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{diez}}\\ &= 8 + 0 + 2 + 1_{\text{diez}}\\ &= 11_{\text{diez}}\end{aligned}$$

Aquí los bits de la palabra se numeran 0, 1, 2, 3, … de *derecha* a *izquierda*. A continuación se muestra la numeración de los bits dentro de una palabra MIPS y la colocación del número 1011_{dos}:



(anchura de 32 bits)

Bit menos significativo: es el bit de más a la derecha en una palabra MIPS.

Bit más significativo: es el bit de más a la izquierda en una palabra MIPS.

Puesto que las palabras se escriben tanto verticalmente como horizontalmente, puede que el extremo izquierdo y el derecho no estén claros. Por eso, la frase **el bit menos significativo** se usa para referirse al bit situado más a la derecha (bit 0 el dibujo anterior) y **el bit más significativo** para el bit situado más a la izquierda (bit 31).

La palabra MIPS tiene una longitud de 32 bits, de manera que podemos representar 2^{32} patrones de 32 bits diferentes. Es natural que estas combinaciones representen los números del 0 a $2^{32} - 1$ ($4\,294\,967\,295_{\text{diez}}$):

$$\begin{aligned}&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{dos}} = 0_{\text{diez}}\\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{dos}} = 1_{\text{diez}}\\ &0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{dos}} = 2_{\text{diez}}\\ &\ldots \qquad\qquad\qquad \ldots\\ &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{dos}} = 4\ 294\ 967\ 293_{\text{diez}}\\ &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{dos}} = 4\ 294\ 967\ 294_{\text{diez}}\\ &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{dos}} = 4\ 294\ 967\ 295_{\text{diez}}\end{aligned}$$

Es decir, los números binarios de 32 bits se pueden representar en términos del valor del bit multiplicado por la potencia de 2 (donde *xi* significa el *i*-ésimo bit de *x*)

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Tenga en cuenta que los patrones de bits binarios anteriores son simples *representaciones* de números. Los números, en realidad, tienen un número infinito de cifras, donde casi todas son 0 excepto unas pocas situadas más a la derecha. Simplemente, no mostramos los 0s delanteros.

El hardware se puede diseñar para sumar, restar, multiplicar y dividir estos patrones de bits binarios. Si el número que es el resultado correcto de estas operaciones no se puede representar por esos bits del hardware situados más a la derecha, se dice que se ha producido *desbordamiento*. Es responsabilidad del sistema operativo y del programa determinar qué hacer si se produce un desbordamiento.

Los programas de computador operan tanto con números positivos como con negativos, de manera que necesitamos una representación que distinga el positivo del negativo. La solución más obvia es añadir un signo por separado, que puede ser convenientemente representado por un único bit; el nombre de esta representación es *signo y magnitud*.

Desgraciadamente, la representación signo y magnitud tiene diversas limitaciones. Primera, no es obvio dónde poner el bit de signo: ¿a la derecha?, ¿a la izquierda? Los primeros computadores eligieron ambas formas. Segundo, los sumadores de signo y magnitud pueden necesitar un paso extra para establecer el signo, porque no podemos saber por adelantado cuál será el signo correcto. Finalmente, un signo separado significa que la representación signo y magnitud tiene tanto un cero positivo como negativo, lo cual puede causar problemas a los programadores poco cuidadosos. Como resultado de estas limitaciones, la representación signo y magnitud fue pronto abandonada.

En la búsqueda de una alternativa más atractiva, se planteó cuál debería ser el resultado para números sin signo si intentásemos restar un número grande a otro menor. La respuesta es que se intentaría tomar prestado de la cadena de 0s iniciales, de modo que el resultado tendría una cadena de 1s delanteros.

Dado que no había una alternativa obvia mejor, la solución final fue elegir la representación que hacía el hardware más simple: los 0s del inicio significan positivo, los 1s del inicio significan negativo. Esta convención para representar los números binarios con signo se llama representación en *complemento a dos*:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{dos} = 0_{diez}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dos} = 1_{diez}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dos} = 2_{diez}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dos} = 2\ 147\ 483\ 645_{diez}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos} = 2\ 147\ 483\ 646_{diez}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{dos} = 2\ 147\ 483\ 647_{diez}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{dos} = -2\ 147\ 483\ 648_{diez}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dos} = -2\ 147\ 483\ 647_{diez}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dos} = -2\ 147\ 483\ 646_{diez}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dos} = -3_{diez}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos} = -2_{diez}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{dos} = -1_{diez}$

La mitad positiva de los números, desde 0 hasta 2 147 483 647_{diez} ($2^{31} - 1$), usa la misma representación que antes. El siguiente patrón de bits ($1000 \ldots 0000_{dos}$) representa el número más negativo –2 147 483 648_{diez} (2^{31}). Le siguen un conjunto decreciente de números negativos –2 147 483 647_{diez} ($1000 \ldots 0001_{dos}$) hasta -1_{diez} ($1111 \ldots 1111_{dos}$).

La representación en complemento a dos tiene un número negativo, –2 147 483 648_{diez}, que no tiene su correspondiente número positivo. Tal desequilibrio es una preocupación para el programador poco atento, pero el signo y magnitud tenía problemas tanto para el programador como para el diseñador del hardware. Consecuentemente, todos los computadores hoy en día usan representaciones en complemento a dos para los números con signo.

La representación en complemento a dos tiene la ventaja de que todos los números negativos tienen un 1 en el bit más significativo. Consecuentemente, el hardware sólo necesita comprobar este bit para saber si un número es positivo o negativo (el cero se considera positivo). Este bit se llama frecuentemente *bit de signo*. Reconociendo el papel de este bit de signo, podemos representar los números de 32 bits positivos y negativos en términos del valor del bit multiplicado por la potencia de 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

El bit de signo se multiplica por -2^{31}, y el resto de bits se multiplican por la versión positiva de sus valores de base respectivos.

Conversión de binario a decimal

EJEMPLO

¿Cuál es el valor decimal de este número de 32 bits en complemento a dos?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{dos}$$

RESPUESTA

Sustituyendo los valores de los bits del número en la fórmula anterior:

$$\begin{aligned}
&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0}) \\
&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0 \\
&= -2\ 147\ 483\ 648_{diez} + 2\ 147\ 483\ 644_{diez} \\
&= -4_{diez}
\end{aligned}$$

Pronto veremos un atajo para simplificar esta conversión.

Del mismo modo que una operación con números sin signo puede desbordar la capacidad del hardware para representar el resultado, así puede suceder con una operación con números en complemento a dos. El desbordamiento ocurre cuando el bit situado más a la izquierda guardado del patrón de bits binario no es el mismo que el valor de los infinitos dígitos de la izquierda (el bit de signo es incorrecto): un 0 a la izquierda del patrón de bits cuando el número es negativo o un 1 cuando el número es positivo.

Interfaz hardware software

A diferencia de los números tratados anteriormente, las direcciones de memoria empiezan de manera natural en 0 y continúan hasta la dirección más alta. Dicho de otra manera, las direcciones negativas no tienen sentido. Así, los programas a veces deben trabajar con números que pueden ser positivos o negativos y a veces con números que sólo pueden ser positivos. Algunos lenguajes de programación reflejan esta situación. El lenguaje C, por ejemplo, nombra a los primeros enteros *integer* (declarados como `int` en el programa) y a los últimos *unsigned integers*, (`unsigned int`). Algunas guías de estilo de C incluso recomiendan declarar los primeros como `signed int` (enteros con signo), para marcar la diferencia claramente.

Veamos unos cuantos atajos cuando trabajamos con números en complemento a dos. El primer atajo es una manera rápida de negar un número binario en complemento a dos. Simplemente se invierte cada 0 a 1 y cada 1 a 0 y sumando 1 al resultado. Este atajo se basa en la observación de que la suma de un número y su inverso debe ser $111\ldots111_{dos}$, lo cual representa –1. Puesto que $x + x \equiv -1$, por lo tanto $x + x + 1 = 0$ o $x + 1 = -x$.

Atajo de negación

EJEMPLO

Negar 2_{diez}, y entonces comprobar el resultado negando -2_{diez}.

RESPUESTA

$2_{diez} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dos}$

Negando este número invirtiendo los bits y añadiendo uno, sumando 1, tenemos,

$$\begin{array}{ll} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{dos} \\ + & \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{dos}} \\ = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos} \\ = & -2_{diez} \end{array}$$

Y, en sentido contrario,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos}$$

primero invertimos y luego incrementamos:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dos} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{dos} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dos} \\ = & 2_{diez} \end{array}$$

El segundo atajo nos dice cómo convertir un número binario representado con *n* bits en un número representado con más de *n* bits. Por ejemplo, el campo inmediato en las instrucciones de *carga, almacenamiento, bifurcación, suma y activar si menor que* contiene un número de 16 bits en complemento a dos, que representa desde $-32\ 768_{diez}$ (-2^{15}) hasta $32\ 767_{diez}$ ($2^{15}-1$). Para sumar el campo inmediato a un registro de 32 bits, el computador debe convertir este número de 16 bits a su equivalente de 32 bits. El atajo es tomar el bit más significativo de la cantidad menor –el bit de signo– y replicarlo para rellenar los nuevos bits de la cantidad mayor. Los bits antiguos se copian simplemente en la parte derecha de la nueva palabra. Este atajo comúnmente se llama *extensión de signo*.

Atajo de la extensión de signo

EJEMPLO

Convertir las versiones binarias de 16 bits de 2_{diez} y -2_{diez} a números binarios de 32 bits.

RESPUESTA

La versión binaria de 16 bits del número 2 es

$$0000\ 0000\ 0000\ 0010_{dos} = 2_{diez}$$

Se convierte a un número de 32 bits haciendo 16 copias del valor en el bit más significativo (0) y colocándolas en la mitad izquierda de la palabra. La mitad derecha recibe el valor anterior:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{dos} = 2_{diez}$$

Neguemos la versión de 16 bits usando el atajo anterior. Así,

$$0000\,0000\,0000\,0010_{dos}$$

se convierte en

$$\begin{array}{l} 1111\,1111\,1111\,1101_{dos} \\ +\qquad\qquad\qquad\quad 1_{dos} \\ \hline = \ 1111\,1111\,1111\,1110_{dos} \end{array}$$

Crear una versión de 32 bits del número negativo significa copiar el signo 16 veces y colocarlo a la izquierda del número:

$$1111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{dos} = -2_{diez}$$

Este truco funciona porque los números positivos en complemento a dos realmente tienen un número infinito de 0s a la izquierda y los que son negativos en complemento a dos tienen infinitos 1s. El patrón de bits binarios que representa un número oculta los bits delanteros para encajar con la anchura del hardware; la extensión de signo simplemente recupera algunos de ellos.

Resumen

El punto principal de esta sección es que necesitamos tanto enteros positivos como enteros negativos en una palabra del computador, y aunque cualquier opción tiene pros y contras, la opción inmensamente mayoritaria desde 1965 ha sido el complemento a dos.

Autoevaluación

¿Cuál es el valor decimal de este número de 64 bits en complemento a 2?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_{dos}$$

1) -4_{diez}

2) -8_{diez}

3) -16_{diez}

4) $18\,446\,744\,073\,709\,551\,609_{diez}$

Extensión: El complemento a dos toma su nombre de la regla según la cual la suma de un número de n-bits y su negativo da 2^n; por tanto, el complemento o negación de un número x en complemento a dos es $2^n - x$.

Complemento a 1: notación que representa el valor más negativo con $10...000_{dos}$ y el valor más positivo con $01...11_{dos}$, con igual número de valores positivos y negativos pero con dos ceros, uno positivo ($00...00_{dos}$) y otro negativo ($11...11_{dos}$). Este término se utiliza también para la inversión de los bits de una secuencia de bits: cambiar 0 por 1 y 1 por 0.

Notación sesgada: notación que representa el valor más negativo por $00 \ldots 000_{dos}$ y el valor más positivo por $11 \ldots 11_{dos}$, donde el 0 típicamente tiene el valor $10 \ldots 000_{dos}$, de modo que sega el número de forma que el número más el sesgo tiene una representación no negativa.

Una tercera representación alternativa es el **complemento a uno**. El negativo de un número representado en complemento a uno se halla invirtiendo cada bit, de 0 a 1 y de 1 a 0, lo cual ayuda a explicar su nombre, porque el complemento de x es $2^n - x - 1$. Fue también un intento de mejorar la solución signo magnitud, y varios computadores científicos usaron esta notación. Esta representación es similar al complemento a dos excepto en que también tiene dos ceros: $00 \ldots 00_{dos}$ es el cero positivo y $11 \ldots 11_{dos}$ es el cero negativo. El número más negativo $10 \ldots 000_{dos}$ representa $-2\ 147\ 483\ 647_{diez}$, y de esta manera los positivos y los negativos están equilibrados. Los sumadores en complemento a uno necesitan un paso extra para restar un número, y por esto el complemento a dos domina hoy en día.

Una notación final, que veremos cuando hablemos del punto flotante, es representar el valor más negativo por $00\ldots000_{dos}$ y el valor más positivo por $11\ldots11_{dos}$, donde el 0 típicamente tiene el valor $10\ldots00_{dos}$. Esta forma se llama **notación sesgada**, puesto que sesga el número de manera que el número más el sesgo tiene una representación no negativa.

Extensión: Para números decimales con signo usamos "–" para representar negativos, porque no hay límites en el tamaño de un número decimal. Dado un tamaño de palabra fijo, las cadenas de bits binarias o hexadecimales pueden incluir la codificación del signo, y por ello normalmente no usamos "+" o "–" con la notación binaria o hexadecimal.

2.5 Representación de instrucciones en el computador

Ahora ya podemos explicar la diferencia entre la manera en que los seres humanos dan instrucciones a los computadores y la manera en que los computadores ven dichas instrucciones.

Las instrucciones se guardan en el computador como series de señales electrónicas altas y bajas y se pueden representar como números. De hecho, cada parte de una instrucción se puede considerar un número individual, y juntando estos números se forma la instrucción.

Puesto que los registros son parte de casi todas las instrucciones debe haber una convención para relacionar nombres de registro con números. En el lenguaje ensamblador MIPS los registros de `$s0` a `$s7` se corresponden con los registros 16 a 23, y los registros de `$t0` a `$t7` se corresponden con los registros 8 a 15. Por tanto, `$s0` significa registro 16, `$s1` significa registro 17, `$s2` significa registro 18…, `$t0` se coloca en el registro 8, `$t1` en el registro 9, y así sucesivamente. Describiremos la convención para el resto de los 32 registros en las secciones siguientes.

Traducción de una instrucción ensamblador MIPS a una instrucción máquina

EJEMPLO

Hagamos el siguiente paso en el refinamiento del lenguaje MIPS como ejemplo. Mostraremos la versión del lenguaje MIPS real de la instrucción, representada simbólicamente como:

```
add $t0,$s1,$s2
```

primero como una combinación de números decimales y después de números binarios.

RESPUESTA

La representación decimal es:

0	17	18	8	0	32

Cada uno de estos segmentos de una instrucción se llama *campo*. El primero y el último campo (que contienen 0 y 32 en este caso) combinados indican al computador MIPS que esta instrucción realiza una suma. El segundo campo indica el número de registro que es el primer operando fuente de la operación suma (17 = `$s1`), y el tercer campo indica el otro operando fuente para la suma (18 = `$s2`). El cuarto campo contiene el número de registro que va a recibir la suma (8 = `$t0`). El quinto campo no se utiliza en esta instrucción, así que se fija en 0. Así pues, esta instrucción suma el registro `$s1` al registro `$s2` y coloca el resultado en el registro `$t0`.

Esta instrucción también puede representarse como campos de números binarios como opuestos a los decimales:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Esta disposición de la instrucción se llama el **formato de instrucción**. Como podemos ver al contar el número de bits, esta instrucción MIPS ocupa exactamente 32 bits —el mismo tamaño que una palabra de datos—. De acuerdo con nuestro principio de diseño en el que la simplicidad favorece la regularidad, todas las instrucciones MIPS tienen una longitud de 32 bits.

Formato de instrucción: forma de representación de una instrucción compuesta por campos de números binarios.

Para distinguirlo del lenguaje ensamblador, llamaremos a la versión numérica de las instrucciones **lenguaje máquina**, y a una secuencia de tales instrucciones *código máquina*.

Lenguaje máquina: representación binaria de las instrucciones usada para la comunicación dentro de un sistema informático.

Puede parecer que ahora debiéramos estar leyendo y escribiendo largas y tediosas cadenas de números binarios. Evitamos ese tedio usando una base más alta que la binaria que se convierta fácilmente en código binario. Dado que casi todos los tamaños de los datos del computador son múltiplos de 4, los **números hexadecimales** (en base 16) son idóneos. Puesto que la base 16 es una potencia de 2, podemos hacer la conversión fácilmente, sustituyendo cada grupo de cuatro dígitos binarios por un único dígito hexadecimal y viceversa. La figura 2.4 convierte de hexadecimal a binario y viceversa.

Hexadecimal: números en base 16.

Hexadecimal	Binario	Hexadecimal	Binario	Hexadecimal	Binario	Hexadecimal	Binario
0_{hex}	0000_{dos}	4_{hex}	0100_{dos}	8_{hex}	1000_{dos}	c_{hex}	1100_{dos}
1_{hex}	0001_{dos}	5_{hex}	0101_{dos}	9_{hex}	1001_{dos}	d_{hex}	1101_{dos}
2_{hex}	0010_{dos}	6_{hex}	0110_{dos}	a_{hex}	1010_{dos}	e_{hex}	1110_{dos}
3_{hex}	0011_{dos}	7_{hex}	0111_{dos}	b_{hex}	1011_{dos}	f_{hex}	1111_{dos}

FIGURA 2.4 Tabla de conversión hexadecimal-binario. Simplemente reemplazamos un dígito hexadecimal por los cuatro dígitos binarios correspondientes, y viceversa. Si la longitud del número binario no es múltiplo de 4 va de derecha a izquierda.

Debido a que con frecuencia tratamos con diversas bases de números, para evitar confusiones utilizaremos el subíndice *diez* con los números decimales, *dos* con los números binarios y *hex* con los números hexadecimales. (Si no hay subíndice, son base 10 por defecto). C y Java utilizan la notación 0x*nnnn* para los números hexadecimales.

EJEMPLO

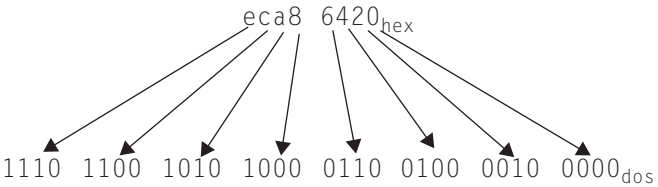
Binario a hexadecimal y viceversa

Convertir los números hexadecimales y binarios siguientes a la otra base:

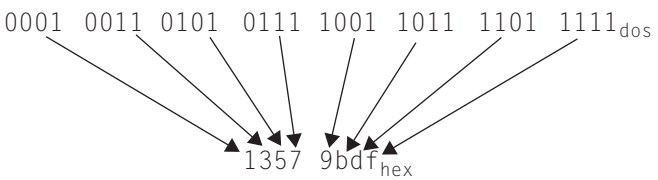
`eca8 6420`$_{hex}$

`0001 0011 0101 0111 1001 1011 1101 1111dos`

Utilizando la figura 2.4, la respuesta es una búsqueda en una tabla en una dirección:

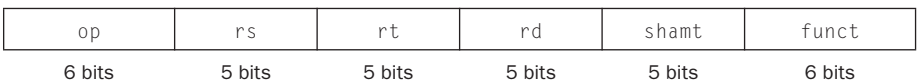


Y en la otra dirección también:



Campos MIPS

A los campos MIPS se les da nombre para identificarlos más fácilmente:



Este es el significado de cada uno de los nombres de los campos de las instrucciones MIPS:

- *op:* operación básica de la instrucción, tradicionalmente llamada código de operación u **opcode**.
- *rs:* el registro del primer operando fuente.
- *rt:* el registro del segundo operando fuente.
- *rd:* el registro del operando destino, donde se almacena el resultado de la operación.
- *shamt:* cantidad de desplazamientos (*shift amount)*. (La sección 2.6 explica instrucciones de desplazamiento y este término, que no será utilizado hasta entonces y por tanto el campo contiene cero).
- *funct:* función. Selecciona la variante específica de la operación en el campo *op* y a veces es llamado *código de función*.

Opcode: campo que indica la operación y el formato de una instrucción.

Puede haber un problema cuando una instrucción necesita campos más largos que los mostrados arriba. Por ejemplo, la instrucción cargar palabra (*load word*) debe especificar dos registros y una constante. Si la dirección tuviera que utilizar uno de los campos de 5 bits del formato anterior, la constante de la instrucción cargar palabra (*load word*) estaría limitada a, como máximo, 2^5 ó 32 bytes. Esta constante se utiliza para seleccionar elementos de las tablas o estructuras de datos, y a menudo se necesita que sea mucho mayor de 32. Este campo de 5 bits es demasiado pequeño para ser útil.

Por tanto, tenemos un conflicto entre el deseo de mantener todas las instrucciones con la misma longitud y el deseo de tener un único formato de instrucción. Esto nos conduce al último principio del diseño de hardware:

Principio de diseño 4: el buen diseño exige buenos compromisos.

El compromiso elegido por los diseñadores de MIPS es mantener todas las instrucciones con la misma longitud, que requieren diferentes clases de formatos de instrucción para diferentes clases de instrucciones. Por ejemplo, el formato anterior es llamado *R-type* (de registro) o *R-format*. Un segundo tipo de formato de instrucción se llama *I-type* (de inmediato) o *I-format*, y es utilizado por las instrucciones de transferencia de datos y las instrucciones con direccionamiento inmediato. Los campos del I-format son:

op	rs	rt	constante o dirección
6 bits	5 bits	5 bits	16 bits

Los 16 bits de la dirección significan que una instrucción cargar palabra (*load word*) puede cargar cualquier palabra de hasta $\pm 2^{15}$ ó 32 768 bytes ($\pm 2^{13}$ o 8192 palabras) de la dirección del registro base rs. De la misma forma, la suma inmediata (*add immediate*) está limitada a las constantes no más grandes que $\pm 2^{15}$. Vemos que más de 32 registros serían difíciles en este formato, ya que los campos rs y rt necesitarían cada uno otro bit, complicando más hacer caber todo en una palabra.

Miremos la instrucción cargar palabra (*load word*) de la página 83:

```
lw  $t0,32($s3)   # reg temporal $t0 toma el valor de A[8]
```

En este caso, en el campo rs se coloca 19 (de `$s3`), en el campo rt se coloca 8 (de `$t0`) y en el campo de la dirección se coloca 32. Obsérvese que el significado del campo rt ha cambiado para esta instrucción: en una instrucción cargar palabra (*load word*) el campo rt especifica el registro *destino*, que recibe el resultado de la carga.

Aunque tener múltiples formatos complica el hardware podemos reducir la complejidad manteniendo los formatos similares. Por ejemplo, los tres primeros campos de los formatos R-type e I-type son del mismo tamaño y tienen los mismos nombres; el cuarto campo del I-type es igual a la longitud de los tres últimos campos del R-type.

Por si se lo está preguntando, los formatos se distinguen por los valores del primer campo: a cada formato se le asigna un conjunto distinto de valores en el primer campo (op), de modo que el hardware sepa si debe tratar la última mitad de la instrucción como tres campos (R-type) o como un solo campo (I-type). La figura 2.5 muestra los números usados en cada campo para las instrucciones MIPS vistas a lo largo de esta sección.

Instrucción	Formato	op	rs	rt	rd	shamt	funct	dirección
`add`	R	0	reg	reg	reg	0	32_{diez}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{diez}	n.a.
`add immediate`	I	8_{diez}	reg	reg	n.a.	n.a.	n.a.	constante
`lw` (load word)	I	35_{diez}	reg	reg	n.a.	n.a.	n.a.	dirección
`sw` (store word)	I	43_{diez}	reg	reg	n.a.	n.a.	n.a.	dirección

FIGURA 2.5 Codificación de instrucciones en MIPS. En la tabla, "reg" significa un número de registro entre 0 y 31, "dirección" significa una dirección de 16 bits, y "n.a." (no aplicable) significa que ese campo no aparece en este formato. Observe que las instrucciones add y sub tienen el mismo valor en el campo op; el hardware usa el campo funct para decidir la variante de una operación: add (32) o sub (34).

Traducción de lenguaje ensamblador MIPS a lenguaje máquina

EJEMPLO

Ahora podemos mostrar un ejemplo hasta el final, de lo que escribe el programador a lo que ejecuta el computador. Si `$t1` tiene la base de la tabla `A` y `$s2` se corresponde con `h`, la sentencia de asignación:

```
A[300] = h + A[300];
```

se compila en:

```
lw  $t0,1200($t1)     # reg temporal t0 toma el valor A[300]
add $t0,$s2,$t0       # reg temporal t0 toma el valor h + A[300]
sw  $t0,1200($t1)     # h + A[300] se almacena de nuevo en A[300]
```

¿Cuál es el código de lenguaje máquina MIPS para estas tres instrucciones?

RESPUESTA

Por conveniencia, representaremos primero las instrucciones de lenguaje máquina usando números decimales. De la figura 2.5 podemos determinar las tres instrucciones de lenguaje máquina:

op	rs	rt	rd	dirección/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

La instrucción `lw` se identifica con 35 (véase figura 2.5) en el primer campo (op). El registro base 9 (`$t1`) se especifica en el segundo campo (rs), y el registro de destino 8 (`$t0`) se especifica en el tercer campo (rt). El desplazamiento para seleccionar `A[300]` (1200 = 300 × 4) se encuentra en el campo final (dirección).

La instrucción `add` que va a continuación se especifica con 0 en el primer campo (op) y 32 en el último campo (funct). Los tres operandos en registro (18, 8 y 8) se encuentran en el segundo, tercer y cuarto campos, y corresponden a `$s2`, `$t0` y `$t0`.

La instrucción `sw` se identifica con 43 en el primer campo. El resto de esta última instrucción es idéntico a la instrucción *lw*.

El equivalente binario a la forma decimal es el siguiente (1200 en base 10 es 0000 0100 1011 0000 en base 2):

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Obsérvese la semejanza de las representaciones binarias de la primera y de la última instrucción. La única diferencia está en el tercer bit desde la izquierda.

La figura 2.6 resume las partes del lenguaje ensamblador MIPS descritas en esta sección. Tal y como veremos en el capítulo 4, la semejanza de las representaciones binarias de instrucciones relacionadas simplifica el diseño del hardware. Estas instrucciones son otro ejemplo de la regularidad en la arquitectura MIPS.

Lenguaje máquina MIPS

Nombre	Formato	Ejemplo						Comentarios
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Tamaño de cada campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas las instrucciones MIPS tienen 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Formato de instrucciones aritméticas
I-format	I	op	rs	rt	address			Formato de instrucciones de transferencia de datos

FIGURA 2.6 Arquitectura MIPS mostrada en la sección 2.5. Los dos formatos de las instrucciones MIPS vistos hasta ahora son R e I. Los primeros 16 bits son los mismos: ambos contienen un campo *op* que indica la operación base, un campo *rs* que da uno de los operandos fuentes, y un campo *rt* que especifica el otro operando fuente, a excepción de la instrucción de cargar palabra (*load word*), donde se especifica el registro destino. El R-format divide los últimos 16 bits en un campo *rd*, *que indica* el registro destino, el campo *shamt* (cantidad de desplazamientos), que se explica en la sección 2.5, y el campo *funct,* que especifica la operación de las instrucciones del R-format. El I-format guarda los últimos 16 bits como un único campo de dirección.

IDEA clave

Actualmente, los computadores se construyen basados en dos principios clave:

1. Las instrucciones se representan como números.
2. Los programas son almacenados en memoria para ser leídos o escritos también como números.

Estos principios conducen al concepto de *programa almacenado* (*stored-program*). Su invención abrió la caja de Pandora de la computación. La figura 2.7 muestra el potencial de este concepto; concretamente, la memoria puede contener el código fuente para un programa editor, el correspondiente código máquina compilado, el texto que el programa compilado está utilizando e incluso el compilador que generó el código máquina.

Una consecuencia de tratar las instrucciones como números es que a menudo los programas son enviados como ficheros de números binarios. La implicación comercial que ello supone es que los computadores pueden heredar software previamente preparado si son compatibles con un repertorio de instrucciones existente. Tal "compatibilidad binaria" con frecuencia lleva a la industria a centrarse alrededor de una pequeña cantidad de arquitecturas del repertorio de instrucción.

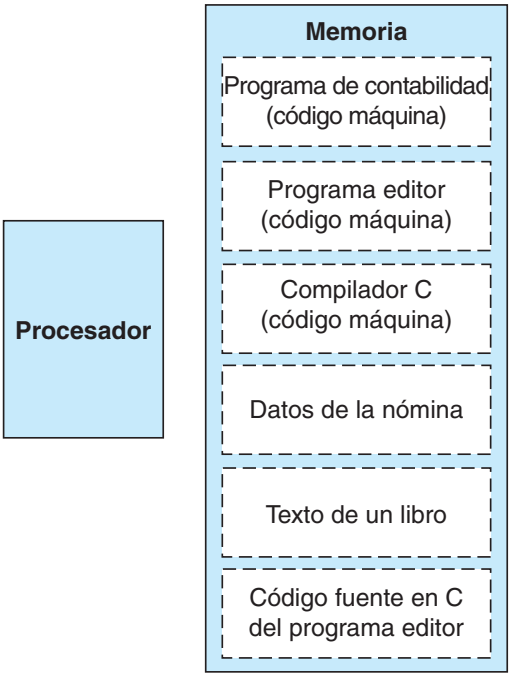


FIGURA 2.7 El concepto de programa almacenado. Los programas almacenados permiten a un computador que hace contabilidad convertirse, en un abrir y cerrar de ojos, en un computador que ayuda a un autor a escribir un libro. El cambio se lleva a cabo simplemente cargando la memoria con programas y datos y después indicando al computador que comience a ejecutar desde una posición de memoria concreta. Tratar las instrucciones de la misma forma que los datos simplifica el hardware de la memoria y el software de los sistemas informáticos. Concretamente, la tecnología de memoria que se necesita para los datos se puede utilizar también para los programas, y programas como los compiladores, por ejemplo, pueden traducir el código escrito en una notación lejana pero más conveniente para los seres humanos al código que el computador puede comprender.

Autoevaluación

¿Qué instrucción MIPS se está representando? Escoja una de la cuatro opciones.

Op	Rs	Rt	Rd	Shamt	Funct
0	8	9	10	0	34

1. `add $s0, $s1, $s2`
2. `add $s2, $s0, $s1`
3. `add $s2, $s1, $s0`
4. `sub $s2, $s0, $s1`

"¡Por el contrario!" —continuó Tweedle-dee— "si fue así, podría serlo; y si así fuera, entonces lo sería; pero como no lo es, entonces no es. ¡Es lógico!"

Lewis Carroll, *Alicia a través del espejo*, 1871.

2.6 Operaciones lógicas

Aunque los primeros computadores se centraron en palabras completas, pronto se vio claro que era útil operar con campos de bits dentro de una palabra o incluso con bits individuales. Examinar los caracteres de una palabra, cada uno de los cuales se almacena como 8 bits, es un ejemplo de este tipo de operación (véase sección 2.9). Entonces se añadieron instrucciones para simplificar, entre otras cosas, las operaciones de empaquetar y desempaquetar los bits en palabras. Estas instrucciones se llaman operaciones lógicas. La figura 2.8 muestra operaciones lógicas en C, en Java y en MIPS.

Operaciones lógicas	Operadores C	Operadores Java	Instrucciones MIPS
Shift left (desplazamiento a la izquierda)	<<	<<	`sll`
Shift right (desplazamiento a la derecha)	>>	>>>	`srl`
AND (bit-a-bit)	&	&	`and, andi`
OR (bit-a-bit)	|	|	`or, ori`
NOT (bit-a-bit)	~	~	`nor`

FIGURA 2.8 Operadores lógicos de C y de Java y sus instrucciones MIPS correspondientes. MIPS implementa la operación NOT con NOR con uno de sus operandos igual a cero.

La primera clase de este tipo de operaciones se llama *desplazamiento* (*shift*). Mueve todos los bits de una palabra a la izquierda o a la derecha, rellenando con ceros los bits vacíos. Por ejemplo, si el registro `$s0` contenía

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{dos} = 9_{diez}$$

y se ejecutó la instrucción de desplazar 4 posiciones a la izquierda, el nuevo valor se parecería a esto:

$$0000\ \ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{dos} = 144_{diez}$$

El complementario de un desplazamiento a la izquierda es un desplazamiento a la derecha. El nombre real de las dos instrucciones de desplazamiento MIPS es *desplazamiento lógico a la izquierda* (*shift left logical,* `sll`) y *desplazamiento lógico a*

la derecha (*shift right logical*, `srl`). La siguiente instrucción realiza la operación anterior, suponiendo que el resultado debería ir al registro `$t2`:

```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

Hemos retrasado la explicación del campo *shamt* en el formato de R-type hasta ahora. Éste representa la cantidad de desplazamiento (*shift amount*) y se utiliza en instrucciones de desplazamiento. Por tanto, la versión en lenguaje máquina de la instrucción anterior es:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

La codificación de `sll` es 0 en los campos op y funct, rd contiene 10 (registro `$t2`), rt contiene 16 (registro `$s0`) y shamt contiene 4. El campo rs no se utiliza, y se fija en 0.

El desplazamiento lógico a la izquierda proporciona una ventaja adicional. El desplazamiento a la izquierda de i bits nos da el mismo resultado que multiplicar por 2^i, igual que desplazar un número decimal i dígitos es equivalente a multiplicar por 10^i. Por ejemplo, el anterior `sll` de 4 arroja el mismo resultado que multiplicar por 2^4 ó 16. La primera configuración de bits anterior representa 9, y el valor de la segunda configuración de bits es $9 \times 16 = 144$.

Otra operación útil que aísla campos es **AND** (*Y*). (Pondremos en inglés la palabra para evitar la confusión entre la operación y la conjunción.) AND es una operación bit-a-bit que deja un 1 en el resultado solamente si ambos bits de los operandos son 1. Por ejemplo, si el registro `$t2` contiene

AND: operación lógica bit-a-bit con dos operandos que genera un 1 solamente si los *dos* operandos son 1.

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{dos}$$

y el registro `$t1` contiene

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{dos}$$

entonces, después de ejecutar la instrucción MIPS

```
and $t0,$t1,$t2     # reg $t0 = reg $t1 & reg $t2
```

el valor del registros `$t0` sería

$$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{dos}$$

Como podemos ver, AND puede aplicar una configuración de bits a un conjunto de bits para forzar ceros allí donde hay un 0 en la configuración de bits. Tradicionalmente, a dicha configuración de bits en conjunción con AND se le llama una *máscara*, puesto que la máscara "oculta" algunos bits.

OR: Operación lógica bit-a-bit con dos operandos que genera un 1 si hay un 1 en *cualquiera* de los dos operandos.

Para poner un valor a uno de estos mares de ceros usamos la operación complementaria de AND, llamada **OR** (*O*). Es una operación bit-a-bit que coloca un 1 en el resultado si cualquier bit del operando es un 1. Es decir, si no cambiamos los registros `$t1` y `$t2` del ejemplo anterior, el resultado de la instrucción MIPS

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

deja el siguiente valor en el registro `$t0`:

$$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000_{dos}$$

NOT: operación lógica bit-a-bit con un operando que invierte los bits; es decir, que sustituye cada 1 por un 0 y cada 0 por un 1.

NOR: operación lógica bit-a-bit con dos operandos que calcula el NOT (*x* OR *y*) de los dos operandos.

La última operación lógica es un complementario. **NOT** (*NO*) toma un operando y coloca un 1 en el resultado si el bit del operando es un 0, y viceversa. En consonancia con el formato de los dos operandos, los diseñadores de MIPS decidieron incluir la instrucción **NOR** (NOT OR) en vez de NOT. Si un operando es cero, entonces es equivalente a NOT. Por ejemplo, A NOR 0 = NOT (A OR 0) = NOT (A).

Si el registro `$t1` del ejemplo anterior no cambia y el registro `$t3` tiene valor 0, el resultado de la instrucción MIPS

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

deja el siguiente valor en el registro `$t0`:

$$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{dos}$$

La figura 2.8 anterior muestra la relación entre los operadores de C y de Java y las instrucciones MIPS. Las constantes son útiles tanto en las operaciones lógicas AND y OR como en operaciones aritméticas, así que MIPS también proporciona las instrucciones *and inmediato* (`andi`) y *or inmediato* (`ori`). Las constantes son raramente usadas para NOR, puesto que su uso principal es invertir los bits de un de único operando; así pues, el hardware no tiene ninguna versión con el direccionamiento inmediato.

Extensión: El repertorio de instrucciones de MIPS incluye también la operación "o exclusiva" (XOR), que pone el bit del resultado a 1 si los correspondientes bits de los operandos son diferentes, y a 0 si son iguales. El lenguaje C permite definir *campos de bits* o *campos* dentro de las palabras, tanto para permitir que varios objetos se empaqueten en una palabra como para ajustarse a una interfaz impuesta externa-

mente, como un dispositivo de E/S. Todos los campos deben caber dentro de una palabra, son enteros sin signo e incluso pueden tener solamente 1 bit. El compilador de C inserta y extrae los campos mediante instrucciones lógicas de MIPS: `and`, `or`, `sll` y `srl`.

¿Qué operaciones pueden aislar un campo en una palabra?

1. AND
2. Un desplazamiento a la izquierda seguido de un desplazamiento a la derecha

Autoevaluación

La utilidad de un computador automático radica en la posibilidad de usar repetidamente una secuencia de instrucciones dada; el número de veces que se itera puede ser dependiente de los resultados del cómputo. Cuando la repetición se completa, debe seguirse una secuencia diferente de [instrucciones], por lo tanto debemos, en la mayoría de los casos, dar dos series paralelas de [instrucciones] precedidas por una instrucción que indica qué rutina debe seguirse. Esta opción puede realizarse en función del signo de un número (el cero se considera positivo para este propósito). Consecuentemente, introducimos una [instrucción] (la [instrucción] de transferencia condicional) que, dependiendo del signo de un número dado, determina que se ejecute la rutina adecuada de dos posibles.
Burks, Goldstine y von Neumann, 1947.

2.7 Instrucciones para la toma de decisiones

Lo que distingue un computador de una calculadora simple es su capacidad de tomar decisiones. Basándose en los datos de entrada y en los valores creados durante el cómputo, se ejecutan diferentes instrucciones. La toma de decisiones se representa comúnmente en los lenguajes de programación utilizando la sentencia *if* (si condicional), combinado a veces con sentencias *go to* (ir a) y etiquetas. El lenguaje ensamblador MIPS incluye dos instrucciones de toma de decisiones, similares a las sentencias *if* y *go to*. La primera instrucción es:

```
beq registro1, registro2, L1
```

Esta instrucción significa ir a la sentencia etiquetada con `L1` si el valor en `registro1` es igual al valor en `registro2`. El mnemónico **beq** significa *salta si es igual* (*branch if equal*). La segunda instrucción es:

```
bne registro1, registro2, L1
```

Esta instrucción significa ir a la sentencia etiquetada con `L1` si el valor en `registro1` no es igual al valor en `registro2`. El mnemónico *bne* significa *salta si no es igual* (*branch if not equal*). Tradicionalmente, estas dos instrucciones son llamadas **bifurcaciones** o **saltos condicionales**.

Salto condicional (o bifurcación condicional): instrucción que requiere la comparación de dos valores y que permite una posterior transferencia del control a una nueva dirección en el programa, basada en el resultado de la comparación.

Compilación de una sentencia *if-then-else* (si-entonces-si no) en un salto condicional

EJEMPLO

En el segmento de código siguiente, `f`, `g`, `h`, `i` y `j` son variables. Si las cinco variables de `f` a `j` corresponden a cinco registros de `$s0` a `$s4`, ¿cuál es el código MIPS compilado para esta sentencia *if* en C?

```
if (i == j) f = g + h; else f = g - h;
```

RESPUESTA

La figura 2.9 es un diagrama de flujo de lo que debería hacer el código MIPS. La primera expresión compara la igualdad, por tanto parece que usaríamos `beq`. En general, el código será más eficiente si comprobamos la condición opuesta para saltar sobre el código que realiza la parte *then* (entonces) que sigue al *if* (si); la etiqueta `Else` (si no) está definida a continuación:

```
bne $s3,$s4,Else   # ir a Else si i ≠ j
```

La siguiente sentencia de asignación realiza una sola operación y, si todos los operandos están asignados a registros, es simplemente una instrucción:

```
add $s0,$s1,$s2    # f = g + h (se omite si i ≠j)
```

Ahora necesitamos ir al final de la sentencia *if*. Este ejemplo introduce otra clase de bifurcación o salto, frecuentemente llamado *salto incondicional*. Esta instrucción dice que el procesador sigue siempre el salto. Para distinguir entre los saltos condicionales e incondicionales, el nombre MIPS para este tipo de instrucción es *jump* (salto), abreviado como `j` –la etiqueta `Salida` (Exit) está definida a continuación–.

```
j Exit     # ir a Salida
```

La sentencia de asignación en la parte del *else* (si no) de la sentencia *if* (si) puede traducirse o compilarse de nuevo en una sola instrucción. Ahora necesitamos añadir la etiqueta `Si no` (*Else*) a esta instrucción. También mostramos la etiqueta `Salida` que está después de esta instrucción, y vemos el final del código compilado *if-then-else* (si-entonces-si no):

```
Else:sub $s0,$s1,$s2   # f = g - h (se omite si i = j)
Exit:
```

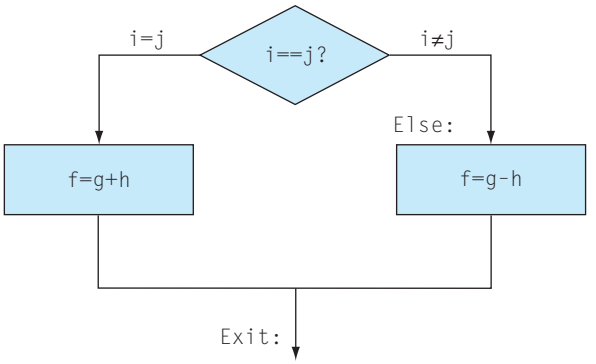


FIGURA 2.9 Ilustración de las opciones de la sentencia *if* anterior. La caja de la izquierda corresponde a la parte *then* (entonces) de la sentencia *if*, y la caja de la derecha corresponde a la parte *else* (si no).

Observe que el ensamblador libera al compilador y al programador del lenguaje ensamblador del tedio de calcular las direcciones para las bifurcaciones, de forma similar a como lo hace para calcular las direcciones de los datos para cargas y almacenamientos (véase la sección 2.12).

Interfaz hardware software

Los compiladores crean con frecuencia bifurcaciones y etiquetas cuando no aparecen en el lenguaje de programación. Evitar el agobio de la escritura de las etiquetas y bifurcaciones explícitas es uno de los beneficios de escribir en lenguajes de programación de alto nivel, y es una razón por la cual la codificación es más rápida a ese nivel.

Lazos

Las decisiones son importantes tanto para elegir entre dos alternativas —las que aparecen en las sentencias *if* (si)— como para iterar un cómputo —que aparece en los lazos—. Las mismas instrucciones ensamblador son las unidades básicas en ambos casos.

EJEMPLO

Compilación de un lazo *while* (mientras) en C

Disponemos de un lazo tradicional en C

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

Suponga que *i* y *k* corresponden a los registros `$s3` y `$s5` y la base de la tabla `guarda` está en `$s6`. ¿Cuál es el código ensamblador MIPS correspondiente a este segmento de C?

RESPUESTA

El primer paso es cargar `guardar[i]` en un registro temporal. Antes de que podamos hacerlo necesitamos tener su dirección, y antes de que podamos sumar *i* a la base de la tabla `guardar` para formar la dirección debemos multiplicar el índice *i* por 4 debido al problema del direccionamiento de byte. Afortunadamente, podemos utilizar el desplazamiento lógico a la izquierda, puesto que desplazando a la izquierda 2 bits multiplicamos por 4 (véase la página 103 en la sección 2.5). Necesitamos añadir la etiqueta *Lazo* (Loop) de modo que podamos saltar de nuevo a esta instrucción al final del mismo:

```
Loop:sll  $t1,$s3,2     # reg temporal $t1 = 4 * i
```

Para conseguir la dirección de `guardar[i]` necesitamos sumar `$t1` y la base de `guardar` en `$s6`:

```
      add $t1,$t1,$s6 # $t1 = dirección de guardar[i]
```

Ahora podemos utilizar esa dirección para cargar `guardar[i]` en un registro temporal:

```
      lw  $t0,0($t1) # reg temporal $t0 = guardar[i]
```

La instrucción siguiente realiza el test del lazo, y va a Salida (Exit) si `guardar[i]` ≠ `k`:

```
      bne  $t0,$s5, Exit # ir a Salida si guardar[i] ≠ k
```

La instrucción siguiente suma `1` a `i`:

```
      add  $s3,$s3,1 # i = i + 1
```

El final del lazo salta de nuevo hacia atrás hasta el test del `while` (mientras) en la parte superior del lazo. Únicamente añadimos la etiqueta `Salida` justo detrás, de la forma siguiente:

```
      j    Loop          # ir a Lazo (Loop)
Exit:
```

(Véanse los ejercicios para una optimización de esta secuencia).

Interfaz hardware software

Estas secuencias de instrucciones que terminan con una bifurcación son tan fundamentales para la compilación que tienen su propio nombre: un **bloque básico** es una secuencia de instrucciones sin bifurcaciones, excepto posiblemente al final, y sin ningún destino de saltos o etiquetas de saltos, excepto posiblemente al comienzo. Una de las primeras fases de la compilación es dividir el programa en bloques básicos

Bloque básico: secuencia de instrucciones sin bifurcaciones (excepto posiblemente al final) y sin destinos de saltos o etiquetas de saltos (excepto posiblemente al comienzo).

La prueba de la igualdad o la desigualdad es probablemente el test más habitual, pero algunas veces es útil ver si el valor de una variable es menor que el de otra variable. Por ejemplo, para un lazo *for* (para) se puede desear comprobar si la variable índice es menor que 0. Este tipo de comparaciones se realizan en el lenguaje ensamblador MIPS con una instrucción que compara dos registros y asigna a un tercer regis-

tro un 1 si el primero es menor que el segundo; si no es así, le asigna un 0. La instrucción MIPS se llama *set on less than* (activar si es menor que) o *slt*. Por ejemplo

```
slt   $t0, $s3, $s4
```

significa que el registro `$t0` se actualiza a `1` si el valor en el registro `$s3` es menor que el valor en el registro `$s4`; si no, el registro `$t0` se actualiza a `0`.

Las constantes como operandos son frecuentes en las comparaciones. Puesto que el registro `$zero` siempre contiene 0, podemos comparar directamente con 0. Para comparar con otros valores hay una versión de la instrucción set on less than (activar si menor que) con valor inmediato. Para comprobar si el registro `$s2` es menor que la constante 10 podemos escribir:

```
slti   $t0,$s2,10   # $t0 = 1 if $s2 < 10
```

Interfaz hardware software

Los compiladores del MIPS utilizan `slt`, `slti`, `beq`, `bne` y el valor fijo 0 (siempre disponible leyendo el registro `$zero`) para crear todas las condiciones relativas: igual, no igual, menor que, menor o igual que, mayor que, mayor o igual que.

Teniendo en cuenta la advertencia de Von Neumann sobre la simplicidad del "equipo", la arquitectura MIPS no incluye la instrucción de bifurcación si es menor que, porque es demasiado complicada: o bien alargaría la duración de ciclo de reloj o la instrucción necesitaría ciclos de reloj extras. Dos instrucciones más rápidas son más útiles.

Interfaz hardware software

Las instrucciones de comparación deben abordar la dicotomía entre números con y sin signo. En ocasiones, una secuencia de bits con un 1 en el bit más significativo representa un número negativo y, por supuesto, es menor que cualquier número positivo, que debe tener su bit más significativo igual a 0. Por otra parte, si los números son sin signo, un número con su bit más significativo igual a 1 es *mayor* que cualquier otro número que comience con 0. (Más adelante tendremos en cuenta este doble significado del bit más significativo para reducir el coste de la comprobación de los límites de un vector.)

MIPS ofrece dos versiones de la comparación *poner a 1 cuando menor que* (*set on less than*) para contemplar estas dos alternativas. *Set on less than* (`slt`) y *set on less than inmediate* (`slti`) operan con números con signo. Las comparaciones de números sin signo se realizan con *set on less than unsigned* (`sltu`) y *set on less than inmediate unsigned* (`sltiu`).

Comparaciones con signo frente a comparaciones sin signo

EJEMPLO

Supongamos que el contenido del registro `$s0` es el número binario

`1111 1111 1111 1111 1111 1111 1111 1111`$_{dos}$

y el del registro `$s1` es el número binario

`0000 0000 0000 0000 0000 0000 0000 0001`$_{dos}$

¿Qué valores tendremos en los registros `$t0` y `$t1` después de las dos siguientes instrucciones?

```
slt  $t0, $s0, $s1     # comparación con signo
sltu $t1, $s0, $s1     # comparación sin signo
```

RESPUESTA

El valor en el registro `$s0` es -1_{diez} si se está representando un entero y 4 294 967 295_{diez} si es un entero sin signo. El valor en el registro `$s1` representa el valor 1_{diez} en ambos casos. Entonces, en `$t0` tendremos el valor 1, porque $-1_{diez} < 1_{diez}$, y en `$t1` el valor 0 porque 4 294 967 $295_{diez} > 1_{diez}$.

Considerar los números con signo como si fuesen números sin signo nos proporciona una manera de comprobar si $0 \leq x < y$ de bajo coste, que coincide con el chequeo de un índice fuera-de-límites de los vectores. La clave es que los enteros negativos en representación binaria de complemento a 2 son similares a número muy altos en una notación sin signo; esto es, el bit más significativo es el bit de signo en el primer caso y una gran parte del número en el segundo caso. Así, la comparación sin signo $x < y$ también comprueba si x es negativo al mismo tiempo que si x es menor que y.

Atajo para la comprobación de límites

EJEMPLO

Utilice este atajo para simplificar la comprobación de si un índice está fuera de los límites: salto a `IndexOutOfBounds` si `$s1` ≥ `$t2` o si `$s1` es negativo.

RESPUESTA

El código de comprobación utiliza `sltu` para ambos tests:

```
sltu $t0, $s1, $t2 # $t0 = 0 si $s1 >= longitud o $s1<0
beq  $t0, $zero, IndexOutOfBounds # si malo, salto a Error
```

La sentencia case / switch

La mayoría de los lenguajes de programación tienen una sentencia alternativa *case* o *switch,* que permite que el programador seleccione una de muchas alternativas dependiendo de un único valor. La manera más simple de implementar *switch* (cambio) es a través de una secuencia de pruebas condicionales, convirtiendo la sentencia *switch* en una cadena de sentencias *if-then-else* (si-entonces-si no).

A veces, las alternativas se pueden codificar eficientemente como una tabla con las direcciones de las secuencias de instrucciones alternativas, llamada **tabla de direcciones de salto**, en la que el programa necesita solamente indexar en la tabla y después saltar a la secuencia apropiada. Así, la tabla de salto es simplemente una tabla de palabras que contiene direcciones que se corresponden con las etiquetas en el código. El programa carga la entrada apropiada de la tabla de salto en un registro. A partir de este momento, para saltar necesita la dirección en el registro. Para permitir este tipo de situaciones, los computadores como MIPS incluyen una instrucción *jump register* (registro de salto) o `jr`, que significa un salto incondicional a la dirección especificada en un registro. El programa carga la entrada apropiada de la tabla de saltos en un registro, y luego salta a la dirección adecuada utilizando un registro de salto. Esta instrucción está descrita en la sección siguiente.

Tabla de direcciones de salto *(jump address table)*: también llamada tabla de salto. Es una tabla de direcciones de las secuencias de instrucciones alternativas.

Interfaz hardware software

Aunque hay muchas sentencias para decisiones y lazos en los lenguajes de programación como C y Java, la sentencia básica que las pone en ejecución en el nivel inferior siguiente es la bifurcación condicional.

Extensión: Si ha oído hablar de los *saltos retardados*, tratados en el capítulo 4, no se preocupe: el ensamblador MIPS los hace invisibles al programador del lenguaje ensamblador.

Autoevaluación

I. C tiene muchas sentencias para decisiones y lazos, mientras que MIPS tiene pocas. ¿Cuál de las afirmaciones siguientes explica o no este desequilibrio? ¿Por qué?

1. La mayoría de las sentencias de decisión hacen que el código sea más fácil de leer y de comprender.
2. Menos sentencias de decisión simplifican la tarea del nivel inferior que es responsable de la ejecución.
3. Más sentencias de decisión significan menos líneas de código, lo que reduce generalmente el tiempo de codificación.
4. Más sentencias de decisión significan pocas líneas de código, lo que da lugar generalmente a la ejecución de pocas operaciones.

II. ¿Por qué C proporciona dos grupos de operadores para AND (& y &&) y dos grupos de operadores para OR (| y ||), mientras que las instrucciones MIPS no lo hacen?

1. Las operaciones lógicas AND y OR usan & y |, mientras que las bifurcaciones condicionales usan && y || en la ejecución.
2. La afirmación anterior tiene su inversa: && y || corresponden a las operaciones lógicas, mientras que & y | corresponden a las bifurcaciones condicionales.
3. Son redundantes y significan la misma cosa: && y || son simplemente herederos del lenguaje de programación B, el precursor de C.

2.8 Apoyo a los procedimientos en el hardware del computador

Procedimiento: subrutina almacenada que realiza una tarea específica basada en los parámetros que le son proporcionados.

Un **procedimiento** o una función (o subrutina) es una herramienta que usan los programadores de C o de Java para estructurar programas, a fin de hacerlos más fáciles de entender y de permitir que el código sea reutilizado. Los procedimientos permiten al programador concentrarse en una sola parte de la tarea en un momento determinado, usando los parámetros como barrera entre el procedimiento y el resto del programa y los datos, lo que permite que pasen los valores y que devuelva resultados. Describimos el equivalente en Java en la **sección 2.15** en el CD, pero Java necesita todo lo que C necesita de un computador.

Podemos pensar en un procedimiento como un espía que funciona con un plan secreto, que adquiere recursos, realiza la tarea, borra sus pistas y después retorna al punto original con el resultado deseado. Ninguna otra cosa debería verse perturbada una vez que la misión se ha completado. Por otra parte, un espía opera solamente con una "necesidad de saber" básica, así que no puede hacer suposiciones sobre su jefe.

Análogamente, en la ejecución de un procedimiento el programa debe seguir estos seis pasos:

1. Colocar los parámetros en un lugar donde el procedimiento pueda acceder a ellos.
2. Transferir el control al procedimiento.
3. Adquirir los recursos del almacenamiento necesarios para el procedimiento.
4. Realizar la tarea deseada.
5. Colocar el valor del resultado en un lugar donde el programa que lo ha llamado pueda tener acceso.
6. Retornar el control al punto del origen, puesto que un procedimiento puede ser llamado desde varios puntos de un programa.

Como ya hemos dicho, los registros son el lugar más rápido para situar datos en un computador; por tanto, procuramos utilizarlos tanto como sea posible. Los programas MIPS siguen la convención siguiente en la asignación de sus 32 registros para llamar a un procedimiento:

- `$a0-$a3`: cuatro registros de argumentos para pasar parámetros.
- `$v0-$v1`: dos registros de valores para retornar valores.
- `$ra`: un registro con la dirección de retorno para volver al punto del origen.

Además de asignar estos registros, el lenguaje ensamblador MIPS incluye una instrucción sólo para los procedimientos: salta a una dirección y simultáneamente guarda la dirección de la instrucción siguiente en el registro `$ra`. La instrucción de **saltar-y-enlazar** o `jal` (*jump-and-link*) se escribe simplemente:

Instrucción saltar-y-enlazar: instrucción que salta a una dirección y simultáneamente guarda la dirección de la instrucción siguiente en un registro (`$ra` en MIPS).

```
jal  ProcedureAddress
```

La parte de *enlace* (*link*) en el nombre de la instrucción significa que se construye una dirección o enlace que apunta al punto desde el que se hace la llamada para permitir que el procedimiento retorne a la dirección apropiada. Este enlace almacenado en el registro `$ra` se llama **dirección de retorno**. Esta dirección de retorno o remitente es necesaria porque el mismo procedimiento se podría llamar desde varias partes del programa.

Dirección de retorno: enlace al sitio que llama que permite que un procedimiento retorne a la dirección apropiada; en MIPS se almacena en el registro `$ra`.

Para trabajar en tales situaciones, los computadores como MIPS utilizan la instrucción *salto según registro* o `jr` (*jump register*), que realiza un salto incondicional a la dirección especificada en un registro:

```
jr   $ra
```

Los saltos de la instrucción salto según registro (*jump register, jr*) saltan a la dirección almacenada en el registro `$ra` —lo cual es justo lo que deseamos—. Así, el programa que llama o invoca, es decir, el **invocador** o **llamador** (*caller*), pone los valores de los parámetros en `$a0-$a3` y utiliza `jal X` para saltar al procedimiento `X` —a veces conocido por **invocado** o **llamado** (*callee*)—. El procedimiento llamado realiza entonces los cálculos, pone los resultados en `$v0-$v1` y devuelve el control al llamador utilizando el `jr $ra`.

Invocador (o llamador): programa que convoca un procedimiento y proporciona los valores de los parámetros necesarios.

Invocado (o llamado): procedimiento que ejecuta una serie de instrucciones almacenadas basadas en los parámetros proporcionados por el llamador y después le devuelve el control.

Implícita en la idea de programa almacenado está la necesidad de tener un registro para guardar la dirección de la instrucción actual que está siendo ejecutada. Por razones históricas, este registro casi siempre se llama **contador de programas**, abreviado como *PC* (*program counter*) en la arquitectura MIPS, aunque un nombre más acorde habría sido registro de dirección de instrucción. La instrucción `jal` guarda PC + 4 en el registro `$ra` para enlazar con la instrucción siguiente y establecer el retorno del procedimiento.

Contador de programas (PC): registro que contiene la dirección de la instrucción que está siendo ejecutada en el programa.

Utilización de más registros

Supongamos que un compilador necesita más registros para un procedimiento que los cuatro registros de argumentos y los dos registros de retorno de valores. Puesto que debemos borrar nuestras pistas después de completar nuestra misión, cualquier registro usado por el llamador debe ser restaurado con los valores que contenía *antes* de que el procedimiento fuera invocado. Esta situación es un ejemplo en el cual necesitamos volcar los registros a la memoria, tal y como se mencionó en la sección *Interfaz hardware software*.

Pila: estructura de datos para los registros volcados organizados como una cola tipo "último en entrar-primero en salir".

Puntero de pila: valor que indica la dirección más recientemente puesta en la pila y que muestra dónde deberían ser volcados los registros o dónde se pueden encontrar los valores antiguos de los registros.

Apilar: añadir un elemento a una pila.

Desapilar: sacar un elemento de una pila.

La estructura de datos ideal para volcar registros es una **pila** (*stack*), una cola tipo "el último en entrar-el primero en salir". Una pila necesita un puntero hacia la dirección más recientemente usada en la pila para mostrar al procedimiento siguiente donde debería colocar los registros que se volcarán, o donde se encuentran los valores antiguos de los registros. El **puntero de pila** (*stack pointer*) se ajusta a una palabra cada vez que se guarda o restaura un registro. MIPS reserva el registro 29 para el puntero de pila, dándole el nombre obvio de `$sp` (viene de *stack pointer*). Las pilas son tan utilizadas que las tranferencias de datos a ellas y desde ellas tienen sus propios nombres: poner datos en la pila se llama **apilar** (*push*), y sacar datos de la pila se llama **desapilar** (*pop*).

Por razones históricas, la pila "crece" desde las direcciones más altas a las más bajas. Esta convención significa que ponemos valores sobre la pila restando del puntero de pila. Sumar al puntero de pila reduce el apilado, quitando de ese modo valores de la pila.

EJEMPLO

Compilación de un procedimiento C que no llama a otro procedimiento

Convirtamos el ejemplo de la página 79 de la sección 2.2 en un procedimiento C:

```
int ejemplo_hoja (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

¿Cuál es el código ensamblador MIPS compilado?

RESPUESTA

Los parámetros variables `g`, `h`, `i` y `j` se corresponden con los registros de argumento `$a0`, `$a1`, `$a2` y `$a3`, y `f` se corresponde con `$s0`. El programa compilado comienza con la etiqueta del procedimiento:

```
ejemplo_hoja:
```

El paso siguiente es guardar los registros usados por el procedimiento. La sentencia de asignación de C en el cuerpo del procedimiento es idéntica al ejemplo de la página 79, que utiliza dos registros temporales. De aquí que necesitemos guardar tres registros: `$s0`, `$t0` y `$t1`. "Apilamos" los valores antiguos sobre la pila creando el espacio para tres palabras en la pila y después las almacenamos:

```
addi $sp,$sp,-12  # ajusta la pila para hacer sitio para
                  #  3 campos
sw $t1, 8($sp)    # guardar el registro $t1 para usarlo
                  # posteriormente
sw $t0, 4($sp)    # guardar el registro $t0 para usarlo
                  # posteriormente
sw $s0, 0($sp)    # guardar el registro $s0 para usarlo
                  # posteriormente
```

La figura 2.10 muestra la pila antes, durante y después de la llamada al procedimiento.

Las tres sentencias siguientes corresponden al cuerpo del procedimiento, que sigue el ejemplo de la página 79:

```
add $t0,$a0,$a1 # registro $t0 contiene g + h
add $t1,$a2,$a3 # registro $t1 contiene i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, que es (g + h)-(i + j)
```

Para retornar el valor de `f`, lo copiamos en un registro de retorno del valor:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Antes del retorno, restauramos los tres valores antiguos de los registros que guardamos "desapilándolos" de la pila:

```
lw $s0, 0($sp)  # restaura registros $s0 para el llamador
lw $t0, 4($sp)  # restaura registros $t0 para el llamador
lw $t1, 8($sp)  # restaura registros $t1 para el llamador
addi $sp,$sp,12 # ajusta la pila para eliminar 3 campos
```

El procedimiento termina con una instrucción de salto indirecto según registro (*jump register*) que utiliza la dirección de retorno:

```
jr $ra     # salto de retorno a la rutina de llamada
```

En el ejemplo anterior utilizamos los registros temporales y suponemos que sus valores antiguos deben ser guardados y restaurados. Para evitar guardar y restaurar un registro cuyo valor nunca se utiliza, cosa que puede suceder con un registro temporal, el software MIPS separa 18 de los registros en dos grupos:

- `$t0-$t9`: 10 registros temporales que *no* son preservados por el procedimiento llamado en una llamada al procedimiento.
- `$s0-$s7`: 8 registros guardados que se deben preservar en una llamada a procedimiento (si fueran utilizados, el procedimiento llamado los guardaría y restauraría).

Esta simple convención reduce el volcado de registros. En el ejemplo anterior, puesto que el llamador (procedimiento que hace la llamada) no espera que los registros `$t0`

y $t1 sean preservados a lo largo de una llamada al procedimiento, podemos evitar dos almacenamientos y dos cargas en el código. Todavía debemos guardar y restaurar $s0, ya que el llamado debe suponer que el llamador necesita su valor.

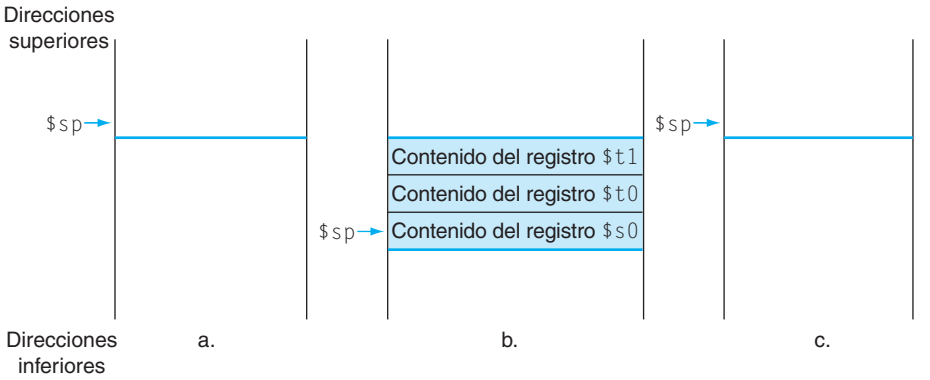


FIGURA 2.10 Los valores del puntero de pila y la pila (a) antes, (b) durante y (c) después de la llamada al procedimiento. El puntero de pila siempre apunta a la "cabeza" de la pila, o a la última palabra de la pila en este dibujo.

Procedimientos anidados

Los procedimientos que no llaman a otros se llaman los *procedimientos hoja* (*leaf*). La vida sería sencilla si todos los procedimientos fueran procedimientos hoja, pero no lo son. De la misma forma que un espía puede emplear a otros espías para cumplir parte de una misión, que a su vez pueden utilizar a otros espías, así los procedimientos invocan a otros procedimientos. Aún más, los procedimientos recurrentes o recursivos incluso invocan a "clones" de sí mismos. Del mismo modo que necesitamos tener cuidado cuando se usan registros en los procedimientos, aún más cuidado debemos tener cuando se invocan procedimientos no-hoja.

Por ejemplo, supongamos que el programa principal llama al procedimiento A con un argumento de 3, pone el valor 3 en el registro $a0 y después usa jal A. Supongamos entonces que el procedimiento A llama al procedimiento B haciendo jal B con un argumento de 7, también colocado en $a0. Puesto que A todavía no tiene terminada su tarea hay un conflicto con el uso del registro $a0. Análogamente, existe un conflicto con la dirección de retorno en el registro $ra, puesto que ahora tiene la dirección de retorno para el B. A menos que tomemos medidas para prevenir el problema, este conflicto eliminará la capacidad del procedimiento A para volver a su llamador.

Una solución es poner en la pila todos los otros registros que se deban preservar, tal y como hicimos con los registros guardados. El llamador pone en la pila cualquier registro de argumentos de la sentencia ($a0-$a3) o registro temporal ($t0-$t9) que le sea necesario después de la llamada. El llamado pone en la pila el registro de dirección de retorno $ra y cualquier registro guardado ($s0- $s7) usado por el llamado. El puntero de pila $sp se ajusta para contar el número de registros puestos en la pila. Antes del retorno, los registros se restauran desde la memoria y se reajusta el puntero de pila.

Compilación de un procedimiento recursivo C, mostrando el encadenamiento de procedimientos anidados

EJEMPLO

Veamos el procedimiento recursivo que calcula el factorial:

```
int fact (int n)
{
     if (n < 1) return (1);
           else return (n * fact(n-1));
}
```

¿Cuál es el código ensamblador MIPS?

RESPUESTA

El parámetro `n` variable se corresponde con el registro de argumentos `$a0`. El programa compilado comienza con la etiqueta del procedimiento y después guarda dos registros en la pila, la dirección de retorno y `$a0`:

```
fact:
     addi   $sp,$sp,-8  # ajustar la pila para 2 elementos
     sw     $ra, 4($sp) # guardar la dirección de retorno
     sw     $a0, 0($sp) # guardar el argumento n
```

La primera vez que se llama al procedimiento `fact` la instrucción `sw` guarda una dirección del programa que ha llamado a `fact`. Las dos instrucciones siguientes comprueban si `n` es menor que 1, y se va a `L1` si $n \geq 1$.

```
     slti $t0,$a0,1     # test para n < 1
     beq $t0,$zero,L1   # si n >= 1, ir a L1
```

Si n es menor que 1, `fact` devuelve 1 y coloca 1 en un registro de valor: se suma 1 con 0 y se sitúa el resultado de la suma en `$v0`. Después se eliminan dos valores guardados de la pila y salta a la dirección de retorno:

```
     addi $v0,$zero,1  # devolver 1
     addi $sp,$sp,8    # eliminar 2 elementos de la pila
     jr $ra            # retornar al llamador
```

Antes de quitar los dos elementos de la pila, habríamos podido cargar `$a0` y `$ra`. Puesto que `$a0` y `$ra` no cambian cuando `n` es menor que 1, nos saltamos esas instrucciones.

Si `n` no es menor que 1, se disminuye el argumento `n` y entonces `fact` se llama otra vez con el valor disminuido:

```
L1: addi $a0,$a0,-1 # n >= 1: el argument toma el valor (n - 1)
    jal fact        # llamar a fact con (n - 1)
```

La siguiente instrucción es donde `fact` retorna. La dirección de retorno antigua y el argumento antiguo se restauran, junto con el puntero de pila:

```
lw   $a0, 0($sp) # retorno de jal:restaura el argumento n
lw   $ra, 4($sp) # restaura la dirección de retorno
addi $sp, $sp,8  # ajusta el puntero de pila para eliminar
                 # 2 elementos
```

A continuación, el registro de valor `$v0` toma el valor del producto del argumento antiguo `$a0` y del valor actual del registro de valor. Suponemos que se dispone de una instrucción de multiplicar, aunque no será tratada hasta el capítulo 3:

```
mul   $v0,$a0,$v0  # devuelve n * fact (n - 1)
```

Finalmente, `fact` salta de nuevo a la dirección de retorno:

```
jr    $ra          # retorna al llamador
```

Interfaz hardware software

Una variable C es una posición de almacenamiento, y su interpretación depende de su *tipo* y *clase de almacenamiento*. Los ejemplos incluyen números enteros y caracteres (véase sección 2.9). C tiene dos clases de almacenamiento: *automático* y *estático*. Las variables automáticas son locales para un procedimiento y se descartan cuando el procedimiento finaliza. Las variables estáticas existen para todos los procedimientos, desde que comienzan hasta que acaban. Las variables de C declaradas fuera de todos los procedimientos se consideran estáticas, al igual que cualquier variable declarada que usa la palabra clave `static`. El resto es automático. Para simplificar el acceso a los datos estáticos el software MIPS reserva otro registro, llamado el **puntero global** o `$gp`.

Puntero global: registro que se reserva para apuntar a datos estáticos.

La figura 2.11 resume qué se preserva a lo largo de una llamada a un procedimiento. Obsérvese que varios sistemas conservan la pila garantizando que el llamador obtendrá, al cargar desde la pila, los mismos datos de retorno que puso en la pila mediante un almacenamiento. La pila por encima de `$sp` se protege simplemente cerciorándose que el llamado no escribe sobre `$sp`; `$sp` es a su vez preservado por el llamado añadiéndole exactamente la misma cantidad de elementos que se le restaron, y los otros registros son conservados salvándolos en la pila (si se utilizan) y restaurándolos desde allí.

Preservado	No preservado
Registros guardados: `$s0`–`$s7`	Registros temporales: `$t0`–`$t9`
Registro puntero de la pila: `$sp`	Registros de argumentos: `$a0`–`$a3`
Registro dirección de retorno: `$ra`	Registros de valor de retorno: `$v0`–`$v1`
Pila por encima del puntero de pila	Pila por debajo del puntero de pila

FIGURA 2.11 Qué es y qué no es preservado a lo largo de una llamada a un procedimiento. Si el programa cuenta con un registro de puntero de bloque de activación o de registro de puntero global, tratados en las siguientes secciones, también se preservan.

Asignación del espacio para los nuevos datos en la pila

La última complicación es que la pila también se utiliza para almacenar las variables locales del procedimiento que no caben en los registros, tales como tablas o estructuras locales. El segmento de la pila que contiene los registros guardados del procedimiento y las variables locales se llama **estructura del procedimiento** o **bloque de activación**. La figura 2.12 muestra el estado de la pila antes, durante y después de la llamada a un procedimiento.

Estructura del procedimiento *(bloque de activación)*: segmento de la pila que contiene los registros guardados y las variables locales de un procedimiento.

Algunos programas MIPS utilizan un **puntero de estructura** (*frame pointer*) llamado `$fp` para señalar a la primera palabra de la estructura de un procedimiento. Un puntero de pila podría cambiar durante el procedimiento, y por tanto las referencias a una variable local en memoria podrían tener diversos desplazamientos dependiendo de donde estuvieran en el procedimiento, haciendo que éste sea más difícil de entender. Como alternativa, un puntero de estructura ofrece un registro base estable dentro de un procedimiento para las referencias locales de la memoria. Obsérvese que un bloque de activación aparece en la pila se haya utilizado o no un puntero de estructura explícito. Hemos estado evitando `$fp` al evitar cambios en `$sp` dentro de un procedimiento: en nuestros ejemplos, la pila se ajusta solamente a la entrada y la salida del procedimiento.

Puntero de estructura *(frame pointer)*: valor que indica la posición de los registros guardados y de las variables locales para un procedimiento dado.

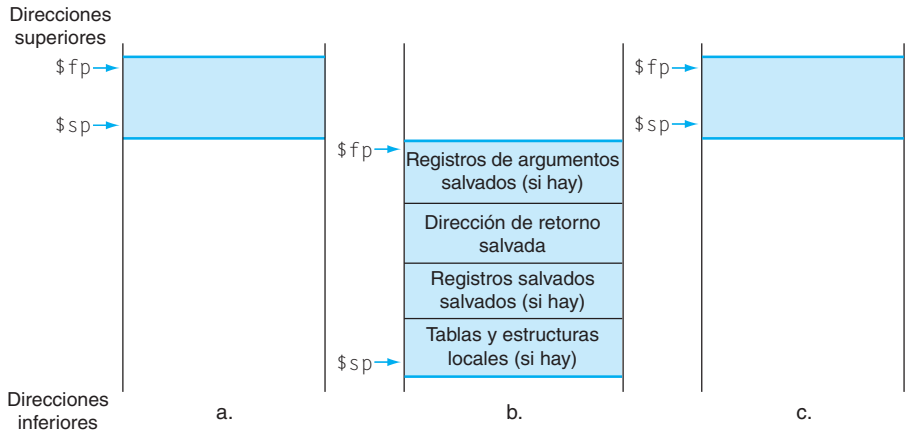


FIGURA 2.12 Illustración de la asignación de la pila (a) antes, (b) durante y (c) después de la llamada al procedimiento. El puntero de estructura (*frame pointer*) o `$fp` señala hacia la primera palabra del bloque de activación, habitualmente un registro de argumento guardado, y el puntero de pila o `$sp` apunta al comienzo de la pila. La pila se ajusta para hacer sitio a todos los registros guardados y a toda variable local residente en la memoria principal. Puesto que el puntero de pila puede cambiar durante la ejecución del programa, es más fácil que los programadores se refieran a variables vía el puntero de estructura, aunque podría hacerse sólo con el puntero de pila y una pequeña aritmética de direcciones. Si no hay variables locales en la pila dentro de un procedimiento, el compilador ahorrará tiempo si no ajusta y no restaura el puntero de estructura. Cuando se utiliza un puntero de estructura se inicializa usando la dirección en `$sp` de la llamada, y se restaura `$sp` usando `$fp`. Esta información se encuentra también en la columna 4 de la Tarjeta de Datos de Referencia de MIPS, que se incluye con este libro.

Asignación del espacio para los nuevos datos sobre el montón (*heap*)

Además de las variables automáticas que son locales para los procedimientos, los programadores de lenguaje C necesitan espacio en la memoria para las variables estáticas y para las estructuras de datos dinámicas. La figura 2.13 muestra el convenio de MIPS para la asignación de la memoria. La pila comienza en el límite superior de la memoria y crece hacia abajo. La primera parte del extremo inferior de la memoria está reservada, y va seguida por el código máquina original de MIPS, tradicionalmente llamado **segmento de texto**. Encima del código está el *segmento de datos estáticos*, donde se almacenan las constantes y otras variables estáticas. Aunque las tablas tienden a ser de longitud fija y entonces son una buena combinación para el segmento de datos estáticos, las estructuras de datos como listas encadenadas tienden a crecer y a contraerse durante su ciclo de vida. Tradicionalmente, el segmento para tales estructuras de datos se llama el *montón* (*heap)* y se coloca después en memoria. Obsérvese que esta asignación permite a la pila y al *montón* crecer el uno hacia el otro, posibilitando el uso eficiente de la memoria mientras los dos segmentos incrementan y disminuyen.

Segmento de texto: segmento de un fichero objeto de Unix que contiene el código en lenguaje máquina para las rutinas en el archivo fuente.

C asigna y libera el espacio en el montón con funciones explícitas: `malloc( )` asigna espacio en el montón y retorna un puntero a él, y `free( )` libera espacio en el montón al cual apunta el puntero. Los programas en C controlan la asignación de memoria C y es la fuente de muchos errores (*bugs*) comunes y difíciles. Olvidarse de liberar espacio conduce a una "pérdida de memoria", que puede llegar a "colgar" (*crash*) el sistema operativo. Liberar espacio demasiado pronto puede conducir a "punteros colgados" (*dangling pointers*), y provoca que algunos punteros apunten a donde el programa no tiene intención de apuntar. Para evitar estos errores, Java utiliza una asignación de manera automática y "garbage collection".

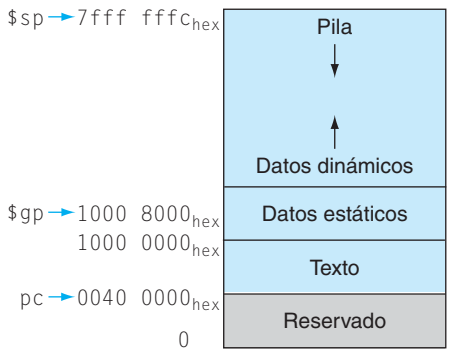


FIGURA 2.13 Asignación de memoria MIPS para el programa y los datos. Estas direcciones son solamente una convención del software y no parte de la arquitectura MIPS. Comenzando de arriba a abajo, el puntero de pila se inicializa en $7fff\ fffc_{hex}$ y crece hacia abajo, hacia el segmento de datos. En el otro extremo, el código del programa ("texto") empieza en $0040\ 0000_{hex}$. Los datos estáticos empiezan en $1000\ 0000_{hex}$. Los datos dinámicos, asignados por `malloc` en C y por `new` en Java, van a continuación y crecen hacia arriba, hacia la pila, en un área llamada el montón (*heap*). El puntero global, `$gp`, se fija a una dirección para hacer fácil el acceso a los datos. Se inicializa en `1000 8000hex`, de modo que pueda tener acceso a partir de $1000\ 0000_{hex}$ hasta $1000\ ffff_{hex}$ usando desplazamientos de 16 bits positivos y negativos desde `$gp`. Esta información se encuentra también en la columna 4 de la Tarjeta de Datos de Referencia de MIPS, que se incluye con este libro.

La figura 2.14 resume las convenciones de los registros para el lenguaje ensamblador MIPS.

Nombre	Número de registros	Uso	Preservado en llamada
$zero	0	El valor constante 0	n.a.
$v0–$v1	2–3	Valores para los resultados y evaluación de la expresión	no
$a0–$a3	4–7	Argumentos	no
$t0–$t7	8–15	Temporales	no
$s0–$s7	16–23	Guardados	sí
$t8–$t9	24–25	Más temporales	no
$gp	28	Puntero global	sí
$sp	29	Puntero de la pila	sí
$fp	30	Puntero de estructura	sí
$ra	31	Dirección de retorno	sí

FIGURA 2.14 Convenciones de registro MIPS. El registro 1, llamado $at, está reservado para el ensamblador (véase la sección 2.12), y los registros 26-27, llamados $k0-$k1, están reservados para el sistema operativo. Esta información se encuentra también en la columna 2 de la Tarjeta de Datos de Referencia de MIPS que se incluye con este libro.

Extensión: ¿Qué pasa si hay más de cuatro parámetros? La convención MIPS es poner los parámetros adicionales en la pila justo por encima del puntero de estructura. El procedimiento espera entonces que los cuatro primeros parámetros estén en los registros $a0 a $a3 y el resto en la memoria, direccionables a través del puntero de estructura (puntero del bloque de activación).

Según lo mencionado en el pie de la figura 2.12, el puntero de estructura es conveniente porque todas las referencias a las variables en la pila dentro de un procedimiento tendrán el mismo desplazamiento. Sin embargo, el puntero de estructura no es necesario. El compilador de C del GNU para MIPS utiliza un puntero de estructura, pero el compilador de C de MIPS no lo hace; utiliza el registro 30 como otro registro guardado ($s8).

Extensión: Algunos procedimientos recursivos pueden implementarse iterativamente sin usar una recursión. De este modo se pueden mejorar las prestaciones de forma significativa porque se elimina el sobrecoste asociado a la llamada al procedimiento. Por ejemplo, consideremos un procedimiento que acumula una suma:

```
int sum(in n, int acc){
  if (n>0)
     return sum(n-1,acc+n);
  else
     return acc;
}
```

Si la llamada es sum(3,0), recursivamente se llama a sum(2,3), sum(1,5) y sum(0,6), y el resultado es 6, que será devuelto cuatro veces. Esta llamada recursiva a suma se conoce como llamada de *cola* (*tail call*), y este ejemplo puede implementarse eficientemente (suponga que $a0 = n y $a1 = acc):

```
sum: slti$a0, 1              # comprueba si n <= 0
     beq$a0, $zero, sum_exit # salto a sum_exit si n <= 0
     add$a1, $a1, $a0        # suma n a acc
```

```
    addi$a0, $a0, -1      # decrementa n
    j sum                 # salto a sum
sum_exit:
    add$v0, $a1, $zero    # devuelve el valor acc
    jr $ra                # retorno de rutina
```

Autoevaluación ¿Cuáles de las siguientes declaraciones sobre C y Java son generalmente ciertas?

1. Los programadores de C manipulan datos explícitamente, mientras que en Java es automático.
2. C es más propenso a errores de punteros y pérdidas de memoria que Java.

!(@|=> (la lengüeta abierta de la ululación en la barra es grande) Cuarta línea del teclado- poema "Hatless Atlas," 1991 (algunos dan nombres a los caracteres ASCII: "!" es la ululación, "(" está abierto, "|" es la barra, y así sucesivamente).

2.9 Comunicarse con la gente

Los computadores fueron inventados para hacer muchos cálculos, pero tan pronto como llegaron a ser comercialmente viables se utilizaron para procesar texto. La mayoría de los computadores utilizan hoy bytes de 8 bits para representar caracteres, con el código *American Standard Code for Information Interchange* (ASCII), que es la representación que casi todos siguen. La figura 2.15 resume el ASCII.

Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter
32	espacio	48	0	64	@	80	P	096	`	112	p
33	!	49	1	65	A	81	Q	097	a	113	q
34	"	50	2	66	B	82	R	098	b	114	r
35	#	51	3	67	C	83	S	099	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	SUPR

FIGURA 2.15 Representación ASCII de los caracteres. Obsérvese que las letras mayúsculas y minúsculas difieren exactamente en 32; esta observación nos puede ayudar a hacer simplificaciones a la hora de comprobar o cambiar mayúsculas y minúsculas. Los valores no mostrados corresponden a caracteres de formato. Por ejemplo, 8 representa la tecla de retroceso, 9 representa un carácter de tabulación y 13 un retorno del carro. Otro valor útil es 0 para nulo (falta de información), que es el valor que el lenguaje C de programación utiliza para marcar el final de una cadena. Esta información se encuentra también en la columna 3 de la Tarjeta de Datos de Referencia de MIPS que se incluye con este libro.

Interfaz hardware software

La base 2 no es natural para el ser humano; tenemos diez dedos y por lo tanto encontramos la base 10 natural. ¿Por qué los computadores no usan una representación decimal? De hecho, el primer computador comercial ofrecía una aritmética decimal. El problema era que este computador todavía usaba señales "on" y "off", de modo que un dígito decimal se representaba por varios dígitos binarios. La representación decimal se mostró tan ineficiente que los computadores siguientes representaban todo en binario, convirtiendo a base 10 sólo las operaciones de entrada/salida, relativamente infrecuentes.

ASCII frente a números binarios

EJEMPLO

Se podrían representar los números como secuencias de dígitos ASCII en lugar de como enteros. ¿Cuánto aumentaría el almacenamiento si el número mil millones se representa como ASCII en lugar de como un entero de 32 bits?

RESPUESTA

Mil millones, 1 000 000 000, necesitaría 10 dígitos ASCII de 8 bits cada uno. De este modo, el aumento en el almacenamiento sería $(10 \times 8)/32 = 2.5$. Además, el hardware para sumar, restar, multiplicar y dividir números decimales es complejo. Estas dificultades explican por qué los profesionales de la computación creen que la representación binaria es natural y por qué consideran extraña la utilización de la representación decimal.

Una serie de instrucciones puede extraer un byte de una palabra, así que las instrucciones de cargar palabra (*load word*) y almacenar palabra (*store word*) son suficientes para la transferencia tanto de bytes como de palabras. No obstante, debido a la popularidad del procesamiento de texto en algunos programas, MIPS proporciona instrucciones para mover bytes. Cargar byte (*load byte,* `lb`) carga un byte desde la memoria y lo coloca en los 8 bits más a la derecha de un registro. Almacenar byte (*store byte,* `sb`) toma un byte de los 8 bits más a la derecha de un registro y lo escribe en la memoria. Así, copiamos un byte con la secuencia

```
lb $t0,0($sp)          # leer byte de la fuente
sb $t0,0($gp)          # escribir byte en el destino
```

Interfaz hardware software

La dicotomía con signo-sin signo se aplica también a las cargas, además de a la aritmética. La *función* de una carga con signo es copiar el signo repetitivamente hasta completar el resto del registro, llamado *extensión de signo*, pero su *propósito* es cargar una representación correcta del número en el registro. La carga sin signo rellena con ceros los bits a la izquierda del dato, de modo que el número representado por la secuencia de bits no tiene signo.

Cuando se carga una palabra de 32 bits en un registro de 32 bits, este aspecto es dudoso; las cargas con y sin signo son idénticas. MIPS ofrece dos variantes para la carga de un byte: *carga un byte* (`lb`) trata el byte como un número con signo y hace extensión de signo para completar los 24 bits más a la izquierda del registro, mientras que carga un *byte sin signo* (`lbu`) lo trata como un entero sin signo. Dado que los programas en C casi siempre usan bytes para representar caracteres en lugar de considerar los bytes como enteros con signo muy cortos, `lbu` se usa casi exclusivamente para la carga de bytes.

Los caracteres normalmente se combinan en cadenas, que tienen un número variable de ellos. Hay tres opciones para representar una cadena: (1) la primera posición de la cadena se reserva para indicar la longitud o tamaño de la cadena, (2) una variable complementaria que tiene la longitud de la cadena (como en una estructura), o (3) la última posición de una cadena se indica con un carácter de fin de cadena. C utiliza la tercera opción: termina una cadena con un byte cuyo valor es 0 —denominado nulo (*null*) en ASCII—. Así, la cadena "Cal" se representa en C con los 4 bytes siguientes, mostrados como números decimales: 67, 97, 108, 0.

EJEMPLO

Compilación de un procedimiento de copia de cadenas que muestra cómo se utilizan las cadenas en C

El procedimiento `strcpy` copia la cadena *y* en la cadena *x* usando la convención de terminación con byte nulo de C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copia y comprueba el
    byte */
    i += 1;
}
```

¿Cuál es el código ensamblador MIPS?

RESPUESTA

A continuación mostramos el fragmento de código básico de ensamblador MIPS. Suponemos que las direcciones base de las tablas *x* e *y* se encuentran en `$a0` y `$a1`, mientras que `i` estará en `$s0`. `strcpy` ajusta el puntero de pila y luego guarda el registro `$s0` en la pila:

```
strcpy:
    addi $sp,$sp,-4  # ajusta la pila para un elemento más
    sw   $s0,0($sp)  # guardar $s0
```

Para inicializar `i` a 0, la siguiente instrucción actualiza `$s0` a 0 sumando 0 a 0 y colocando esa suma en `$s0`:

```
    add    $s0,$zero,$zero   # i = 0 + 0
```

éste es el principio del lazo. La dirección de `y[i]` se forma primero sumando `i` a `y[]`:

```
L1: add    $t1,$s0,$a1       # dirección de y[i] en $t1
```

Observe que no tenemos que multiplicar `i` por 4 puesto que *y* es una tabla de bytes y no de palabras, como en ejemplos anteriores.

Para cargar el carácter en `y[i]` utilizamos cargar byte (*load byte*), que pone el carácter en `$t2`:

```
    lb     $t2, 0($t1)       # $t2 = y[i]
```

Un cálculo de dirección similar coloca la dirección de `x[i]` en `$t3`, y entonces el carácter en `$t2` se almacena en esa dirección.

```
    add    $t3,$s0,$a0  # dirección de x[i] en $t3
    sb     $t2, 0($t3)  # x[i] = y[i]
```

A continuación salimos del lazo si el carácter es 0; es decir, si es el último carácter de la cadena:

```
    beq  $t2,$zero,L2 # si y[i] == 0, ir a L2
```

si no, incrementamos `i` y se vuelve hacia atrás al lazo:

```
    addi $s0,$s0,1    # i = i + 1
    j    L1           # ir a L1
```

si no se salta hacia atrás, porque era el último carácter de la cadena, restauramos `$s0` y el puntero de pila y después se retorna.

```
L2: lw   $s0,0($sp)    # y[i] == 0: fin de cadena;
                       # restaura el antiguo $s0
    addi $sp,$sp,4     # elimina una palabra de la pila
    jr   $ra           # retorna
```

En C, el copiado de cadenas utiliza generalmente punteros en vez de tablas, para evitar las operaciones con *i* en el código anterior. Véase la sección 2.15 para una explicación comparativa de tablas *versus* punteros.

Puesto que el procedimiento `strcpy` anterior es un procedimiento hoja, el compilador podría asignar `i` a un registro temporal y evitar tener que guardar y restaurar `$s0`. Por tanto, en vez de considerar los registros `$t` simplemente como lugares temporales, podemos pensar en ellos como registros que el llamado debe utilizar siempre que crea conveniente. Cuando un compilador encuentra un procedimiento hoja, utiliza todos los registros temporales antes de usar los registros que debe guardar.

Caracteres y cadenas en Java

Unicode es una codificación universal de los alfabetos de la mayoría de los idiomas humanos. La figura 2.16 es una lista de los alfabetos de Unicode; existen tantos *alfabetos* en Unicode como *símbolos* útiles hay en ASCII. Para ser más inclusivos, Java utiliza Unicode para los caracteres. Por defecto, utiliza 16 bits para representar un carácter.

El repertorio de instrucción MIPS tiene instrucciones explícitas para cargar y para almacenar cantidades de 16 bits llamadas *medias palabras* (*halfwords)*. Cargar media (*load half,* `lh`) carga media palabra de la memoria, y la coloca en los 16 bits de más a la derecha de un registro. De forma similar a la carga (*load*), la instrucción de *carga mitad* (*load half*, `lh`) considera la media palabra como un número con signo y utiliza la extensión de signo para completar los 16 bits de la izquierda del registro; por otra parte, la instrucción *carga mitad sin signo* (*load half unsigned*, `lhu`) considera enteros sin signo. Esto hace que `lhu` sea la más popular de las dos. Almacenar media (*store half,* `sh`) toma la mitad de una palabra, los 16 bits de la parte de más a la derecha de un registro, y la escribe en la memoria. Copiamos una media palabra con la secuencia:

```
lhu $t0,0($sp) # lee media palabra (16 bits) desde una fuente
sh  $t0,0($gp) # escribe media palabra (16 bits) a un destino
```

Las cadenas son una clase estándar de Java con una ayuda especial incorporada y métodos predefinidos para el encadenamiento, la comparación y la conversión. A diferencia de C, Java incluye una palabra que da la longitud de la cadena, similar a las tablas de Java.

Extensión: Los programas MIPS intentan mantener la pila alineada a direcciones de palabras, permitiendo al programa utilizar solamente `lw` y `sw` (que deben estar alineados) para tener acceso a la pila. Esta convención significa que una variable `char` (tipo carácter) asignada a la pila ocupa 4 bytes, aunque necesite menos. Sin embargo, una variable de cadena de C o una tabla de bytes empaquetará 4 bytes por palabra, y una variable de cadena de Java o una tabla de cortos empaquetará 2 medias palabras por palabra.

Latino	Malayalam	Tagbanwa	Puntuación general
Griego	Sinhala	Khmer	Letras modificadoras de espacio
Cirílico	Tailandés	Mongol	Símbolos actuales
Armenio	Lao	Limbu	Combinar marcas diacríticas
Hebreo	Tibetano	Tai Le	Combinar las marcas para los símbolos
Árabe	Myanmar	Kangxi Radicals	Superíndices y subíndices
Sirio	Georgiano	Hiragana	Formas de número
Thaana	Hangul coreano	Katakana	Operadores matemáticos
Devanagari	Ethiopic	Bopomofo chino	Símbolos alfanuméricos matemáticos
Bengalí	Cheroqui	Kanbun	Patrones de Braille
Gurmukhi	Sílabas aborígenes canadienses	Shavian	Reconocimiento de caracteres ópticos
Guyarate	Ogham	Osmanya	Símbolos musicales Byzantine
Oriya	Runic	Cypriot Syllabary	Símbolos musicales
Tamil	Tagalog	Tai Xuan Jing Symbols	Flechas
Telugu	Hanunoo	Yijing Hexagram Symbols	Caja de dibujo
Kannada	Buhid	Aegean Numbers	Formas geométricas

FIGURA 2.16 Ejemplos de alfabetos en Unicode. La versión 4.0 de Unicode tiene más de 160 "bloques", que es el nombre que usa para una colección de símbolos. Cada bloque es un múltiplo de 16. Por ejemplo, el griego empieza en 0370_{hex}, y el cirílico en 0400_{hex}. Las primeras tres columnas muestran 48 bloques que corresponden a idiomas humanos en aproximadamente el orden numérico de Unicode. La última columna tiene 16 bloques que son multilingües y no están en orden. Una codificación de 16 bits, llamada UTF-16, es la básica por defecto. Una codificación de longitud variable, llamada UTF-8, guarda el subconjunto ASCII como 8 bits y utiliza 16-32 bits para los otros caracteres. UTF-32 utiliza 32 bits por carácter. Para aprender más, véase *www.unicode.org*.

Autoevaluación

I. ¿Cuáles de las siguientes sentencias sobre caracteres y cadenas en C y en Java son verdaderas?

1. Una cadena en C ocupa aproximadamente la mitad de memoria que la misma cadena en Java.
2. "Cadena" es precisamente un nombre informal para las tablas de una sola dimensión de caracteres en C y en Java.
3. En C y en Java se usa el carácter nulo (0) para marcar el final de una cadena.
4. Las operaciones con cadenas, así como su longitud, son más rápidas en C que en Java.

II. ¿Qué tipo de variable que puede contener el valor $1\ 000\ 000\ 000_{diez}$ necesita más espacio de memoria?

1. `int` de C
2. `string` de C
3. `string` de Java

2.10 Direcciones y direccionamiento inmediato MIPS para 32 bits

Aunque mantener todas las instrucciones MIPS con un tamaño de 32 bits simplifica todo el hardware, en ocasiones sería conveniente tener una constante de 32 bits o una dirección de 32 bits. Esta sección comienza con la solución general para las constantes grandes y después muestra las optimizaciones para las direcciones de instrucción usadas en saltos condicionales (bifurcaciones) e incondicionales.

Operandos inmediatos de 32 bits

Aunque las constantes son con frecuencia cortas y caben en campos de 16 bits, algunas veces son más grandes. El repertorio de instrucciones MIPS incluye específicamente la instrucción *carga superior inmediata* (*load upper immediate*, `lui`) para fijar los 16 bits superiores de una constante en un registro, permitiendo a una instrucción posterior especificar los 16 bits inferiores de la constante. La figura 2.17 muestra la operación de `lui`.

EJEMPLO

Carga de una constante de 32 bits

¿Cuál es el código ensamblador MIPS para cargar la siguiente constante de 32 bits en el registro `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

RESPUESTA

Primero cargaríamos los 16 bits de mayor peso, que es 61 en decimal, usando `lui`:

```
lui $s0, 61 # 61 en decimal = 0000 0000 0011 1101 binario
```

El valor del registro `$s0` después de esto es:

```
0000 0000 0011 1101 0000 0000 0000 0000
```

El paso siguiente es sumar los 16 bits de menor peso, cuyo valor decimal es 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

El valor final del registro `$s0` es el valor deseado:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

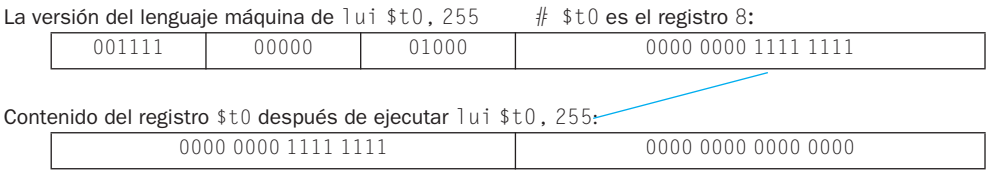


FIGURA 2.17 El efecto de la instrucción `lui`. La instrucción `lui` transfiere el valor del campo constante inmediato de 16 bits a los 16 bits del extremo izquierdo del registro, y rellena los 16 bits más bajos con ceros.

Interfaz hardware software

Tanto el compilador como el ensamblador deben descomponer las constantes grandes en partes y después volverlas a montar en un registro. Como es de esperar, la restricción del tamaño del campo inmediato puede ser un problema para las direcciones de memoria en cargas y almacenamientos, igual que para las constantes en instrucciones inmediatas. Si este trabajo recae en el ensamblador, como pasa en los programas MIPS, éste debe tener un registro temporal disponible para crear los valores largos. Ésta es la razón por la que el registro `$at` se reserva para el ensamblador.

De aquí que la representación simbólica del lenguaje máquina MIPS no esté limitada por el hardware, sino por lo que el creador del ensamblador elige incluir (véase la sección 2.12). Nos quedamos cerca del hardware para explicar la arquitectura del computador, atentos a cuándo utilizamos el lenguaje mejorado del ensamblador que no se encuentra en el procesador.

Extensión: Hay que tener cuidado en la creación de constantes de 32 bits. La instrucción `addi` copia el bit de mayor peso del campo inmediato de 16 bits de la instrucción en los 16 bits de mayor peso de una palabra. La instrucción *or lógico inmediato* (`ori`) de la sección 2.6 carga ceros en los 16 bits de mayor peso y por tanto es utilizado por el ensamblador conjuntamente con `lui` para crear constantes de 32 bits.

Direccionamiento en saltos condicionales (bifurcaciones) e incondicionales

Las instrucciones de salto MIPS tienen el direccionamiento más simple. Utilizan el formato de instrucción MIPS final, llamado de *J-type*, que consiste en 6 bits para el campo de la operación y el resto para el campo de la dirección. Así,

```
j   1000    # ir a la posición 10000
```

podría ser montado en este formato (en realidad es un poco más complicado, como veremos en la página siguiente):

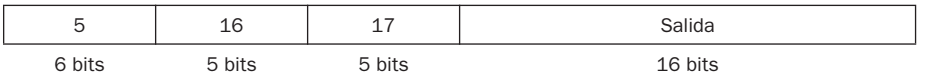
2	10000
6 bits	26 bits

donde el valor del código de operación del salto es 2 y la dirección del salto es `10000`.

A diferencia de la instrucción de salto incondicional, la instrucción de bifurcación condicional debe especificar dos operandos además de la dirección de bifurcación. Así pues,

```
bne $s0,$s1,Exit        # ir a Salida si $s0 ≠ $s1
```

se ensambla en esta instrucción, y deja solamente 16 bits para la dirección de bifurcación:



Si las direcciones del programa tuvieran que caber en este campo de 16 bits, significaría que ningún programa podría ser más grande que 2^{16}, cosa que está lejos de ser una opción realista hoy en día. Una alternativa sería especificar un registro que se añadiera siempre a la dirección de bifurcación, de modo que una instrucción de bifurcación calcularía lo siguiente:

Contador de programa = registro + dirección de bifurcación

Esta suma permite al programa alcanzar los 2^{32} bytes y aún ser capar de utilizar las bifurcaciones condicionales, solucionando el problema del tamaño de la dirección de bifurcación. La pregunta entonces es: ¿qué registro?

La respuesta viene dada al considerar cómo se utilizan las bifurcaciones condicionales. Los saltos condicionales se encuentran en lazos y en sentencias *if*, por tanto tienden a bifurcar una instrucción cercana. Por ejemplo, casi la mitad de todas las bifurcaciones condicionales en los programas de prueba SPEC van a parar a posiciones no más allá de 16 instrucciones. Puesto que el contador de programa (PC) contiene la dirección de la instrucción actual, podemos saltar dentro del rango de palabras $\pm 2^{15}$ desde la instrucción actual si utilizamos el PC como el registro a ser añadido a la dirección. Casi todos los lazos y sentencias *if* son mucho más pequeños de $\pm 2^{16}$ palabras, así que el PC es la elección ideal.

Direccionamiento relativo al PC: régimen de direccionamiento en el cual la dirección es la suma del contador del programa (PC) y de una constante en la instrucción.

Esta forma de direccionamiento de las bifurcaciones o saltos condicionales se llama **direccionamiento relativo al PC**. Tal y como veremos en el capítulo 4, es conveniente que el hardware incremente el PC pronto para apuntar a la instrucción siguiente. Por tanto, la dirección MIPS hace referencia en realidad a la dirección de la instrucción siguiente (PC + 4) en lugar de a la instrucción actual (PC).

Como la mayoría de los computadores recientes, MIPS utilizan el modo de direccionamiento relativo al PC para todas las bifurcaciones o saltos condicionales porque el destino de estas instrucciones es probable que esté cerca de la instrucción de bifurcación. Por otra parte, las instrucciones de saltar-y-enlazar (*jump-and-link*) invocan procedimientos que no tienen ninguna razón para estar cerca de la llamada, así que normalmente utilizan otros modos de direccionamiento. Por tanto, la arquitectura MIPS permite direcciones largas en las llamadas a procedimientos al utilizar el formato J-type para las instrucciones de salto y de saltar-y-enlazar.

Puesto que todas las instrucciones MIPS tienen 4 bytes de largo, MIPS alarga la distancia de las bifurcaciones teniendo el direccionamiento relativo al PC referido al número de *palabras* de la instrucción siguiente, en vez de al número de bytes. Así, el campo de 16 bits puede bifurcar cuatro veces más lejos si interpreta el campo como una dirección relativa a la palabra en lugar de como una dirección relativa al byte. Análogamente, el campo de 26 bits en instrucciones de salto es también una dirección de palabra, así que representa una dirección byte de 28 bits.

Extensión: Puesto que el PC es de 32 bits, 4 bits deben venir de alguna otra parte. La instrucción de salto MIPS reemplaza solamente los 28 bits de menor peso del PC, y deja los 4 bits de mayor peso sin cambios. El cargador y el enlazador (sección 2.12) deben tener cuidado de evitar colocar un programa que cruce una dirección límite de 256 MB (64 millones de instrucciones); de lo contrario, un salto se debe reemplazar por una instrucción de salto indirecto sobre registro precedida por otras instrucciones para cargar la dirección completa de 32 bits en un registro.

Muestra del desplazamiento de una bifurcación o salto condicional en lenguaje máquina

EJEMPLO

El lazo *while* en la página 107-108 se ha compilado en este código ensamblador MIPS

```
Loop:sll $t1,$s3,2      # registro temporal $t1 = 4 * i
     add $t1,$t1,$s6    # $t1 = dirección de guardar[i]
     lw $t0,0($t1)      # registro temporal $t0 = guardar[i]
     bne $t0,$s5,Salida # ir a Salida si guardar[i] _ k
     addi $s3,$s3,1     # i = i + 1
     j Loop             # ir a Lazo
Exit:
```

Si suponemos que colocamos el lazo comenzando en la posición 80000 en memoria, ¿cuál es el código máquina MIPS para este lazo?

RESPUESTA

Las instrucciones ensambladas y sus direcciones tienen el siguiente aspecto:

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Recuérdese que las instrucciones MIPS utilizan direcciones de byte; por tanto, las direcciones de palabras consecutivas se diferencian en 4, el número de bytes en una palabra. La instrucción *bne* en la cuarta línea añade 2 palabras o 8 bytes a la dirección de la instrucción *siguiente* (80016), especificando así el destino de la bifurcación o salto condicional en relación con la siguiente instrucción (8 + 80016), en vez de en relación con la instrucción actual de bifurcación (12 + 80012) o usando la dirección de destino completa (80024). La instrucción de salto (*jump*) en la última línea utiliza la dirección completa (20000 × 4 = 80000) correspondiente a la etiqueta `Lazo`.

Interfaz hardware software

Casi todas las bifurcaciones o saltos condicionales están en una localización próxima, pero ocasionalmente se bifurca o salta más lejos, más allá de lo que puede ser representado con los 16 bits de la instrucción de bifurcación o salto condicional. El ensamblador viene al rescate tal y como lo hizo con las constantes o direcciones grandes: inserta un salto incondicional al destino de la bifurcación e invierte la condición, de modo que la bifurcación o salto condicional decida si esquiva el salto.

EJEMPLO

Bifurcación a distancia

Dada una bifurcación condicional que se produce si el registro `$s0` es igual al registro `$s1`,

```
beq   $s0,$s1, L1
```

substituirlo por un par de instrucciones que ofrezcan una distancia de bifurcación mucho mayor.

RESPUESTA

Estas instrucciones pueden ser reemplazadas por la bifurcación o salto condicional con direccionamiento corto:

```
      bne   $s0,$s1, L2
      j     L1
L2:
```

Resumen de los modos de direccionamiento de MIPS

Modo de direccionamiento: uno de varios regímenes de direccionamiento delimitados por el uso variado de operandos y de direcciones.

Las múltiples formas de direccionamiento genéricamente se llaman **modos de direccionamiento** *(addressing modes)*. La figura 2.18 muestra cómo se identifican los operandos para cada modo de direccionamiento. Los modos de direccionamiento de MIPS son los siguientes:

1. *Direccionamiento inmediato*: cuando el operando es una constante que aparece en la misma instrucción.
2. *Direccionamiento a registro*: cuando el operando está en un registro.

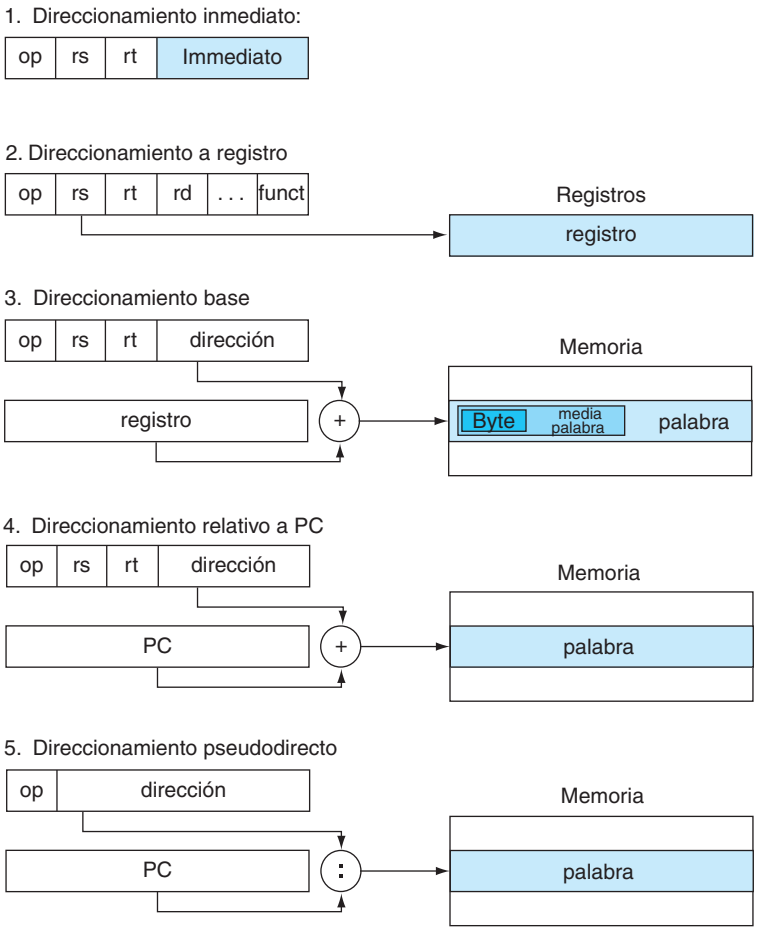


FIGURA 2.18 Ilustración de los cinco modos de direccionamiento MIPS. Los operandos aparecen resaltados. El operando del modo 3 está en memoria, mientras que el operando del modo 2 es un registro. Observe que las versiones de carga y almacenamiento acceden a bytes, medias palabras o palabras. En el modo 1 el operando está en los 16 bits de la misma instrucción. Los modos 4 y 5 se usan para direccionar instrucciones en memoria: en el modo 4 se suman los 16 bits de la dirección desplazados 2 bits a la izquierda al PC, y en el modo 5 se concatenan una dirección de 26 bits desplazados 2 bits a la izquierda con los 4 bits de mayor peso del PC.

3. *Direccionamiento base o desplazamiento*: cuando el operando está en la posición de memoria cuya dirección es la suma de un registro y de una constante en la instrucción.
4. *Direccionamiento relativo al PC*: cuando la dirección es la suma del PC y de una constante en la instrucción.
5. *Direccionamiento pseudodirecto*: cuando la dirección de salto es los 26 bits de la instrucción concatenados con los bits de mayor peso del PC.

Interfaz hardware software

Aunque mostramos la arquitectura MIPS con direcciones de 32 bits, casi todos los microprocesadores (MIPS incluido) tienen extensiones de direcciones de 64 bits (véase el apéndice E). Estas extensiones nacieron como respuesta a las necesidades de software para programas más grandes. El proceso de ampliación del repertorio de instrucciones permite crecer a las arquitecturas de manera que se mantenga la compatibilidad de los programas hasta la siguiente generación de la arquitectura.

Obsérvese que una operación simple puede utilizar más de un modo de direccionamiento. La instrucción sumar (`add`), por ejemplo, usa tanto el direccionamiento inmediato (`addi`) como el direccionamiento a registro (`add`).

Descodificación de lenguaje máquina

A veces nos vemos forzados a aplicar técnicas de ingeniería inversa al lenguaje máquina para reconstruir el lenguaje ensamblador del original. Por ejemplo, al examinar un volcado de memoria. La figura 2.19 muestra la codificación MIPS de los campos para el lenguaje máquina MIPS. Esta figura ayuda a traducir "a mano" el lenguaje ensamblador al lenguaje máquina y viceversa.

EJEMPLO

Descodificación de código máquina

RESPUESTA

¿Cuál es el lenguaje ensamblador que corresponde a esta instrucción de máquina?

```
00af8020hex
```

El primer paso es convertir de hexadecimal a binario para encontrar los campos op:

```
(Bits:   31  28  26                                  5    2    0)
         0000 0000 1010 1111 1000 0000 0010 0000
```

Miramos en el campo op para determinar la operación. Basándonos en la figura 2.25, cuando los bits 31-29 son 000 y los bits 28-26 son 000 se trata de una instrucción de R-format. Así pues, la instrucción binaria se reconstruye a partir de los campos de R-format listados en la figura 2.26:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

La porción inferior de la figura 2.19 determina la operación de una instrucción de R-format. En este caso, los bits 5-3 son 100 y los bits 2-0 son 000, lo que significa que este patrón binario representa la instrucción de la suma (`add`).

Descodificamos el resto de la instrucción mirando los valores de los campos. Los valores decimales son 5 para el campo rs, 15 para el rt y 16 para el rd (el *shamt* no se usa). La figura 2.14 indica que estos números representan los registros `$a1`, `$t7` y `$s0`. Ahora podemos mostrar la instrucción de ensamblador:

```
add $s0,$a1,$t7
```

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl				swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

FIGURA 2.19 Codificación de las instrucciones MIPS. Esta notación da el valor de un campo por fila y por columna. Por ejemplo, la parte superior de la figura muestra cargar palabra `load word` en la fila número 4 (100_{dos} para los bits 31-29 de la instrucción) y la columna número 3 (011_{dos} para los bits 28-26 de la instrucción), así que el valor correspondiente del campo op (bits 31-26) es 100011_{dos}. El resaltado significa que el campo está utilizado en alguna otra parte. Por ejemplo, el `R-format` en la fila 0 y la columna 0 (op = 000000_{dos}) está definido en la parte inferior de la figura. Por tanto, restar `subtract` en la fila 4 y la columna 2 de la sección inferior significa que el campo funct (bits 5-0) de la instrucción es 100010_{dos} y el campo op (bits 31-26) es 000000_{dos}. El valor `punto flotante` en la fila 2 y columna 1 se define en la figura 3.18 en el capítulo 3. `Bltz/gez` es el código de operación (opcode) para cuatro instrucciones que se encuentran en el apéndice B: `bltz`, `bgez`, `bltzal` y `bgezal`. Este capítulo describe las instrucciones dadas con el nombre completo coloreado, mientras que el capítulo 3 describe las instrucciones dadas con mnemónicos coloreados. El apéndice B trata todas las instrucciones.

Nombre	Campos						Comentarios
Tamaño del campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas las instrucciones MIPS tienen 32 bits
R-format	op	rs	rt	rd	shamt	funct	Formato de instrucciones aritméticas
I-format	op	rs	rt	address/immediate			Formato para transferencias, bifurcaciones, imm.
J-format	op	target address					Formato de las instrucciones de salto incondicional

FIGURA 2.20 Formatos de instrucción MIPS.

La figura 2.20 muestra todos los formatos de instrucción MIPS. La figura 2.1 en la página 78 muestra el lenguaje ensamblador de MIPS tratado en este capítulo; la parte no estudiada de instrucciones MIPS se ocupa principalmente de la aritmética, que veremos en el capítulo siguiente.

Autoevaluación

I. ¿Cuál es el rango de direcciones para las bifurcaciones o saltos condicionales en MIPS (K = 1024)?

1. Direcciones entre 0 y 64K – 1.
2. Direcciones entre 0 y 256K – 1.
3. Direcciones desde aproximadamente 32K antes de la bifurcación hasta 32K después.
4. Direcciones desde aproximadamente 128K antes de la bifurcación hasta 128K después.

II. ¿Cuál es el rango de direcciones para saltar y saltar-y-enlazar en MIPS (M = 1024K)?

1. Direcciones entre 0 y 64M – 1.
2. Direcciones entre 0 y 256M – 1.
3. Direcciones desde aproximadamente 32M antes del salto hasta aproximadamente 32M después.
4. Direcciones desde aproximadamente 128 M antes del salto hasta aproximadamente 128M después.
5. Cualquiera dentro de un bloque de direcciones de 64M donde el PC suministra los 6 bits de mayor peso.
6. Cualquiera dentro de un bloque de direcciones de 256M donde el PC suministra los 4 bits de mayor peso.

III. ¿Cuál es la instrucción del lenguaje ensamblador MIPS correspondiente a la instrucción máquina con valor $0000\ 0000_{hex}$?

1. `j`
2. `R-format`
3. `addi`
4. `sll`
5. `mfc0`
6. El opcode está indefinido: no hay instrucción legal que corresponda a 0.

2.11 Paralelismo e instrucciones: sincronización

La programación paralela es más fácil cuando las tareas son independientes, pero a menudo es necesaria una cooperación entre ellas. Cooperación significa habitualmente que algunas tareas están escribiendo nuevos valores que otra tareas necesitan leer. Para saber cuando una tarea ha finalizado la escritura y, por lo tanto, otra tarea ya puede hacer una lectura segura, las tareas necesitan sincronizarse. Si no hay esta sincronización existe el peligro de una **carrera de datos** (*data race*), en la que los resultados de un programa pueden variar dependiendo del orden en el que ocurran ciertos sucesos.

Carrera de datos: Dos accesos a memoria producen una carrera de datos si provienen de dos hilos (threads) diferentes, los accesos son a la misma posición de memoria, al menos uno es una escritura y ocurren uno después del otro.

Por ejemplo, pensemos otra vez en la analogía de los ocho periodistas escribiendo un artículo periodístico que ya hemos utilizado en la página 43 del capítulo 1. Suponga que un periodista necesita leer todas las secciones anteriores antes de escribir las conclusiones. Así, necesita saber cuando los otros periodistas han terminado sus secciones, para no tener que preocuparse de cambios que puedan producirse posteriormente. Es decir, tienen que sincronizar la escritura y lectura de cada sección para que las conclusiones sean coherentes con lo que se ha escrito en las anteriores secciones.

En computación, los mecanismos de sincronización se construyen, típicamente, con rutinas software a nivel de usuario que se apoyan en instrucciones de sincronización de bajo nivel (*hardware-supplied*). En esta sección nos centraremos en la implementación de las operaciones de sincronización *lock* (bloquear) y *unlock* (desbloquear). Estas operaciones pueden utilizarse de forma directa para crear regiones a las que sólo puede acceder un procesador, llamadas regiones de exclusión mutua, o para implementar mecanismos de sincronización más complejos.

Para implementar la sincronización en un sistema multiprocesador se requiere, como capacidad crítica, un conjunto de primitivas hardware que nos permitan leer y modificar posiciones de memoria de forma atómica. Es decir, nada se puede interponer entre la lectura y la escritura de la posición de memoria. Sin esta capacidad, la construcción de las primitivas básicas de sincronización tendría un coste demasiado elevado que además aumentaría con el número de procesadores.

Hay muchas formulaciones alternativas de las primitivas hardware básicas y en todas ellas se dispone de la capacidad de lectura y modificación atómica de una posición de memoria, junto con alguna forma de indicar si la lectura y la escritura se realizaron de forma atómica. En general, los diseñadores no esperan que los usuarios empleen las primitivas hardware básicas pero, por el contrario, esperan que los programadores de sistemas utilicen estas primitivas para construir bibliotecas de sincronización; este proceso es complejo y difícil.

Comenzaremos con la descripción de una de estas primitivas hardware y mostraremos cómo puede utilizarse para la construcción de una primitiva básica de sincronización. Una operación típicamente utilizada en la construcción de operaciones de sincronización es el *intercambio atómico*, que intercambia lo valores de un registro y una posición de memoria.

Como ejemplo de utilización de esta operación para construir una primitiva básica de sincronización, se diseñará un bloqueo simple donde el valor 0 se usa para indicar que el acceso a la variable protegida está libre (bloqueo = 0) y 1 para indicar que no lo está (bloqueo = 1). Un procesador intenta poner el valor 1 en el bloqueo intercambiando un registro con el valor 1 y la posición de memoria correspondiente al bloqueo. Si algún otro procesador ya ha fijado el valor del bloqueo a 1, el intercambio devuelve el valor 1, y en caso contrario devuelve el valor 0. En este último caso además el bloqueo se pone a 1 para evitar que cualquier otro procesador pueda poner a 1 el bloqueo.

Por ejemplo, consideremos que dos procesadores intentan hacer el intercambio simultáneamente; uno de los procesadores hará el intercambio en primer lugar, devolviendo un 0 como resultado, y cuando el segundo intenta hacer el intercambio devolverá como resultado un 1. La clave para que la primitiva de intercambio pueda ser usada para implementar la sincronización es que la operación es atómica: el intercambio es indivisible, y cuando se solicitan dos intercambios simultáneos el hardware se encarga de reordenarlos. Es imposible que, cuando dos procesadores intentan poner a 1 la variable de sincronización usando el intercambio, ambos crean que han puesto la variable a 1.

La implementación de una operación de memoria atómica plantea algunos retos en el diseño del procesador, porque se necesita que tanto la lectura como la escritura sea una instrucción única e ininterrumpible.

Un alternativa es disponer de una pareja de instrucciones en donde la segunda instrucción devuelve un valor que indica si la pareja fue ejecutada de forma atómica. La pareja de instrucciones es efectivamente atómica si todas las operaciones restantes ejecutadas por cualquier procesador se ha hecho antes o después de la ejecución de la pareja. De esta forma, cuando un pareja de instrucciones es efectivamente atómica, ningún otro procesador puede cambiar el valor entre las instrucciones de la pareja.

En el MIPS esta pareja de instrucciones está formada por una instrucción de carga y almacenamiento especiales, *carga enlazada* (*load linked*) y *almacenamiento condicional* (*store conditional*), respectivamente. Estas instrucciones se utilizan en secuencia: el almacenamiento condicional falla si los contenidos de la posición de memoria especificada en la carga enlazada se cambian antes de la instrucción de almacenamiento condicional. El almacenamiento condicional guarda el valor de un registro en memoria y además pone el valor 1 en este registro si la operación termina con éxito, o el valor 0 si falla. Dado que la carga devuelve el valor inicial y el almacenamiento el valor 1 sólo si tiene éxito, la siguiente secuencia de instrucciones implementa un intercambio atómico en la posición de memoria especificada por el contenido de `$s1`:

```
try: add  $t0, $zer0, $s4   ; copia valor de intercambio
     ll   $t1, 0($s1)       ; carga enlazada
     sc   $t0, 0($s1)       ; almacenamiento condicional
     beq  $t0, $zero, try   ; salto, almacenamiento fallido
     add  $s4, $zero, $t1   ; llevar valor cargado a $s4
```

El resultado de esta secuencia es un intercambio atómico entre los contenidos de `$s4` y de la posición de memoria especificada por `$s4`. Si cualquier otro procesador interviene y modifica el valor de la memoria entre las instrucciones `ll` y `sc`, la instrucción `sc` devuelve el valor 0 en `$t0` y la secuencia de código se ejecuta otra vez.

Extensión: Aunque el intercambio atómico se ha presentado en un entorno de sincronización de multiprocesadores, es útil también para la sincronización, por parte del sistema operativo, de procesos en un sistema con un único procesador. Para asegurarse de que nada interfiere en el procesador, el almacenamiento condicional falla, también, si se produce un cambio de contexto del procesador entre las dos instrucciones (véase capítulo 5).

Dado que el almacenamiento condicional falla si otro procesador intenta ejecutar una instrucción de almacenamiento con la dirección de la carga enlazada o si se produce una excepción, se deben elegir cuidadosamente qué instrucciones se insertan entre las dos instrucciones. En particular, solamente son seguras las instrucciones registro-registro; con cualquier otro tipo de instrucciones es posible que se creen situaciones de punto muerto (*deadlock*) y que el procesador no pueda completar nunca la instrucción `sc` debido a fallos de página repetitivos. Además, el número de instrucciones entre la carga enlazada y el almacenamiento condicional debe ser pequeño con el objetivo de minimizar la probabilidad de que, o bien un suceso no relacionado, o bien otro procesador hagan que el almacenamiento condicional falle frecuentemente.

Una ventaja del mecanismo carga enlazada/almacenamiento condicional es que puede utilizarse para construir otras primitivas de sincronización, tales como *comparación e intercambio atómico* o *búsqueda-e-incremento atómico*, que se utilizan en algunos modelos de programación paralela. Estas primitivas tienen más instrucciones entre `ll` y `sc`.

Autoevaluación

¿Cuándo se utilizan primitivas como carga enlazada y almacenamiento condicional?

1. Cuando los hilos cooperantes de un programa paralelo necesitan sincronizarse para leer y escribir datos compartidos correctamente
2. Cuando los procesos cooperantes de un monoprocesador necesitan sincronizarse para leer y escribir datos compartidos.

2.12 Traducción e inicio de un programa

Esta sección describe los cuatro pasos necesarios para transformar un programa en C que se encuentre en un fichero de un disco en un programa ejecutable en un computador. La figura 2.21 muestra la secuencia de traducción. Algunos sistemas combinan estos pasos para reducir el tiempo de la traducción, pero éstas son las cuatro fases lógicas que todos los programas necesitan. Esta sección sigue esta jerarquía de traducción.

Compilador

El compilador transforma el programa en C en un *programa en lenguaje ensamblador*, una forma simbólica de lo que entiende la máquina. Los programas en lenguaje de alto nivel utilizan bastantes menos líneas de código que el lenguaje ensamblador, por lo que la productividad del programador es mucho más alta.

En 1975, muchos sistemas operativos y ensambladores fueron escritos en **lenguaje ensamblador** porque las memorias eran pequeñas y los compiladores eran ineficientes. El aumento de 500 000 veces la capacidad de memoria por cada chip de DRAM ha reducido las preocupaciones por el tamaño del programa, y los actuales compiladores que optimizan pueden producir hoy programas en lenguaje ensamblador prácticamente tan buenos como los de un experto en lenguaje ensamblador, e incluso a veces mejores en el caso de programas grandes.

Lenguaje ensamblador: lenguaje simbólico que se puede traducir a binario.

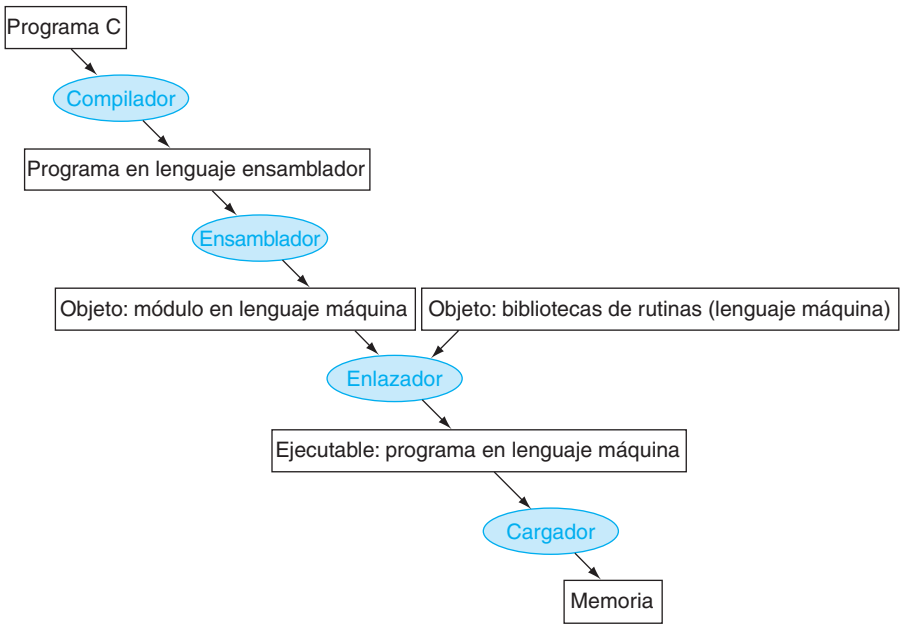


FIGURA 2.21 Una jerarquía de traducción para C. Un programa en lenguaje de alto-nivel es primero compilado en un programa del lenguaje ensamblador y después se ensambla en un módulo objeto en lenguaje máquina. El enlazador (*linker*) combina los múltiples módulos con rutinas de la biblioteca para resolver todas las referencias. Luego el cargador pone el código máquina en las posiciones de memoria apropiadas para la ejecución por el procesador. Para acelerar el proceso de la traducción, algunos pasos se saltan o se combinan. Algunos compiladores producen módulos objeto directamente, y algunos sistemas utilizan cargadores enlazadores que realizan los dos últimos pasos. Para identificar el tipo de fichero, UNIX usa una convención de sufijos para los ficheros: los ficheros fuente de C se denominan `x.c`, y los ficheros en ensambador son `x.s`, los ficheros objeto se denominan `x.o`, las rutinas de la biblioteca enlazadas estáticamente son `x.a`, las rutas de la biblioteca dinámicamente enlazadas son `x.so`, y los ficheros ejecutables por defecto se llaman `a.out`. El MS-DOS utiliza las extensiones `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL` y `.EXE` para el mismo efecto.

Ensamblador

Puesto que el lenguaje ensamblador es la interfaz para programas de alto nivel, el ensamblador puede también tratar variaciones comunes de las instrucciones del lenguaje máquina como si fueran instrucciones de pleno derecho. El hardware no necesita implementar estas instrucciones; sin embargo, su aparición en el lenguaje ensamblador simplifica la traducción y la programación. Tales instrucciones se llaman **pseudoinstrucciones**.

Pseudoinstrucciones: variación común de las instrucciones del lenguaje ensamblador, generalmente tratadas como si fueran instrucciones de pleno derecho.

Como se ha mencionado anteriormente, el hardware del MIPS asegura que el registro `$zero` tiene siempre el valor 0. Es decir, siempre que se utilice el registro `$zero` éste proporciona un 0, y el programador no puede cambiar este valor. El registro `$zero` se utiliza para crear la instrucción de lenguaje ensamblador `move` (mover) que copia el contenido de un registro a otro. Así, el ensamblador MIPS acepta esta instrucción aunque no se encuentre en la arquitectura MIPS:

```
move $t0,$t1        # el registro $t0 toma el valor del
                    # registro $t1
```

El ensamblador convierte esta instrucción de lenguaje ensamblador en la instrucción del lenguaje máquina equivalente a la instrucción siguiente:

```
add $t0,$zero,$t1   # el registro $t0 toma el valor 0 +
                    # registro $t1
```

El ensamblador de MIPS también convierte `blt` (bifurcación en menor que, *branch on less than*) en las dos instrucciones `slt` y `bne` mencionadas en el ejemplo de la página 128. Otros ejemplos incluyen `bgt`, `bge` y `ble`. También convierte bifurcaciones a posiciones lejanas en una bifurcación o salto condicional y un salto incondicional. Tal y como hemos mencionado anteriormente, el ensamblador de MIPS puede incluso permitir constantes de 32 bits para ser cargadas en un registro, a pesar del límite de los 16 bits de las instrucciones inmediatas.

Resumiendo, las pseudoinstrucciones dan a MIPS un repertorio de instrucciones de lenguaje ensamblador más rico que aquellas que implementa el hardware. El único coste es la reserva de un registro, `$at`, para ser usado por el ensamblador. Si se desea escribir programas en ensamblador es conveniente utilizar pseudoinstrucciones para simplificar la tarea. De todos modos, para comprender el hardware de MIPS y para estar seguro de obtener las mejores prestaciones será mejor estudiar las instrucciones MIPS reales que aparecen en las figuras 2.1 y 2.19.

Los ensambladores también aceptan números en una gran variedad de bases. Además de binario y de decimal, aceptan una base que es más compacta que la binaria y más fácil de convertir en patrones de bits. Los ensambladores del MIPS utilizan la hexadecimal.

Este tipo de características son oportunas, pero la principal tarea de un ensamblador es ensamblar en código máquina. El ensamblador convierte el programa en lenguaje ensamblador en un *fichero objeto*, que es una combinación de instrucciones en lenguaje máquina, datos e información necesaria para colocar las instrucciones correctamente en memoria.

Para producir la versión binaria de cada instrucción del programa en lenguaje ensamblador, el ensamblador debe determinar las direcciones que corresponden a todas las etiquetas. Los ensambladores mantienen pistas de las etiquetas usadas en bifurcaciones o saltos condicionales y en instrucciones de transferencia de datos en una **tabla de símbolos**. Como es de esperar, la tabla contiene pares de símbolos y direcciones.

Tabla de símbolos: tabla que relaciona o empareja nombres de etiquetas con las direcciones de las palabras de la memoria que ocupan las instrucciones.

El fichero objeto para sistemas UNIX contiene típicamente seis partes diferenciadas:

- La *cabecera del fichero objeto* describe el tamaño y la posición de las otras partes del fichero objeto.
- El *segmento de texto* contiene el código del lenguaje máquina.
- El *segmento de datos estáticos* contiene los datos asignados para la duración del programa. (UNIX permite que los programas utilicen tanto *datos estáticos*, que son asignados a lo largo del programa, como *datos dinámicos*, que pueden crecer o contraerse según lo requerido por el programa. Véase figura 2.13.)
- La *información de reubicación* identifica instrucciones y palabras de datos que dependen de direcciones absolutas cuando el programa se carga en memoria.

- La *tabla de símbolos* contiene las etiquetas restantes que no están definidas, como las referencias externas.
- La *información de depuración* contiene una descripción concisa de cómo fueron compilados los módulos, para que el depurador pueda asociar las instrucciones máquina con ficheros fuente en C y hacer las estructuras de datos legibles.

Las siguiente subsecciones muestran cómo utilizar aquellas rutinas que ya han sido ensambladas, por ejemplo las rutinas de biblioteca.

Enlazador (*linker*)

Lo que hemos presentado hasta ahora sugiere que un simple cambio en una línea de un procedimiento requiere compilar y ensamblar el programa completo. La retraducción completa es una pérdida terrible de recursos de computación. Esta repetición es particularmente costosa para las rutinas estándares de biblioteca, porque los programadores deberían compilar y ensamblar rutinas que por definición casi nunca cambian. Una alternativa es compilar y ensamblar cada procedimiento independientemente, de modo que un cambio en una línea implicara compilar y ensamblar solamente un procedimiento. Esta alternativa requiere un nuevo programa de sistemas, denominado **enlazador** o editor de enlaces (*linker* o *link editor*), que toma todos los programas en lenguaje máquina ensamblados independientemente y los une.

Enlazador (también llamado **editor de enlaces** o, a veces, montador): programa de sistemas que combina independientemente programas en lenguaje máquina ensamblados y que resuelve todas las etiquetas indefinidas en un fichero ejecutable.

Hay tres pasos que debe seguir el enlazador:

1. Colocar simbólicamente en memoria módulos de código y de datos.
2. Determinar las direcciones de los datos y de las etiquetas de las instrucciones.
3. Arreglar tanto las referencias internas como las externas.

El enlazador utiliza la información de reubicación y la tabla de símbolos de cada módulo objeto para resolver todas las etiquetas indefinidas. Este tipo de referencias ocurren en instrucciones de bifurcaciones o salto condicional, instrucciones de salto incondicional y direcciones de datos. Por tanto, el trabajo de este programa se parece al de un editor: encuentra las direcciones antiguas y las sustituye por las nuevas direcciones. En inglés, el término *link editor* (editor de enlaces) o, abreviadamente, *linker* (enlazador) viene de *editing*. La razón por la que tiene sentido un enlazador es que es mucho más rápido unir código que volverlo a compilar y ensamblar.

Si todas las referencias externas quedan resueltas, el enlazador determina a continuación las posiciones de memoria que ocupará cada módulo. La figura 2.13 en la página 120 muestra la convención MIPS para la asignación del programa y de los datos en memoria. Debido a que los ficheros se ensamblan aisladamente, el ensamblador no podría saber dónde se situarían las instrucciones y los datos de un módulo y cómo serían puestos en relación a otros módulos. Cuando el enlazador coloca un módulo en memoria todas las referencias *absolutas*, es decir, direcciones de memoria que no son relativas a un registro, se deben volver a *reubicar* para reflejar su localización verdadera.

Fichero ejecutable: programa funcional ejecutable en el formato de un fichero objeto que no contiene ninguna referencia no resuelta y sí contiene la información de la reubicación, la tabla de símbolos o la información para la depuración.

El enlazador produce un **fichero ejecutable** que se puede ejecutar en un computador. Típicamente este fichero tiene el mismo formato que un fichero objeto, salvo que no contiene ninguna referencia sin resolver. Es posible tener ficheros parcialmente enlazados (montados), tales como rutinas de biblioteca, que todavía contengan direcciones sin resolver y que por tanto aparezcan como ficheros objeto.

Edición de enlaces de ficheros objeto

EJEMPLO

Enlazar los dos ficheros objeto de abajo. Mostrar las direcciones actualizadas de las primeras instrucciones del fichero ejecutable terminado. Mostramos las instrucciones en lenguaje ensamblador simplemente para hacer el ejemplo comprensible; en realidad las instrucciones serían números.

Obsérvese que en los ficheros objeto hemos resaltado las direcciones y los símbolos que deben ser actualizados en el proceso de enlace: las instrucciones que se refieren a las direcciones de los procedimientos `A` y `B` y las instrucciones que se refieren a las direcciones de las palabras de datos `X` e `Y`.

Cabecera de fichero objeto			
	Nombre	Procedimiento `A`	
	Tamaño texto	100_{hex}	
	Tamaño datos	20_{hex}	
Segmento de texto	Dirección	Instrucción	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Segmento de datos	0	`(X)`	
	…	…	
Información de reubicación	Dirección	Tipo de instrucción	Dependencia
	0	`lw`	`X`
	4	`jal`	`B`
Tabla de símbolos	Etiqueta	Dirección	
	`X`	—	
	`B`	—	
Cabecera de fichero objeto			
	Nombre	Procedimiento `B`	
	Tamaño de texto	200_{hex}	
	Tamaño de datos	30_{hex}	
Segmento de texto	Dirección	Instrucción	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Segmento de datos	0	`(Y)`	
	…	…	
Información de reubicación	Dirección	Tipo de instrucción	Dependencia
	0	`sw`	`Y`
	4	`jal`	`A`
Tabla de símbolos	Etiqueta	Dirección	
	`Y`	—	
	`A`	—	

RESPUESTA

El procedimiento A necesita encontrar la dirección de la variable etiquetada como X para ponerla en la instrucción de carga y buscar la dirección del procedimiento B para colocarla en la instrucción `jal`. El procedimiento B necesita la dirección de la variable etiquetada como Y para la instrucción de almacenamiento y la dirección del procedimiento A para su instrucción `jal`.

De la figura 2.13, en la página 120, sabemos que el segmento de texto empieza en la dirección $40\ 0000_{hex}$ y el segmento de datos en $1000\ 0000_{hex}$. El texto del procedimiento A se coloca en la primera dirección y sus datos en la segunda. La cabecera del fichero objeto del procedimiento A indica que su texto son 100_{hex} bytes y sus datos son 20_{hex} bytes, así que la dirección de inicio para el texto del procedimiento B son $40\ 0100_{hex}$ y sus datos empiezan a partir de $1000\ 0020_{hex}$.

Cabecera del fichero ejecutable		
	Tamaño del texto	300_{hex}
	Tamaño de los datos	50_{hex}
Segmento de texto	Dirección	Instrucción
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal` $40\ 0100_{hex}$
	…	…
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal` $40\ 0000_{hex}$
	…	…
Segmento de datos	Dirección	
	$1000\ 0000_{hex}$	(X)
	…	…
	$1000\ 0020_{hex}$	(Y)

Ahora el enlazador actualiza los campos de dirección de las instrucciones. Éste utiliza el campo de tipo de instrucción para saber el formato de la dirección que debe ser editada. Vemos dos tipos aquí:

1. Las instrucciones `jal` son fáciles porque utilizan un modo de direccionamiento pseudodirecto. A la instrucción `jal` de la dirección $40\ 0004_{hex}$ le ponemos $40\ 0100_{hex}$ (la dirección del procedimiento B) en su campo de dirección, y a la instrucción `jal` de $40\ 0104_{hex}$ le ponemos $40\ 0000_{hex}$ (la dirección del procedimiento A) en su campo de dirección.

2. Las direcciones de carga y almacenamiento son más difíciles porque son relativas a un registro base. En este ejemplo utilizamos el puntero global como registro base. La figura 2.17 muestra que `$gp` está inicializado a $1000\ 8000_{hex}$. Para obtener la dirección $1000\ 0000_{hex}$ (la dirección de la palabra X) colocamos 8000_{hex} en el campo de dirección de `lw` en la dirección $40\ 0000_{hex}$. El capítulo 3 explica la aritmética del computador en complemento a dos para 16 bits, por la cual 8000_{hex} en el campo de dirección produce $1000\ 0000_{hex}$ como dirección. Similarmente, colocamos 8020_{hex} en el campo de dirección de la instrucción *sw* en la dirección $40\ 0100_{hex}$ para obtener la dirección $1000\ 0020_{hex}$ (la dirección de la palabra Y).

Extensión: Reiteramos que las instrucciones MIPS se alinean por palabra, así `jal` descarta los dos bits de la derecha para aumentar el rango de direcciones de la instrucción. De este modo, utiliza 26 bits para formar una dirección de 28 bits. La dirección real en los 26 bits de la instrucción `jal` del ejemplo es $10\ 0040_{hex}$ en lugar de $40\ 0100_{hex}$.

Cargador

Una vez que el fichero ejecutable está en disco, el sistema operativo lo lee para pasarlo a la memoria y lo inicia. En los sistemas UNIX el **cargador** sigue estos pasos:

Cargador: programa de sistema que coloca un programa objeto en la memoria principal de modo que esté listo para ejecutarse.

1. Lee la cabecera del fichero ejecutable para determinar el tamaño de los segmentos de texto y de datos.
2. Crea un espacio de direcciones suficientemente grande para el texto y los datos.
3. Copia las instrucciones y los datos del fichero ejecutable en memoria.
4. Copia los parámetros del programa principal (si los hay) sobre la pila.
5. Inicializa los registros de la máquina y actualiza el puntero de pila con la primera posición libre.
6. Salta a una rutina de inicio, que copia los parámetros en los registros de argumento y llama a la rutina principal del programa. Cuando la rutina principal retorna, la rutina de inicio (*start-up*) termina el programa con una llamada de sistema de `salida` (*exit*).

Las secciones B.3 y B.4 del apéndice B describen enlazadores y cargadores más detalladamente.

Bibliotecas enlazadas (montadas) dinámicamente

La primera parte de esta sección describe el enfoque tradicional para enlazar bibliotecas antes de que se ejecute el programa. Aunque este acercamiento estático es la manera más rápida para llamar rutinas de biblioteca, tiene algunas desventajas:

- Las rutinas de biblioteca se convierten en parte del código ejecutable. Si se lanza una nueva versión de la biblioteca que arregla errores o que soporta nuevos dispositivos hardware, el programa estáticamente enlazado sigue usando la vieja versión.
- Carga la biblioteca entera aunque no se utilice toda cuando el programa se ejecuta. La biblioteca puede ser grande con relación al programa; por ejemplo, la biblioteca estándar de C es de 2.5 MB.

Bibliotecas enlazadas dinámicamente (DLL): procedimientos de una biblioteca que se enlazan durante la ejecución del programa.

Estas desventajas nos conducen a las **bibliotecas enlazadas (montadas) dinámicamente (DLLs)**, donde las rutinas de biblioteca no se enlazan ni se cargan hasta que el programa se ejecuta. Las rutinas de biblioteca y de programa albergan información extra sobre la posición de los procedimientos no locales y de sus nombres. En la versión inicial de DLLs, el cargador ejecutaba un enlazador dinámico, que usaba la información extra en el fichero para encontrar las bibliotecas apropiadas y actualizar todas las referencias externas.

El inconveniente de la versión inicial de DLLs era que todavía enlazaba todas las rutinas de la biblioteca que pueden ser llamadas en lugar de enlazar sólo aquellas que se llaman durante la ejecución del programa. Esta observación condujo a la versión de las DLLs de procedimiento de enlazado tardío, donde cada rutina se enlaza ("linka") solamente *después* de ser llamada.

Como muchas veces ocurre en nuestro campo, este truco cuenta con un nivel de indirección. La figura 2.22 nos muestra la técnica. Comienza con las rutinas no locales que llaman a un sistema de rutinas simuladas (*dummy*) al final del programa, con una entrada por rutina no local. Cada una de estas entradas simuladas contiene un salto indirecto.

La primera vez que se llama a la rutina de biblioteca, el programa llama a la entrada simulada y sigue el salto indirecto. Apunta al código que pone un número en un registro para identificar la rutina de biblioteca deseada y después salta al enlazador-cargador dinámico. El enlazador-cargador busca la rutina deseada, la recoloca (*remap*) y cambia la dirección en la posición del salto indirecto para apuntar a esa rutina. Entonces salta a ella. Cuando la rutina termina vuelve al sitio original del programa invocador o llamador. A partir de entonces, salta indirectamente a la rutina sin los saltos adicionales.

Resumiendo, las DLLs requieren espacio extra para la información que necesitan para el enlace dinámico, pero no requieren copiar o enlazar bibliotecas enteras. Tienen mucha sobrecarga la primera vez que se llama a la rutina, pero después solamente deben ejecutar un único salto indirecto. Obsérvese que ningún retorno de biblioteca tiene sobrecarga extra. Windows de Microsoft confía extensamente en bibliotecas dinámicamente enlazadas tardías, y es también la manera normal de ejecutar programas sobre sistemas UNIX.

Poner en marcha un programa Java

La discusión anterior parte del modelo tradicional de ejecutar un programa, donde el énfasis se pone en la rapidez de ejecución para un programa destinado a una arquitectura del repertorio de instrucciones específica, o incluso a una implementación específica de dicha arquitectura. De hecho, es posible ejecutar los programas de Java de un modo parecido a los de C. Sin embargo, Java fue inventado con una serie de objetivos diferentes. Uno era poder ejecutar rápidamente sin incidentes en cualquier computador, aunque se ralentizara el tiempo de ejecución.

La figura 2.23 muestra la traducción típica y los pasos de la ejecución para Java. Más que compilar el lenguaje ensamblador de un computador destino, Java primero compila instrucciones que son fáciles de interpretar: el repertorio de ins-

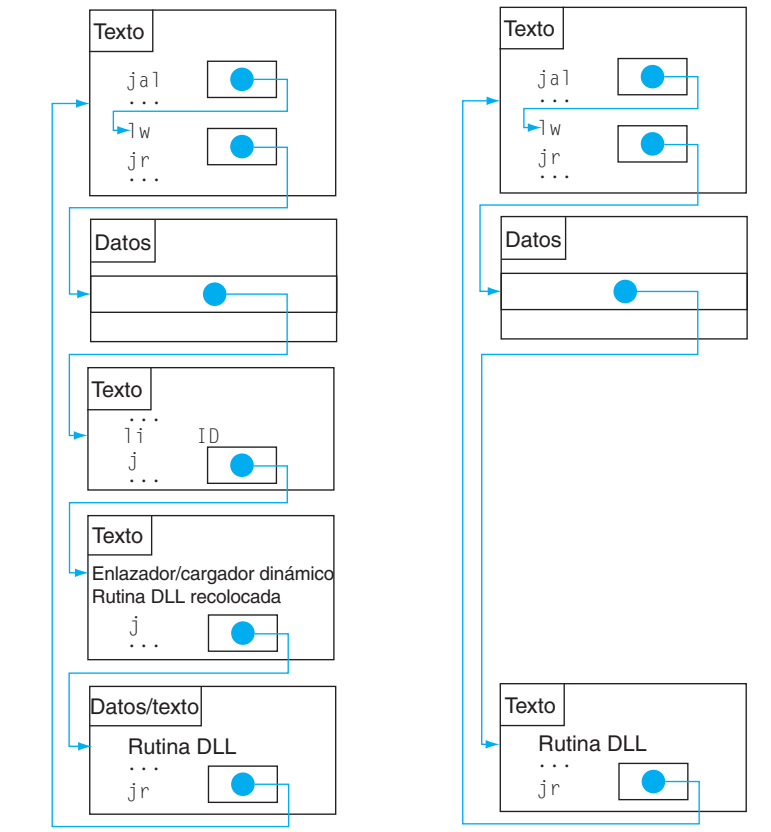


FIGURA 2.22 Biblioteca enlazada (montada) dinámicamente mediante un procedimiento de enlazado tardío. (a) Pasos para la primera vez que se llama a la rutina DLL. (b) Los pasos para buscar la rutina, recolocarla y enlazarla se saltan en las llamadas siguientes. Como veremos en el capítulo 5, el sistema operativo puede evitar copiar la rutina deseada si la recoloca usando el gestor de memoria virtual.

trucciones ***Java bytecode*** (véase **sección 2.15** en el CD). Este repertorio de instrucciones se diseña para que sea cercano al lenguaje Java de modo que este paso de la compilación sea trivial. No se realiza prácticamente ninguna optimización. Como el compilador de C, el compilador de Java comprueba los tipos de datos y produce la operación apropiada para cada uno. Los programas de Java se distribuyen en la versión binaria de estos bytecodes.

Java bytecode: instrucciones de un repertorio de instrucciones diseñado para interpretar los programas de Java.

Un software intérprete, llamado **máquina virtual de Java** (*Java Virtual Machine*, JVM), puede ejecutar los *Java bytecodes*. Un intérprete es un programa que simula una arquitectura del repertorio de instrucciones. Por ejemplo, el simulador de MIPS usado con este libro es un intérprete. No hay necesidad de un paso de ensamblaje aparte, puesto que la traducción es tan simple que el compilador rellena las direcciones o la JVM las busca durante la ejecución.

Máquina virtual de Java: programa que interpreta los *Java bytecodes*.

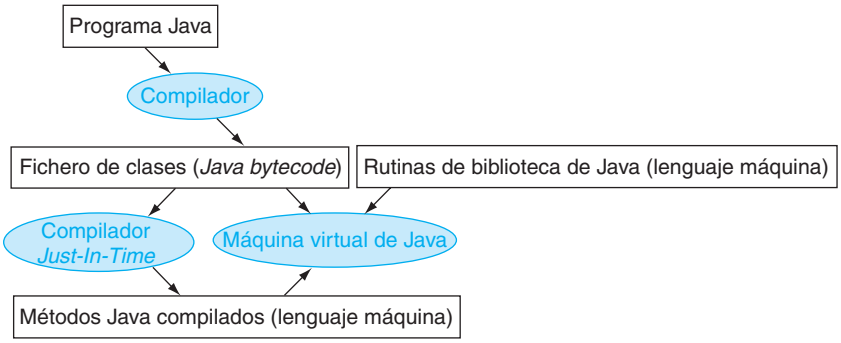


FIGURA 2.23 Una secuencia de traducción para Java. Un programa Java se compila primero en una versión binaria de código Java, con todas las direcciones definidas por el compilador. El programa Java está entonces listo para ejecutarse en un software intérprete, llamado máquina virtual de Java (JVM). La JVM enlaza con los métodos deseados en la biblioteca de Java mientras el programa se está ejecutando. Para lograr mejores prestaciones, la JVM puede invocar el compilador *Just–In-Time* (JIT), el cual compila selectivamente métodos en el lenguaje máquina nativo de la máquina en la que se está ejecutando.

Lo bueno de la interpretación directa es la portabilidad. La disponibilidad del software de la máquina virtual de Java significó que muchos podrían escribir y ejecutar programas Java poco después que Java fuera dado a conocer. Las máquinas virtuales de Java se encuentran hoy en millones de dispositivos de todo tipo, desde los teléfonos móviles a los buscadores de Internet.

La desventaja de esta traducción es que se obtienen unas prestaciones menores. La increíble mejora de las prestaciones en las décadas de 1980 y 1990 hicieron la interpretación viable para muchos usos importantes, pero la disminución de las prestaciones en un factor de 10 cuando eran comparados con los programas de C tradicionalmente compilados hicieron a Java poco atractivo para algunas aplicaciones.

Para preservar la portabilidad y mejorar la velocidad de ejecución, la siguiente fase del desarrollo de Java fue lograr que los compiladores tradujeran *mientras* el programa funcionara. Tales **compiladores *Just-In-Time*** (JIT) hacen el perfil del programa ejecutado para buscar donde están los puntos "calientes" y los compilan en el repertorio de instrucciones nativo en el que la máquina virtual está funcionando. La parte compilada se guarda para la próxima vez que se ejecute el programa, de modo que pueda funcionar más rápido cada vez que se ejecute. Este reequilibrio entre traducción y compilación se desarrolla con el tiempo, de modo que los programas Java que se ejecutan frecuentemente sufren poco retraso debido a la interpretación.

Compiladores *Just-In-Time*: nombre dado normalmente a un compilador que funciona durante el tiempo de ejecución, traduciendo los segmentos de código interpretados al código nativo del computador.

A medida que los computadores son más rápidos y los compiladores pueden hacer más, y a medida que los investigadores inventan mejores modos de compilar Java improvisadamente, las diferencias en prestaciones entre Java y C o C++ van disminuyendo. La **sección 2.15** en el CD trata con una profundidad mucho mayor la implementación de compiladores Java, de los *Java bytecodes*, de JVM y de JIT.

Autoevaluación

¿Cuál de las ventajas de un intérprete sobre un traductor cree usted que fue más importante para los diseñadores de Java?

1. Facilidad para escribir un intérprete.
2. Mensajes de error mejores.
3. Un código de objeto más pequeño.
4. Independencia de la máquina.

2.13 Un ejemplo de ordenamiento en C para verlo todo junto

Uno de los riesgos de mostrar fragmentos de código del lenguaje ensamblador es que no tenemos ninguna idea de cómo es un programa completo en lenguaje ensamblador. En esta sección deducimos el código MIPS a partir de dos procedimientos escritos en C: uno para intercambiar elementos de una tabla y otro para ordenarlos.

```
void swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

FIGURA 2.24 Procedimiento C que intercambia dos posiciones en memoria. La subsección siguiente utiliza este procedimiento en un ejemplo de ordenación (`sort`).

El procedimiento intercambio (`swap`)

Comencemos con el código para el procedimiento de intercambio (`swap`) de la figura 2.24. Este procedimiento intercambia simplemente dos posiciones de memoria. Al traducir de C al lenguaje ensamblador a mano seguimos estos pasos generales:

1. Asignar los registros a las variables del programa.
2. Producir el código para el cuerpo del procedimiento.
3. Preservar los registros a lo largo de la invocación del procedimiento.

Esta sección describe el procedimiento de intercambio (`swap`) en estas tres partes, y concluye juntándolas.

Asignación de registro para el intercambio (`swap`)

Según lo mencionado en la página 112-113, la convención MIPS para el paso de parámetros es utilizar los registros `$a0`, `$a1`, `$a2` y `$a3`. Puesto que el intercambio tiene sólo dos parámetros, `v` y `k`, éstos se situarán en los registros `$a0` y `$a1`.

Únicamente se necesita otra variable, `temp`, que asociamos al registro `$t0` puesto que el intercambio es un procedimiento hoja (véase la página 116). Esta asignación de registros se corresponde con las declaraciones de variables de la primera parte del procedimiento de intercambio de la figura 2.24.

Código para el cuerpo del procedimiento de intercambio (`swap`)

Las líneas restantes del código C del intercambio son:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Recordemos que la dirección de memoria para MIPS se refiere a la dirección del byte, y por tanto que las palabras están realmente separadas por 4 bytes. Por tanto necesitamos multiplicar el índice `k` por 4 antes de sumarle la dirección. *Olvidar que las direcciones de palabras secuenciales difieren de 4 en vez de 1 es un error común en la programación en lenguaje ensamblador*. Por tanto, el primer paso es conseguir la dirección de `v[k]` multiplicando `k` por 4; mediante un desplazamiento a la izquierda de 2 posiciones.

```
sll  $t1, $a1,2   # reg $t1 = k * 4
add  $t1, $a0,$t1 # reg $t1 = v + (k * 4)
                  # reg $t1 tiene la dirección de v[k]
```

Ahora cargamos `v[k]` utilizando `$t1`, y luego `v[k+1]` añadiendo 4 a `$t1`:

```
lw   $t0, 0($t1)  # reg $t0 (temp) = v[k]
lw   $t2, 4($t1)  # reg $t2 = v[k + 1]
                  # se refiere al siguiente elemento de v
```

Luego almacenamos `$t0` y `$t2` en las direcciones intercambiadas:

```
sw   $t2, 0($t1)  # v[k] = reg $t2
sw   $t0, 4($t1)  # v[k+1] = reg $t0 (temp)
```

Ya hemos asignado los registros y hemos escrito el código para realizar las operaciones del procedimiento. Lo único que falta es el código para preservar los registros guardados usados dentro del intercambio. Puesto que no estamos utilizando los registros guardados en este procedimiento hoja, no hay nada que preservar.

El procedimiento completo de intercambio (`swap`)

Ahora estamos listos para la rutina completa, que incluye la etiqueta del procedimiento y el retorno del salto. Para hacerlo más fácil de seguir, identificamos en la figura 2.25 cada bloque del código con su propósito en el procedimiento.

El procedimiento de ordenación (`sort`)

Para asegurar que apreciamos el rigor de la programación en lenguaje ensamblador escogemos un segundo ejemplo más largo. En este caso, construiremos una rutina que llame al procedimiento de intercambio. Este programa ordena una tabla de números enteros usando la burbuja o la ordenación por intercambio, que

Cuerpo del procedimiento

```
swap: sll   $t1, $a1, 2        # reg $t1 = k * 4
      add   $t1, $a0, $t1      # reg $t1 = v + (k * 4)
                               # reg $t1 t1 tiene la dirección de v[k]
      lw    $t0, 0($t1)        # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)        # reg $t2 = v[k + 1]
                               # referencia al siguiente elemento de v
      sw    $t2, 0($t1)        # v[k] = reg $t2
      sw    $t0, 4($t1)        # v[k+1] = reg $t0 (temp)
```

Retorno del procedimiento

```
      jr    $ra                # retorna a la rutina que lo llamó
```

FIGURA 2.25 Código ensamblador MIPS del procedimiento intercambio (`swap`) en la figura 2.24.

es una de las más simples aunque no la ordenación más rápida. La figura 2.26 muestra la versión C del programa. De nuevo, presentamos este procedimiento en varios pasos y concluimos con el procedimiento completo.

```
void sort (int v[], int n)
{
   int i, j;
   for (i = 0; i < n; i += 1) {
      for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
          swap(v,j);
      }
   }
}
```

FIGURA 2.26 Procedimiento C que realiza la ordenación (sort) de una tabla `v`.

Asignación de registros para la ordenación (`sort`)

Los dos parámetros del procedimiento de ordenación (`sort`), `v` y `n`, están en los registros `$a0` y `$a1` y asignamos el registro `$s0` a `i` y el registro `$s1` a `j`.

Código del cuerpo del procedimiento de ordenación (`sort`)

El cuerpo del procedimiento consiste en dos lazos anidados *for* y una llamada al procedimiento de intercambio (`swap`) que incluye parámetros. El código se descompone del exterior hacia dentro.

El primer paso de la traducción es el primer lazo *for*:

```
for (i = 0; i < n; i += 1) {
```

Recordemos que la sentencia *for* de C tiene tres partes: inicialización, comprobación del lazo e incremento de iteración. Solo hace falta una instrucción para inicializar `i` a 0, la primera parte de la sentencia *for*:

```
move    $s0, $zero   # i = 0
```

(Recuérdese también que *move* es una pseudoinstrucción proporcionada por el ensamblador para ayudar al programador del lenguaje ensamblador; véase la página 141). De igual modo, solo se necesita instrucción incrementar `i`, la última parte de la sentencia *for*:

```
addi    $s0, $s0, 1 # i += 1
```

El lazo debería finalizar si `i < n` no se cumple o, dicho de otra manera, debería salir si `i ≥ n`. La instrucción *activar si menor que* activa el registro `$t0` a 1 si `$s0 < $a1`, y a 0 en cualquier otro caso. Puesto que deseamos probar si `$s0 ≥ $a1`, bifurcamos si el registro `$t0` es 0. Esta prueba necesita dos instrucciones:

```
for1tst:slt $t0, $s0, $a1 # reg $t0 = 0 si $s0 ≥ $a1 (i≥n)
        beq $t0, $zero,exit1 # ir a exit1 si $s0≥$a1 (i≥n)
```

La parte final del lazo salta hacia atrás hacia la prueba de lazo:

```
          j   for1tst   # salta para probar el lazo exterior
exit1:
```

El esquema del código del primer lazo *for* es por tanto*:*

```
      move   $s0, $zero    # i = 0
for1tst:slt $t0, $s0, $a1 # reg $t0 = 0 si $s0 ≥ $a1 (i≥n)
      beq    $t0, $zero,exit1 # ir a exit1 si $s0≥$a1 (i≥n)
             ...
             (cuerpo del primer lazo for)
             ...
      addi   $s0, $s0, 1 # i += 1
      j      for1tst     # salta para probar el lazo exterior
exit1:
```

¡*Voilà*! (En los ejercicios se explora la escritura de un código más rápido para lazos similares).

El segundo lazo *for* se parece a esto en C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

La parte de inicialización de este lazo es de nuevo una instrucción:

```
addi    $s1, $s0, -1 # j = i - 1
```

El decremento de `j` al final del lazo es también una instrucción:

```
addi    $s1, $s1, -1 # j -= 1
```

La prueba del lazo tiene dos partes. Salimos del lazo si cualquier condición falla, por tanto, el primer test debe salir del lazo si falla (`j < 0`):

```
for2tst:slti$t0, $s1, 0 # reg $t0 = 1 si $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # ir a exit2 si $s1<0 (j < 0)
```

Esta bifurcación pasará por alto el segundo test de condición. Si no lo salta, entonces `j ≥ 0`.

La segunda prueba sale del lazo si `v[j] > v[j + 1]` no es verdad, o bien si `v[j] ≤v[j + 1]`. Primero creamos la dirección multiplicando `j` por 4 (puesto que necesitamos una dirección del byte) y la sumamos la dirección base de `v`:

```
sll     $t1, $s1,2  # reg $t1 = j * 4
add     $t2, $a0,$t1# reg $t2 = v + (j * 4)
```

Ahora cargamos `v[j]`:

```
lw      $t3, 0($t2) # reg $t3  = v[j]
```

Puesto que sabemos que el segundo elemento es justo la palabra siguiente, sumamos 4 a la dirección en el registro `$t2` para conseguir `v[j + 1]`:

```
lw      $t4, 4($t2) # reg $t4  = v[j + 1]
```

El test de `v[j] ≤ v[j + 1]` es igual que el `v[j + 1] ≥ v[j]`. Por tanto, las dos instrucciones de la prueba de salida son:

```
slt     $t0, $t4, $t3       # reg $t0 = 0 si $t4 ≥ $t3
beq     $t0, $zero,exit2    # ir a exit2 si $t4 ≥ $t3
```

La parte final del lazo salta de nuevo atrás hacia el test del lazo interno:

```
j      for2tst    # salta al test del lazo interno
```

Combinando las piezas conjuntamente, el esquema del segundo lazo *for* aparece como:

```
        addi $s1, $s0, -1       # j = i - 1
for2tst:slti $t0, $s1, 0 # reg $t0 = 1 si $s1 < 0 (j<0)
        bne  $t0, $zero,exit2   # ir a exit2 si $s1<0 (j<0)
        sll  $t1, $s1,2         # reg $t1 = j * 4
        add  $t2, $a0,$t1       # reg $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # reg $t3  = v[j]
        lw   $t4, 4($t2)        # reg $t4  = v[j + 1]
        slt  $t0, $t4, $t3      # reg $t0 = 0 si $t4 ≥ $t3
        beq  $t0, $zero,exit2   # ir a exit2 si $t4 ≥ $t3
             ...
             (Cuerpo del segundo lazo for)
             ...
        addi $s1, $s1, -1   # j -= 1
        j    for2tst        # salta al test del lazo interno
exit2:
```

La llamada de procedimiento en ordenación (`sort`)

El siguiente paso es el cuerpo del segundo lazo *for*:

```
swap(v,j);
```

La llamada al procedimiento de intercambio (*swap*) es bastante fácil:

```
jal      swap
```

El paso de parámetros en ordenación (sort)

El problema aparece cuando deseamos pasar parámetros porque el procedimiento de ordenación necesita los valores en los registros $a0 y $a1 y el procedimiento de intercambio necesita tener sus parámetros puestos en esos mismos registros. Una solución es copiar los parámetros para la ordenación (sort) en otros registros al principio del procedimiento, de modo que los registros $a0 y $a1 estén disponibles para la llamada del intercambio (swap). (Esta copia es más rápida que guardar y restaurar de la pila). Primero se copia $a0 y $a1 en $s2 y $s3 durante el procedimiento:

```
move     $s2, $a0   # copia el parámetro $a0 en $s2
move     $s3, $a1   # copia el parámetrr $a1 en $s3
```

Luego se pasan los parámetros al procedimiento de intercambio (*swap*) con las dos instrucciones siguientes

```
move     $a0, $s2   # el primer parámetro de intercambio
                    # (swap) es v
move     $a1, $s1   # el segundo parámetro de intercambio
                    # (swap) es j
```

Preservación de registros en ordenación (sort)

El único código que nos falta tratar es guardar y restaurar los registros. Claramente, debemos guardar la dirección de retorno en el registro $ra, puesto que la ordenación (sort) es un procedimiento y es llamado como tal. El procedimiento de ordenación también utiliza los registros guardados $s0, $s1, $s2 y $s3, así que deben ser guardados. El prólogo del procedimiento de ordenación es el siguiente:

```
addi  $sp,$sp,-20  # reserva espacio en la pila para 5 reg.
sw    $ra,16($sp)  # guardar $ra en la pila
sw    $s3,12($sp)  # guardar $s3 en la pila
sw    $s2, 8($sp)  # guardar $s2 en la pila
sw    $s1, 4($sp)  # guardar $s1 en la pila
sw    $s0, 0($sp)  # guardar $s0 en la pila
```

El final del procedimiento invierte simplemente todas estas instrucciones, y después añade un jr para retornar.

El procedimiento de ordenación (sort) completo

Ahora ponemos todas las piezas juntas en la figura 2.27, teniendo cuidado de reemplazar las referencias a los registros $a0 y $a1 en los lazos *for* por referencias a los registros $s2 y $s3. De nuevo, para hacer el código más fácil de seguir, identificamos cada bloque del código con su propósito en el procedimiento. En este ejemplo, 9 líneas del procedimiento de ordenación (sort) en C se convirtieron en 35 líneas de lenguaje ensamblador MIPS.

Extensión: Una optimización que funcionaría con este ejemplo es el procedimiento en línea (*inlining*). En vez de pasar argumentos en parámetros y de invocar el código con una instrucción `jal`, el compilador copiaría el código del cuerpo del procedimiento de intercambio (`swap`) donde la llamada al procedimiento de intercambio apareciera en el código. La inclusión (*inlining*) evitaría cuatro instrucciones en este ejemplo. La desventaja de la optimización por inclusión (*inlining*) es que el código compilado sería más grande si el procedimiento en línea se llamara desde distintos lugares. Este tipo de extensión del código podría suponer menores prestaciones si incrementara la tasa de fallos de la memoria caché; véase el capítulo 5.

Guardar registros

```
sort:   addi    $sp,$sp, -20        # reserva espacio en la pila para 5 registros
        sw      $ra, 16($sp)        # guardar $ra en la pila
        sw      $s3,12($sp)         # guardar $s3 en la pila
        sw      $s2, 8($sp)         # guardar $s2 en la pila
        sw      $s1, 4($sp)         # guardar $s1 en la pila
        sw      $s0, 0($sp)         # guardar $s0 en la pila
```

Cuerpo del procedimiento

Copiar parámetros		move $s2, $a0	# copia el parámetro $a0 en $s2 (g uardar $a0)
		move $s3, $a1	# copia el parámetro $a1 en $s3 (guardar $a1)
Lazo exterior		move $s0, $zero	# i = 0
	for1tst:	slt $t0, $s0, $s3	# reg $t0 = 0 si $s0 ≥ $s3 (i ≥ n)
		beq $t0, $zero, exit1	# ir a exit1 si $s0 ≥ $s3 (i ≥ n)
Lazo interior		addi $s1, $s0, -1	# j = i - 1
	for2tst:	slti $t0, $s1, 0	# reg $t0 = 1 si $s1 < 0 (j < 0)
		bne $t0, $zero, exit2	# ir a exit2 si $s1 < 0 (j < 0)
		sll $t1, $s1, 2	# reg $t1 = j * 4
		add $t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw $t3, 0($t2)	# reg $t3 = v[j]
	lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt $t0, $t4, $t3	# reg $t0 = 0 si $t4 ≥ $t3
		beq $t0, $zero, exit2	# ir a exit2 si $t4 ≥ $t3
Paso de parámetros y llamada		move $a0, $s2	# el primer parámetro de intercambio (swap) es v (antiguo $a0)
		move $a1, $s1	# el segundo parámetro de intercambio (swap) es j
		jal swap	# swap código mostrado en la figura 2.25
Lazo interior		addi $s1, $s1, -1	# j -= 1
		j for2tst	# salta al test del lazo interior
Lazo exterior	exit2:	addi s0, $s0, 1 # i += 1	
		j for1tst	# salta al test del lazo exterior

Restauración de registros

```
exit1:lw        $s0, 0($sp)         # recupera $s0 de la pila
      lw        $s1, 4($sp)         # recupera $s1 de la pila
      lw        $s2, 8($sp)         # recupera $s2 de la pila
      lw        $s3,12($sp)         # recupera $s3 de la pila
      lw        $ra,16($sp)         # recupera $ra de la pila
addi            $sp,$sp, 20         # recupera el puntero de la pila
```

Retorno del procedimiento

```
        jr      $ra                 # retorno a la rutina llamadora
```

FIGURA 2.27 Versión ensamblada MIPS del procedimiento de ordenación (`sort`) de la figura 2.26.

Comprender las prestaciones de los programas

La figura 2.28 muestra el impacto de la optimización del compilador en las prestaciones, en el tiempo de compilación, en los ciclos de reloj, en el recuento de instrucciones y en el CPI del procedimiento de ordenación. Obsérvese que el código no optimizado tiene mejor CPI y la optimización O1 tiene menor número de instrucciones, pero O3 es la más rápida, y nos recuerda que el tiempo es la única medida exacta de las prestaciones del programa.

La figura 2.29 compara el impacto de los lenguajes de programación, compilación frente a interpretación, y de los algoritmos en las prestaciones de los procedimientos de ordenación. La cuarta columna muestra que el programa de C no optimizado es 8.3 veces más rápido que el código de Java interpretado para la ordenación por burbuja (*bubble sort*). El uso del compilador *Just-In-Time* de Java hace que el código Java sea 2.1 veces más rápido que el C no optimizado, y alrededor de un 1.13 más que el código de optimización de C más alto. (La **sección 2.15** en el CD da más detalles de la interpretación *versus* la compilación de Java y del código para la ordenación por burbuja Java y MIPS). Los ratios no están tan cerca para Quicksort en la columna 5, probablemente porque es más difícil amortizar el coste de la compilación en tiempo de ejecución (*runtime*) en el tiempo de ejecución más corto. La última columna muestra el efecto de un algoritmo mejor, que proporciona un incremento de las prestaciones de tres órdenes de magnitud para una ordenación de 100 000 elementos. Incluso comparando el Java interpretado en la columna 5 con el compilador de C en la optimización de grado más alto en la columna 4, Quicksort gana a la ordenación por burbuja (*bubble sort*) por un factor de 50 (0.05 × 2468, o 123 contra 2.41).

Extensión: Los compiladores MIPS siempre reservan espacio en la pila para los argumentos en el caso de que necesiten ser almacenados, así que en realidad siempre disminuyen `$sp` en 16 para reservar espacio para los cuatro registros de argumentos (16 bytes). Una razón es que C proporciona una opción `vararg` que permite a un puntero tomar, por ejemplo, el tercer argumento de un procedimiento. Cuando el compilador encuentra el poco frecuente `vararg`, copia los cuatro registros de argumentos en la pila en las cuatro posiciones reservadas.

Optimización gcc	Prestaciones relativas	Ciclos de reloj (millones)	Número de instrucciones (millones)	CPI
ninguna	1.00	158 615	114 938	1.38
01 (media)	2.37	66 990	37 470	1.79
02 (completa)	2.38	66 521	39 993	1.66
03 (integración del procedimiento)	2.41	65 747	44 993	1.46

FIGURA 2.28 Comparación de las prestaciones, del número de instrucciones y del CPI usando las optimizaciones del compilador para la ordenación por el método de la burbuja (*bubble sort*). Los programas ordenaron 100 000 palabras con la tabla iniciada con valores al azar. Esos programas fueron ejecutados en un Pentium 4 con una frecuencia de reloj de 3.06 GHz y un bus del sistema de 533 MHz con 2 GB de memoria SDRAM PC2100 DDR. Se utilizó la versión 2.4.20 de Linux.

Lenguaje	Método de ejecución	Optimización	Prestaciones relativas Ordenación por burbuja	Prestaciones relativas Quicksort	Ventaja de Quicksort frente a la ordenación por burbuja
C	compilador	ninguna	1.00	1.00	2468
	compilador	01	2.37	1.50	1562
	compilador	02	2.38	1.50	1555
	compilador	03	2.41	1.91	1955
Java	intérprete	—	0.12	0.05	1050
	compilador *Jus-In-Time*	—	2.13	0.29	338

FIGURA 2.29 Prestaciones de dos algoritmos de ordenación en C y en Java usando interpretación y optimización de compiladores en relación con la versión de C no optimizada. La última columna muestra la ventaja en velocidad de Quicksort sobre la ordenación por burbuja para cada lenguaje y opción de ejecución. Estos programas fueron ejecutados en el mismo sistema que el utilizado en la figura 2.28. La JVM es la versión 1.3.1 de Sun, y el JIT es la versión 1.3.1 de Hotspot de Sun.

2.14 Tablas frente a punteros

Un desafío típico para cualquier programador novel es comprender los punteros. La comparación del código ensamblador que utiliza tablas y los índices a tablas con el código ensamblador que utiliza punteros ofrece elementos de comprensión de los punteros. Esta sección muestra las versiones de dos procedimientos en C y en lenguaje ensamblador MIPS para inicializar una secuencia de palabras en memoria: una utilizando índices de tablas y otra utilizando punteros. La figura 2.30 muestra los dos procedimientos en C.

El propósito de esta sección es mostrar cómo se corresponden los punteros con instrucciones MIPS, y no promover un estilo de programación anticuado. Veremos el impacto de la optimización de un compilador moderno en estos dos procedimientos al final de la sección.

Versión de iniciar (`clear`) con tablas

Comencemos con la versión con tablas, `clear1`, centrándonos en el cuerpo del lazo e ignorando el código de enlace de procedimientos. Suponemos que los dos parámetros *tabla* y *tamaño* se encuentran en los registros `$a0` y `$a1`, y que `i` está asignado al registro `$t0`.

La iniciación de `i`, la primera parte del lazo *for*, es directa:

```
        move    $t0,$zero    # i = 0 (registro $t0 = 0)
```

Para iniciar `tabla[i]` a 0, primero debemos conseguir su dirección. Se comienza multiplicando `i` por 4 para conseguir la dirección del byte:

```
loop1:sll   $t1,$t0,2    # $t1 = i * 4
```

Puesto que la dirección inicial de la tabla está en un registro, debemos añadirlo al índice para conseguir la dirección de `tabla[i]` usando una instrucción de suma:

```
        add     $t2,$a0,$t1  # $t2 = dirección de tabla[i]
```

Finalmente, podemos almacenar 0 en esa dirección:

```
clear1(int tabla[], int tamaño)
{
 int i;
 for (i = 0; i < tamaño; i += 1)
       tabla[i] = 0;
}

clear2(int *tabla, int tamaño)
{
 int *p;
 for (p = &tabla[0]; p < &tabla[tamaño]; p = p + 1)
       *p = 0;
}
```

FIGURA 2.30 Dos procedimientos de C para iniciar una tabla a ceros. `Clear1` utiliza índices, mientras que `clear2` utiliza punteros. El segundo procedimiento necesita una cierta explicación para los que no están familiarizados con `C`. La dirección de una variable se indica con `&`, y la referencia al objeto apuntado por un puntero se indica con `*`. Las sentencias dicen que `tabla` y `p` son punteros a números enteros. La primera parte del lazo *for* en `clear2` asigna la dirección del primer elemento de `tabla` al puntero `p`. La segunda parte del lazo *for* comprueba si el puntero está señalando más allá del último elemento de tabla. El incremento de un puntero en uno, en la última parte secuencial del lazo *for*, significa mover el puntero al siguiente objeto secuencial teniendo en cuenta su tamaño declarado. Puesto que `p` es un puntero a números enteros, el compilador generará instrucciones MIPS para incrementar `p` en cuatro, el número de bytes de un número entero MIPS. La asignación en el lazo coloca un 0 en el objeto señalado por `p`.

```
        sw      $zero, 0($t2)  # tabla[i] = 0
```

Esta instrucción está al final del cuerpo del lazo. Por tanto, el paso siguiente es incrementar `i`:

```
        addi    $t0,$t0,1    # i = i + 1
```

El test del lazo comprueba si `i` es menor que `tamaño`:

```
        slt   $t3,$t0,$a1      # $t3 = (i < tamaño)
        bne   $t3,$zero,loop1  # si (i < tamaño) ir a loop1
```

Ahora hemos visto todas las piezas del procedimiento. Éste es el código MIPS para inicializar una tabla usando índices:

```
        move  $t0,$zero       # i = 0
loop1:  sll   $t1,$t0,2       # $t1 = i * 4
        add   $t2,$a0,$t1     # $t2 = dirección de tabla[i]
        sw    $zero, 0($t2)   # tabla[i] = 0
        addi  $t0,$t0,1       # i = i + 1
        slt   $t3,$t0,$a1     # $t3 = (i < tamaño)
        bne   $t3,$zero,loop1 # si (i < tamaño) ir a loop1
```

(Este código es válido mientras *tamaño* sea mayor que 0; ANSI C requiere comprobar el tamaño antes del lazo, pero aquí omitiremos esta cuestión.)

Versión de iniciar (`clear`) con punteros

El segundo procedimiento, que utiliza punteros, asigna los dos parámetros `tabla` y `tamaño` a los registros `$a0` y `$a1` y coloca `p` en el registro `$t0`. El código para el segundo procedimiento comienza asignando el indicador `p` a la dirección del primer elemento de la tabla:

```
        move  $t0,$a0       # p = dirección de tabla[0]
```

El código siguiente es el cuerpo del lazo *for*, que almacena simplemente 0 en la posición `p`:

```
loop2:sw    $zero,0($t0)  # Memoria[p] = 0
```

Esta instrucción implementa el cuerpo del lazo, así que el código siguiente es el incremento de la iteración, que cambia `p` para apuntar a la palabra siguiente:

```
        addi  $t0,$t0,4    # p = p + 4
```

El incremento de un puntero en 1 significa mover el puntero al siguiente objeto secuencial en C. Puesto que `p` es un puntero a números enteros, que utilizan 4 bytes, el compilador incrementa `p` en 4.

El test del lazo es el siguiente. El primer paso es calcular la dirección del último elemento de `tabla`. Se comienza multiplicando `tamaño` por 4 para conseguir su dirección byte:

```
        sll   $t1,$a1,2    # $t1 = tamaño * 4
```

y entonces sumamos el producto a la dirección inicial de la tabla para conseguir la dirección de la primera palabra *después* de la tabla:

```
        add   $t2,$a0,$t1  # $t2 = direccion de tabla[tamaño]
```

El test del lazo es simplemente comprobar si `p` es menor que el último elemento de la tabla:

```
      slt  $t3,$t0,$t2      # $t3 = (p<&tabla[tamaño])
      bne  $t3,$zero,loop2  # si (p<&tabla[tamaño]) ir a loop2
```

Con todas las piezas terminadas, podemos mostrar una versión completa del código con punteros para inicializar una tabla a cero:

```
      move $t0,$a0          # p = dirección de tabla[0]
    loop2:sw$zero,0($t0)    # Memoria[p] = 0
      addi $t0,$t0,4        # p = p + 4
      add  $t1,$a1,$a1      # $t1 = tamaño * 2
      add  $t1,$t1,$t1      # $t1 = tamaño * 4
      add  $t2,$a0,$t1      # $t2 = dirección de tabla[tamaño]
      slt  $t3,$t0,$t2      # $t3 = (p<&tabla[tamaño])
      bne  $t3,$zero,loop2  # si (p<&tabla[tamaño]) ir a loop2
```

Como en el primer ejemplo, este código supone que `tamaño` es mayor que 0.

Obsérvese que este programa calcula la dirección del final de la tabla en cada iteración del lazo, a pesar de que no cambia. Una versión más rápida del código traslada este cálculo fuera del lazo:

```
      move   $t0,$a0          # p = direcciones de tabla[0]
      sll    $t1,$a1,2        # $t1 = tamaño * 4
      add    $t2,$a0,$t1      # $t2 = dirección de tabla[tamaño]
loop2:sw     $zero,0($t0)     # Memoria[p] = 0
      addi   $t0,$t0,4        # p = p + 4
      slt    $t3,$t0,$t2      # $t3 = (p<&tabla[tamaño])
      bne    $t3,$zero,loop2  # si (p<&tabla[tamaño]) ir a loop2
```

Comparación de las dos versiones de iniciar a cero (`clear`)

Comparar las dos secuencias de código, una al lado de la otra, ilustra la diferencia entre índices de tablas y de punteros (se resaltan los cambios introducidos por la versión con punteros):

```
      move  $t0,$zero     # i = 0                      move  $t0,$a0        # p = & tabla[0]
loop1:sll   $t1,$t0,2     # $t1 = i * 4                sll   $t1,$a1,2      # $t1 = tamaño * 4
      add   $t2,$a0,$t1   # $t2 = &tabla[i]            add   $t2,$a0,$t1    # $t2 = &tabla[tamaño]
      sw    $zero, 0($t2) # tabla[i] = 0       loop2: sw    $zero,0($t0)   # Memoria[p] = 0
      addi  $t0,$t0,1     # i = i + 1                  addi  $t0,$t0,4      # p = p + 4
      slt   $t3,$t0,$a1   # $t3 = (i < tamaño)         slt   $t3,$t0,$t2    # $t3=(p<&tabla[tamaño])
      bne   $t3,$zero,loop1# si () ir a loop1          bne   $t3,$zero,loop2# si () ir a loop2
```

La versión de la izquierda debe tener la "multiplicación" y la suma dentro del lazo porque se incrementa `i` y cada dirección se debe recalcular para el nuevo índice; la versión con punteros a memoria de la derecha incrementa el puntero `p` directamente. La versión con punteros mueve estas operaciones fuera del lazo, reduciendo así las instrucciones ejecutadas por iteración de 6 a 4. Esta optimización manual corresponde a la optimización del compilador de la reducción de la intensidad (desplazar en vez de multiplicar) y de la eliminación de variables de inducción (que elimina cálculos de dirección de la tabla dentro de lazos). La **sección 2.15** en el CD describe estas dos optimizaciones y muchas otras.

Extensión: El compilador de C añadiría una prueba para estar seguro de que `tamaño` es mayor que 0. Una forma sería añadir un salto justo antes de la instrucción de la primera instrucción del lazo hacia la instrucción `slt`.

Se solía enseñar a utilizar punteros en C para conseguir mayor eficiencia que la disponible con tablas: "Utilicen los punteros, aunque no puedan comprender el código". Los compiladores de optimización modernos pueden producir código igual de bueno para la versión de tablas. La mayoría de los programadores prefieren hoy que el compilador haga la parte pesada.

Comprender las prestaciones de los programas

Perspectiva histórica y lecturas recomendadas

En esta sección se da una visión general de como trabaja el compilador de C y de como se ejecuta java. La comprensión de la tecnología de los compiladores es crítica para entender las prestaciones de un computador, puesto que el compilador afecta de forma significativa a las prestaciones. Téngase en cuenta que la construcción de un compilador es materia para un curso de uno o dos semestres, por lo tanto, nuestra introducción no puede ir más allá de los aspectos básicos.

La segunda parte de esta sección está destinada a aquellos lectores interesados en conocer como se ejecuta un **lenguaje orientado a objetos**, como Java, en una arquitectura MIPS. Muestra las instrucciones de Java usadas para la interpretación y los códigos MIPS para la versión Java de algunos de segmentos en C de secciones anteriores, incluyendo el Ordenamiento de Burbujas. Cubre la máquina virtual de Java y los compiladores JIT.

Lenguaje orientado a objetos: lenguaje de programación que está orientado a los objetos en vez de a las acciones. O dicho de otra forma, datos frente a lógica.

El resto de esta sección está en el CD.

2.16 Caso real: instrucciones ARM

La arquitectura del conjunto de instrucciones para dispositivos empotrados más popular es ARM, con más de tres mil millones de dispositivos por año utilizando esta arquitectura. Desarrollada originalmente para el procesador Acorn RISC Machine, que después se llamó Advanced RISC Machine, ARM se lanzó el mismo año que MIPS y siguen filosofías similares. La figura 2.31 muestra estas similitudes. La principal diferencia entre MIPS y ARM es que MIPS tiene más registros y ARM tiene más modos de direccionamiento.

El núcleo de instrucciones aritmético-lógicas y de transferencia de datos es muy similar en MIPS y en ARM, como se muestra en la figura 2.32.

Modos de direccionamiento

Los modos de direccionamiento de datos soportados por ARM se muestran en la figura 2.33. A diferencia de lo que ocurre con el MIPS, en ARM no hay un registro reservado con el valor 0. En ARM hay 9 modos de direccionamiento de datos

	ARM	MIPS
Fecha de lanzamiento	1985	1985
Tamaño de la instrucción (bits)	32	32
Espacio de direcciones (tamaño, modelo)	32 bits, plano	32 bits, plano
Alineamiento de datos	Alineado	Alineado
Modos de direccionamiento	9	3
Registros de enteros (número, modelo, tamaño)	15 GPR x 32 bits	31 GPR x 32 bits
E/S	Proyectada (*mapped*) en memoria	Proyectada (*mapped*) en memoria

FIGURA 2.31 Similitudes entre los conjuntos de instrucciones de ARM y MIPS.

	Nombre de la instrucción	ARM	MIPS
Registro-registro	Suma	add	addu, addlu
	Suma (excepción si desbordamiento)	adds; swivs	add
	Resta	sub	subu
	Resta (excepción si desbordamiento)	subs; swivs	sub
	Multiplicación	mul	mult, multu
	División	–	div, divu
	And	and	and
	Or	orr	or
	Or exclusiva	eor	xor
	Carga parte más significativa del registro	_	lui
	Desplazamiento lógico a la izquierda	lsl[1]	sllv, sll
	Desplazamiento lógico a la derecha	lsr[1]	srlv, srl
	Desplazamiento aritmético a la derecha	asr[1]	srav, sra
	Comparación	cmp, cmn, tst, teq	slt/i, slt/iu
Transferencia de datos	Carga de un byte con signo	ldrsb	lb
	Carga de un byte sin signo	ldrb	lbu
	Carga de media palabra con signo	ldrsh	lh
	Carga de media palabra sin signo	ldrh	lhu
	Carga de una palabra	ldr	lw
	Almacenamiento de un byte	strb	sb
	Almacenamiento de media palabra	strh	sh
	Almacenamiento de una palabra	str	sw
	Lectura, escritura de registros especiales	mrs, msr	move
	Intercambio atómico	swp, swpb	ll;sc

FIGURA 2.32 Instrucciones registro-registro y de transferencia de datos de ARM equivalentes a instrucciones MIPS. El guión significa que la operación no está disponible en esa arquitectura o que ha sido sintetizada en una pocas instrucciones. Si hay varias instrucciones que pueden reemplazar al núcleo MIPS, éstas se separan por comas. ARM incluye desplazamientos como parte de cualquier instrucción de operación de datos, por lo tanto, los desplazamientos con superíndice 1 son simplemente una variación de una instrucciones de transferencia, por ejemplo `lsr`[1]. Observa que ARM no tiene instrucciones de división.

(véase figura 2.18), algunos de los cuales incluyen cálculos bastante complejos, mientras que en MIPS hay sólo tres. Por ejemplo, uno de los modos de direccionamiento de ARM puede hacer un desplazamiento de cualquier cantidad del contenido de un registro, sumarlo al contenido de otro registro para formar la dirección y actualizar un registro con esta dirección.

Modo de direccionamiento	ARM v.4	MIPS
Operando en registro	X	X
Operando inmediato	X	X
Registro + desplazamiento (desplazamiento o basado)	X	X
Registro + registro (indexado)	X	–
Registro + registro escalado (escalado)	X	–
Registro + desplazamiento y actualización de registro	X	–
Registro + registro y actualización de registro	X	–
Autoincremento, autodecremento	X	–
Dato relativo al PC	X	–

FIGURA 2.33 Resumen de modos de direccionamiento de datos. ARM tiene modos de direccionamiento indirecto a registro y registro + desplazamiento diferenciados, en lugar de poner un desplazamiento igual a 0 en este último. Para disponer de un mayor rango de direccionamiento, ARM desplaza el desplazamiento a la izquierda 1 o 2 bits cuando el tamaño del dato es media palabra o una palabra.

Comparación y salto condicional

En MIPS se utiliza el contenido de registros para la evaluación de los saltos condicionales. Por el contrario, en ARM se utilizan los tradicionales cuatro bits de código de condición almacenados en la palabra de estado: *negativo*, *cero*, *acarreo* (*carry*) y *desbordamiento* (*overflow*). Estos bits pueden ser activados con cualquier instrucción aritmética o lógica y, a diferencia de arquitecturas anteriores, esta activación es opcional para cada instrucción. Esta selección explícita hace que la implementación segmentada tenga menos problemas. En los saltos condicionales de ARM se utilizan los códigos de condición para determinar todas la posibles relaciones con y sin signo.

La instrucción CMP resta dos operandos y el resultado determina el valor de los bits del código de condición. La instrucción Compara negativo (CMN) *suma* dos operandos y el resultado determina el valor del código de condición. La instrucción TST hace el AND lógico entre los dos operandos y pone el valor de los bits del código de condición excepto del bit de desbordamiento; de modo similar, TEQ realiza la operación OR exclusiva y fija el valor de los tres primeros bits del código de condición.

Las instrucciones de ARM incorporan una característica poco usual: todas instrucciones tienen la opción de ejecutarse condicionalmente dependiendo de los bits del código de condición. Cada instrucción comienza con un campo de cuatro bits que indica, dependiendo del código de condición, si actúa como una instrucción de no operación (nop) o como una instrucción real. De este modo, los saltos condicionales pueden ser considerados como una ejecución condicional de una instrucción de salto incondicional. La ejecución condicional permite eliminar las instrucciones de salto que se introducen para saltar únicamente una instrucción; es más eficiente desde el punto de vista del tamaño del código y tiempo de ejecución simplemente ejecutar condicionalmente una instrucción.

Los formatos de las instrucciones ARM y MIPS se muestran en la figura 2.34. Las principales diferencias son que cada instrucción ARM incorpora un campo de ejecución condicional de 4 bits y que el campo de registro es más reducido, porque ARM tiene la mitad de los registros de MIPS.

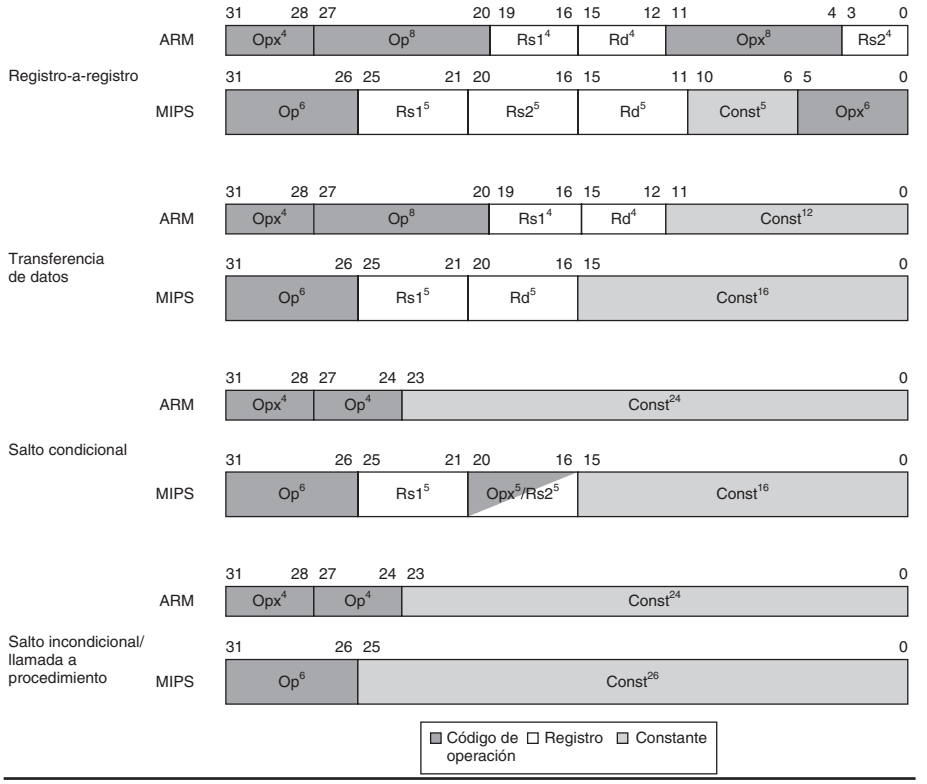


FIGURA 2.34 Formatos de instrucciones de ARM y MIPS. Las diferencias están en el número de registros de la arquitectura, 16 o 32.

Características únicas de ARM

Las instrucciones aritmético-lógicas de ARM que no tienen equivalente en MIPS se indican en la figura 2.35. Como ARM no tiene un registro especial con el valor 0, dispone de códigos de operación diferenciados para algunas operaciones que en MIPS se pueden hacer con `$zero`. Además, en ARM se dispone de soporte para aritmética multipalabra.

El campo inmediato de 12 bits de ARM tiene una interpretación novedosa. Los ocho bits menos significativos se extienden con ceros hasta 32 bits, y después se rotan a la derecha el número de bits especificado en los primeros cuatro bits del campo multiplicado por dos. La ventaja de este esquema es que se pueden representar todas las potencias de dos en una palabra de 32 bits. Sería interesante analizar si esta partición realmente nos permite representar más valores inmediatos que un campo convencional de 12 bits.

El desplazamiento de operandos no está limitado únicamente a los inmediatos. Se dispone de la opción de desplazar el contenido del segundo registro de todas las operaciones aritméticas y lógicas antes de realizar la operación. Los desplazamientos pueden ser desplazamiento lógico a la izquierda, desplazamiento lógico a la derecha, desplazamiento aritmético a la derecha y rotación a la derecha.

Nombre	Definición	ARM v.4	MIPS
Carga inmediata	Rd = Imm	mov	addi, $0,
Negación	Rd = ~(Rs1)	mvn	nor, $0,
Mover	Rd = Rs1	mov	or, $0,
Rotación a la derecha	Rd = Rs i>> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And negado	Rd = Rs1 & ~(Rs2)	blc	
Resta negada	Rd = Rs2 - Rs1	rsb, rsc	
Apoyo para suma entera multipalabra	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	_
Apoyo para resta entera multipalabra	CarryOut, Rd = Rd - Rs1 + OldCarryOut	sbcs	_

FIGURA 2.35 Instrucciones aritméticas y lógicas de ARM no disponibles en MIPS.

En ARM también se dispone de instrucciones para almacenar grupos de registros, llamados *cargas y almacenamiento por bloques* (*block load*, *block store*). Utilizando como control una máscara de 16 bits incluida en la instrucción, cualesquiera de los 16 registros pueden cargarse o almacenarse en memoria en una única instrucción. Estas instrucciones puede utilizarse para salvar y restaurar registros en las llamadas y retornos de un procedimiento. Sin embargo, el uso actual más importante de estas instrucciones es la copia de bloques de memoria.

2.17 Casos reales: instrucciones x86

La belleza está en los ojos del que mira.

Margaret Wolfe Hungerford, *Molly Bawn*, 1877

Los diseñadores del repertorio de instrucciones proporcionan a veces operaciones más potentes que las que aparecen en ARM o MIPS. La meta es generalmente reducir el número de instrucciones ejecutadas por un programa. El peligro está en que esta reducción puede hacerse a costa de la simplicidad, y se aumenta el tiempo que tardan los programas en ejecutarse porque las instrucciones son más lentas. Esta lentitud puede ser el resultado de un ciclo de reloj más lento o de que se requiere más ciclos de reloj que una secuencia más simple.

El camino hacia la mayor complejidad de las operaciones está así plagado de peligros. Para evitar estos problemas los diseñadores se han decantado por instrucciones más simples. La sección 2.18 muestra las trampas de la complejidad.

Evolución del Intel x86

ARM y MIPS fueron el resultado de la visión que tuvieron en 1985 un pequeño grupo de personas; las piezas de esta arquitectura encajan fácilmente unas con otras y la arquitectura entera se puede describir de forma sucinta. Pero este no es el caso del x86; éste es el producto de varios grupos independientes que desarrollaron la arquitectura a lo largo de casi 30 años, agregando nuevas características al repertorio de instrucciones original como el que añade ropa a una maleta. Aquí están los hitos más importantes del x86:

Registro de propósito general (GPR): registro que se puede utilizar para las direcciones o para los datos con virtualmente cualquier instrucción.

- **1978:** La arquitectura del Intel 8086 fue anunciada como una extensión compatible en lenguaje ensamblador del exitoso Intel 8080, un microprocesador de 8 bits. El 8086 es una arquitectura de 16 bits, con todos los registros internos de un tamaño de 16 bits. A diferencia de MIPS, los registros tienen usos específicos, y por tanto los 8086 no está considerados una arquitectura de **registro de propósito general**.
- **1980:** Se anuncia el coprocesador de punto flotante Intel 8087. Esta arquitectura amplía los 8086 con cerca de 60 instrucciones de punto flotante. En lugar de usar los registros, se utiliza la pila (véase la **sección 2.20** en el CD y la sección 3.7).
- **1982**: El 80286 amplía la arquitectura 8086 aumentando el espacio de direcciones a 24 bits, creando una distribución de memoria y un modelo de protección elaborado (véase el capítulo 5) y agregando algunas instrucciones más para completar el repertorio de instrucciones y para manipular el modelo de protección.
- **1985**: Los 80386 ampliaron la arquitectura 80286 a 32 bits. Además de la arquitectura de 32 bits con registros de 32 bits y un espacio de dirección de 32 bits, el 80386 añade nuevos modos de direccionamiento y nuevas operaciones. Las instrucciones añadidas hacen de los 80386 casi una máquina de registros de propósito general. El 80386 también añade soporte de paginación además del direccionamiento segmentado (véase el capítulo 5). Como los 80286, los 80386 tienen un modo para ejecutar programas 8086 sin hacer ningún cambio.
- **1989-95**: Los subsiguientes 80486 de 1989, el Pentium en 1992 y el Pentium Pro en 1995 tenían el propósito de aumentar las prestaciones, con solamente cuatro instrucciones añadidas al repertorio de instrucciones visible por el usuario: tres para ayudar con el multiproceso (capítulo 7) y una instrucción de movimiento condicional.
- **1997**: Después de lanzar el Pentium y el Pentium Pro, Intel anunció que ampliaría las arquitecturas Pentium y Pentium Pro con MMX (eXtensiones Multi Media). Este nuevo repertorio de 57 instrucciones utilizaba la pila del punto flotante para acelerar aplicaciones multimedia y de comunicaciones. Las instrucciones MMX funcionan típicamente con múltiples datos pequeños a la vez, siguiendo la tradición de las arquitecturas de flujo único de instrucciones-múltiples flujos de datos (SIMD) (véase el capítulo 7). El Pentium II no introdujo ninguna instrucción nueva.
- **1999**: Intel añadió otras 70 instrucciones, etiquetadas SSE (extensiones de flujo SIMD, *Streaming SIMD Extensions*) como parte del Pentium III. Los principales cambios eran añadir ocho registros separados, doblar su anchura a 128 bits y agregar un tipo de datos de punto flotante de precisión simple. Por tanto, cuatro operaciones de punto flotante de 32 bits podían realizarse en paralelo. Para mejorar las prestaciones de la memoria, SSE incluía instrucciones de prebúsqueda de cache más las instrucciones de almacenamiento de flujo que puentean las cache y escriben directamente en la memoria.
- **2001**: Intel agregó otras 144 instrucciones, esta vez etiquetadas SSE2. Este nuevo tipo de datos es de aritmética de doble precisión, que permite pares de operaciones de punto flotante de 64-bit en paralelo. Casi todas estas

144 instrucciones eran versiones de las instrucciones existentes de MMX y de SSE que funcionan en 64 bits de datos en paralelo. Este cambio no sólo permitía más operaciones multimedia, también daba al compilador un objetivo diferente para las operaciones de punto flotante que la arquitectura de pila única. Los compiladores podían elegir utilizar los ocho registros de SSE como registros de punto flotante del tipo de los encontrados en otros computadores. Este cambio estimuló las prestaciones del punto flotante en el Pentium 4, el primer microprocesador en incluir las instrucciones SSE2.

- **2003**: Otra compañía además de Intel aumentó la arquitectura x86. AMD anunció un conjunto de extensiones arquitectónicas para aumentar el espacio de dirección de 32 bits a 64. De forma similar a la transición del espacio de dirección de 16 a 32 bits en 1985 con los 80386, AMD64 amplía todos los registros a 64 bits. También aumenta el número de registros a 16 y el número de registros de 128 bits SSE a 16. El cambio principal del ISA (arquitectura del repertorio de instrucciones) viene de añadir un nuevo modo llamado el *modo largo (long mode)*, que redefine la ejecución de todas las instrucciones x86 con direcciones y datos de 64 bits. Para abordar este número más grande de registros, agrega un nuevo prefijo a las instrucciones. Dependiendo de cómo contemos, el modo largo también añade de 4 a 10 nuevas instrucciones y quita unas 27 viejas. El direccionamiento de datos relativo al PC es otra extensión. AMD64 todavía tiene un modo que es idéntico a x86 (*modo de herencia, legacy mode*) más un modo que restringe que el usuario programe en x86 pero que permite que los sistemas operativos utilicen AMD64 (*modo de compatibilidad*). Estos modos permiten una transición más agradable al direccionamiento de 64 bits que la arquitectura de HP/Intel IA-64.
- **2004**: Intel capitula y adopta AMD64, y la reetiqueta Tecnología de Memoria Extendida 64 (EM64T). La diferencia principal es que Intel añadió una instrucción atómica de 128 bits de comparación e intercambio que probablemente se debía haber incluido en AMD64. Al mismo tiempo, Intel anuncia otra generación de las extensiones multimedia. SSE3 añade 13 instrucciones para soportar aritmética compleja, operaciones con gráficos en tablas de estructuras, codificación vídeo, conversión punto flotante y la sincronización de hilos (*thread*) (véase la sección 2.11). AMD ofrecerá SSE3 en los próximos chips y agregará a AMD64 la mayoría de las instrucciones y la instrucción atómica que falta del intercambio para mantener compatibilidad binaria con Intel.
- **2006**: Intel añade otras 54 instrucciones, como extensión de su repertorio de instrucciones y lo llama SSE4. Esta extensión incluye variaciones como la suma de diferencias absolutas, producto escalar para tablas de estructuras, extensión de signo o extensión con ceros de datos con pocos bits a anchos mayores, etc. Se añadió también apoyo para máquinas virtuales (véase capítulo 5)
- **2007**: AMD anuncia 170 nuevas instrucciones que forman parte de la extensión SSE5. Se incluyen 46 instrucciones del repertorio de instrucciones básico pero con tres operandos, como en MIPS.
- **2008**: Intel anuncia el Advanced Vector Extension, que expande el ancho del registro SSE de 128 a 256 bits, redefine 250 instrucciones y añade 128 instrucciones nuevas.

Esta historia ilustra el impacto de la "esposas de oro" de la compatibilidad en el x86, pues la base del software existente en cada paso era demasiado importante para hacerla peligrar con cambios significativos en la arquitectura. Si se analiza la vida del x86, veremos que de media se ha añadido ¡una instrucción cada mes!

A pesar de los fallos artísticos del x86, tengamos presente que hay más copias de esta familia arquitectónica en computadores de sobremesa que de cualquier otra arquitectura, con un aumento de 250 millones por año. Sin embargo, estos altibajos han conducido a una arquitectura que es difícil de explicar e imposible de apreciar.

¡Prepárese para lo que está a punto de ver! No intente leer esta sección con el cuidado que necesitaría para escribir los programas x86, pues la intención es familiarizarnos con las ventajas e inconvenientes de la arquitectura de computadores de sobremesa más popular del mundo.

Más que mostrar entero el repertorio de instrucciones de 16 y de 32 bits, en esta sección nos centramos en el subconjunto de instrucciones de 32 bits que se originaron con los 80386, pues es esta parte de la arquitectura la que se usa en la actualidad. Comenzamos nuestra explicación con los registros y los modos de direccionamiento, seguimos por las operaciones de números enteros y concluimos con un examen de la codificación de las instrucciones.

Registros y modos de direccionamiento x86

Los registros del 80386 muestran la evolución del repertorio de instrucciones (figura 2.36). Los 80386 ampliaron todos los registros de 16 bits a 32 bits (excepto los registros de segmento) y añadieron el prefijo E a su nombre para distinguir la versión de 32 bits. Nos referiremos a ellos genéricamente como GPRs (registros de propósito general). El 80386 contiene solamente ocho GPRs. Esto significa que los programas MIPS y ARM pueden utilizar hasta cuatro veces y dos veces más registros, respectivamente.

Las instrucciones aritméticas, lógicas y de transferencia de datos son instrucciones de dos operandos que permiten las combinaciones mostradas en la figura 2.37. Hay dos diferencias importantes aquí. Las instrucciones aritméticas y lógicas del x86 deben tener un operando que actúe a la vez como fuente y destino; ARM y MIPS permiten registros separados para fuente y destino. Esta restricción añade más presión a los limitados registros, puesto que un registro fuente puede ser modificado. La segunda diferencia importante es que uno de los operandos puede estar en memoria. Así, virtualmente cualquier instrucción puede tener un operando en memoria, a diferencia de MIPS y de ARM.

Los modos de direccionamiento a memoria, descritos detalladamente más adelante, ofrecen dos tamaños de direcciones en las instrucciones. Éstos, que podemos llamar también *desplazamientos*, pueden ser de 8 o de 32 bits.

Aunque un operando de memoria puede utilizar cualquier modo de direccionamiento, hay restricciones sobre los *registros* que se pueden utilizar en un modo. La figura 2.38 muestra los modos de direccionamiento del x86 y qué GPRs no se pueden utilizar con ese modo, además de cómo se obtendría el mismo efecto usando instrucciones MIPS.

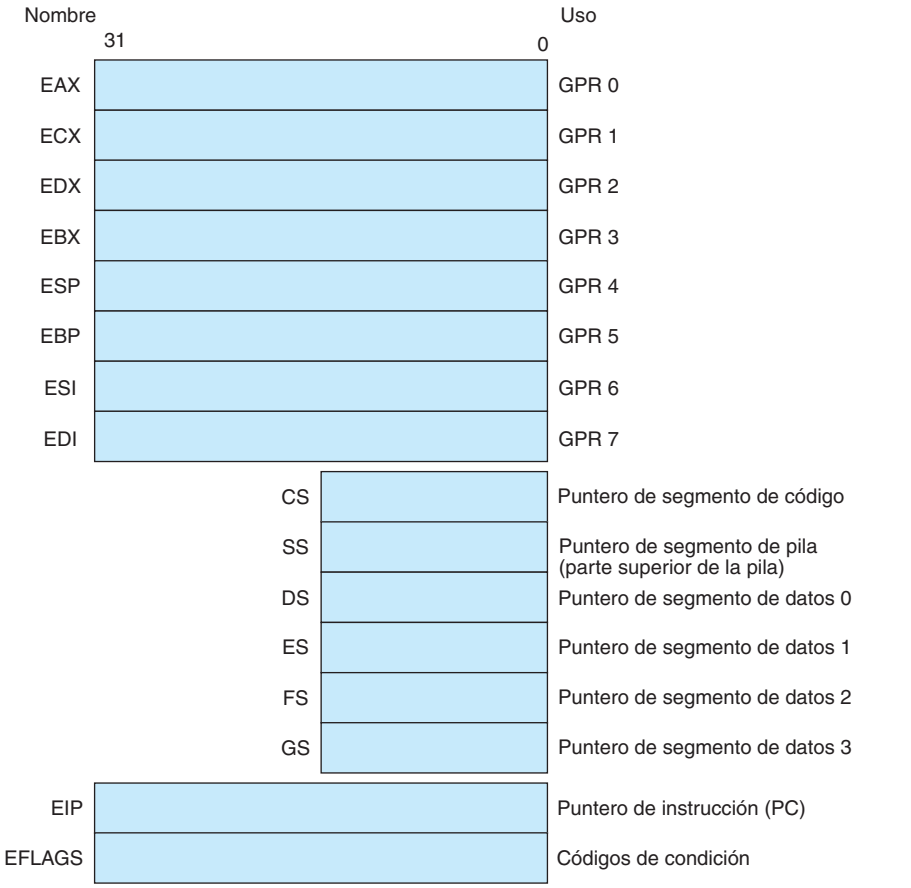


FIGURA 2.36 Conjunto de registros del 80386. A partir del 80386, los 8 registros de arriba fueron extendidos a 32 bits y se pudieron usar como registros de propósito general.

Tipos de operando fuente/destino	Segundo operando fuente
Registro	Registro
Registro	Immediato
Registro	Memoria
Memoria	Registro
Memoria	Immediato

FIGURA 2.37 Tipos de instrucciones para las instrucciones aritméticas, lógicas y de transferencia de datos. El x86 permite las combinaciones mostradas. La única restricción es la ausencia de un modo memoria-memoria. Los immediatos pueden ser de 8, 16 o 32 bits de longitud; un registro es cualquiera de los 14 registros principales de la figura 2.40 (no EIP o EFLAGS).

Modo	Descripción	Restricciones de registros	Equivalente MIPS
Registro indirecto	La dirección está en un registro	ni ESP ni EBP	`lw $s0,0($s1)`
Modo base con desplazamiento de 8 o de 32 bits	La dirección es un registro base más un desplazamiento	ni ESP ni EBP	`lw $s0,100($s1)` # ≤16-bit # desplazamiento

Base más índice escalado	La dirección es Base + (2^{escala} × Índice) donde escala tiene el valor 0, 1, 2 o 3.	Base: cualquier GPR Índice: no ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base más índice escalado con desplazamiento de 8 o de 32 bits	La dirección es Base + (2^{escala} × Índice) + desplazamiento donde escala tiene el valor 0, 1, 2 o 3.	Base: cualquier GPR Índice: no ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)` # ≤16-bit # desplazamiento

FIGURA 2.38 Modos de direccionamiento de 32 bits del x86 con restricciones de registros y con el código equivalente MIPS. El modo de direccionamiento base más índice escalado no se encuentra en MIPS o en ARM, se incluye para evitar la multiplicación por 4 (factor de escalar de 2) para convertir un índice guardado en un registro en una dirección byte (véase las figuras 2.25 y 2.27). Para datos de 16 bits se utiliza un factor de escala de 1, y para 64 bits de 3. El factor escala 0 significa que la dirección no está escalada. Si el desplazamiento es más largo de 16 bits en el segundo o el cuarto modo, entonces el modo equivalente de MIPS necesitaría dos instrucciones más: una `lui` para cargar los 16 bits de mayor peso del desplazamiento y una `add` para sumar la parte superior de la dirección con el registro base `$s1`. (Intel da dos nombres diferentes a lo que se llama modo de direccionamiento base –base e indexado–, pero son esencialmente idénticos y aquí los combinamos).

Operaciones de enteros x86

Los 8086 proporcionan soporte para tipos de datos de 8 bits (*byte*) y de 16 bits (*palabra*). Los 80386 agregan direcciones y datos de 32 bits (*doble palabra*) en el x86. (AMD64 añade direcciones y datos de 64 bits, llamados palabras cuadrúples, *quad word*; en esta sección seguiremos con el 80386.) Los distintos tipos de datos se aplican a las operaciones con registro, así como a los accesos a memoria. Casi todas las operaciones trabajan tanto con datos de 8 bits como con un tamaño de datos mayor. Este tamaño se determina por el modo y es de 16 o de 32 bits.

Algunos programas desean claramente funcionar con datos de los tres tamaños, así que los arquitectos del 80386 proporcionan una manera conveniente de especificar cada caso sin tener que extender el tamaño del código significativamente. Decidieron que en mucho programas dominarían los datos de 16 o de 32 bits, y por tanto parecía tener sentido poder fijar un tamaño grande por defecto. Este tamaño de datos por defecto es fijado mediante un bit en el registro de segmento de código. Para invalidar el tamaño de los datos por defecto se concatena a la instrucción un *prefijo* de 8 bits para decir a la máquina que debe utilizar el otro tamaño grande para esta instrucción.

La solución del prefijo se tomó prestada de los 8086, que permiten múltiples prefijos para modificar el comportamiento de la instrucción. Los tres prefijos originales invalidan el registro del segmento por defecto, bloquean el bus para soportar sincronización (véase la sección 2.11), o repiten la instrucción siguiente hasta que el registro ECX decrementa hasta 0. Este último prefijo fue pensado para ser emparejado con una instrucción de movimiento de byte, para mover un número variable de bytes. Los 80386 también añadieron un prefijo para invalidar el tamaño de la dirección por defecto.

Las operaciones de enteros x86 se pueden dividir en cuatro clases importantes:

1. Instrucciones de movimiento de datos, que incluyen mover (*move*), introducir (*push*) y sacar (*pop*).
2. Instrucciones de aritméticas y lógicas, que incluyen operaciones aritméticas de test, número entero y decimal.

3. Instrucciones para controlar el flujo, que incluyen bifurcaciones o saltos condicionales, saltos incondicionales, llamadas y retornos.
4. Instrucciones de cadenas, que incluyen movimiento y comparación de cadenas.

Las dos primeras categorías no son relevantes, no tienen nada especial, salvo que las instrucciones de operaciones aritméticas y lógicas permiten que el destino sea tanto un registro como una posición de memoria. La figura 2.39 muestra algunas instrucciones típicas x86 y sus funciones.

Las bifurcaciones o saltos condicionales en el x86 se basan en *códigos de condición* o indicadores (*flags*). Los códigos de condición se fijan como un efecto secundario de una operación; la mayoría se utilizan para comparar el valor de un resultado con 0. Luego las bifurcaciones o saltos condicionales comprueban los códigos de condición. Las direcciones de bifurcaciones o saltos condicionales relativas al PC se deben especificar en bytes, dado que, a diferencia de ARM y MIPS, las instrucciones 80386 no son todas de 4 bytes de longitud.

Las instrucciones de manipulación de cadenas son parte del 8080 anterior al x86 y no se ejecutan en la mayoría de los programas. Son a menudo más lentas que rutinas equivalentes del software (véase la falacia en la página 174).

La figura 2.40 enumera algunas de las instrucciones x86 de enteros. Muchas de las instrucciones están disponibles tanto en el formato byte como en el formato palabra.

Instrucción	Función
`JE` name	`si igual(código de condición){EIP=nombre};` `EIP-128 ≤ nombre < EIP+128`
`JMP` name	`EIP=nombre`
`CALL` name	`SP=SP-4; M[SP]=EIP+5; EIP=nombre;`
`MOVW EBX,[EDI+45]`	`EBX=M[EDI+45]`
`PUSH ESI`	`SP=SP-4; M[SP]=ESI`
`POP EDI`	`EDI=M[SP]; SP=SP+4`
`ADD EAX,#6765`	`EAX= EAX+6765`
`TEST EDX,#42`	Actualiza los códigos de condición (flags) con EDX y 42
`movsl`	`M[EDI]=M[ESI];` `EDI=EDI+4; ESI=ESI+4`

FIGURA 2.39 Algunas instrucciones típicas x86 y sus funciones. En la figura 2.40 aparece una lista de operaciones frecuentes. La instrucción `CALL` guarda el EIP de la siguiente instrucción en la pila. (EIP es el PC de Intel).

Codificación de las instrucciones en x86

Hemos guardado lo peor para el final: la codificación de instrucciones en los 80836 es compleja, con muchos formatos de instrucción diversos. Las instrucciones del 80386 pueden variar a partir de 1 byte, cuando no hay operandos, hasta 15 bytes.

La figura 2.41 muestra el formato de instrucciones para varias de las instrucciones del ejemplo de la figura 2.39. El byte del código de operación generalmente contiene un bit que indica si el operando es de 8 bits o de 32 bits. Para algunas instrucciones, el código de operación puede incluir el modo de direccionamiento y el registro; esto es cierto en muchas instrucciones que tienen la forma "registro =

Instrucción	Significado
Control	**Saltos condicionales e incondicionales**
JNZ, JZ	Salta si se cumple la condición a EIP + 8 bits de desplazamiento; JNE (por JNZ) y JE (por JZ) son nombres alternativos
JMP	Salto incondicional–desplazamiento de 8 bits o de 16 bits
CALL	Llamada a subrutina–desplazamiento de 16 bits; deja la dirección de retorno en la pila
RET	Saca la dirección de retorno de la pila y salta a ella
LOOP	Salto del lazo–decremento de ECX; salta a EIP + 8 bits de desplazamiento si ECX ≠0
Transferencia de datos	**Mueven datos entre registros o entre registros y memoria**
MOV	Mueven datos entre dos registros o entre registros y memoria
PUSH, POP	Push: guarda el operando fuente en la pila; pop: saca el operando de la parte alta de la pila y lo pone en un registro
LES	Carga ES y unos de los registros GPRs de memoria
Aritmética, lógica	**Operaciones aritméticas y lógicas utilizando registros y memoria**
ADD, SUB	Suma la fuente al destino; resta la fuente del destino; formato registro-memoria
CMP	Compara fuente y destino; formato registro-memoria
SHL, SHR, RCR	Desplazamiento a la izquierda; desplazamiento lógico a la derecha; rota a la derecha dando entrada al código de condición de acarreo
CBW	Convierte el byte de los 8 bits más a la derecha de EAX en una palabra de 16 bits a la derecha de EAX
TEST	AND lógico de fuente y destino, activa los códigos de condición
INC, DEC	Incrementa el destino, decrementa el destino
OR, XOR	OR lógico; OR exclusivo; formato registro-memoria
Cadenas	**Mueven datos entre operandos tipo cadena, con la longitud dada por un prefijo de repetición**
MOVS	Copia la cadena desde la fuente al destino incrementando ESI y EDI; se puede repetir
LODS	Carga un byte, una palabra o una doble palabra de una cadena en el registro EAX

FIGURA 2.40 Algunas operaciones típicas en el x86. Muchas operaciones utilizan el formato de registro-memoria, donde tanto el operando fuente como el destino pueden estar en memoria y el otro puede ser un registro o un operando inmediato.

registro operación inmediato". Otras instrucciones utilizan un *postbyte* o byte de código de operación adicional, etiquetado "mod, reg, r/m", que contiene la información del modo de direccionamiento. Este postbyte se utiliza para muchas de las instrucciones que tratan con la memoria. El modo base más índice escalado utiliza un segundo postbyte, etiquetado con "sc, índice, base".

La figura 2.42 muestra la codificación de los dos especificadores de dirección postbyte posibles para los modos de 16 bits y de 32 bits. Desafortunadamente, para entender completamente qué registro se coloca y qué modos de dirección están disponibles necesitamos ver la codificación de todos los modos de direccionamiento y a veces incluso la codificación de las instrucciones.

Conclusión sobre x86

Intel tenía un microprocesador de 16 bits dos años antes que las arquitecturas más elegantes de sus competidores, tales como el Motorola 68000, y esta delantera le llevó a tener ventaja en la elección de los 8086 como la CPU para el PC de IBM. Los ingenieros

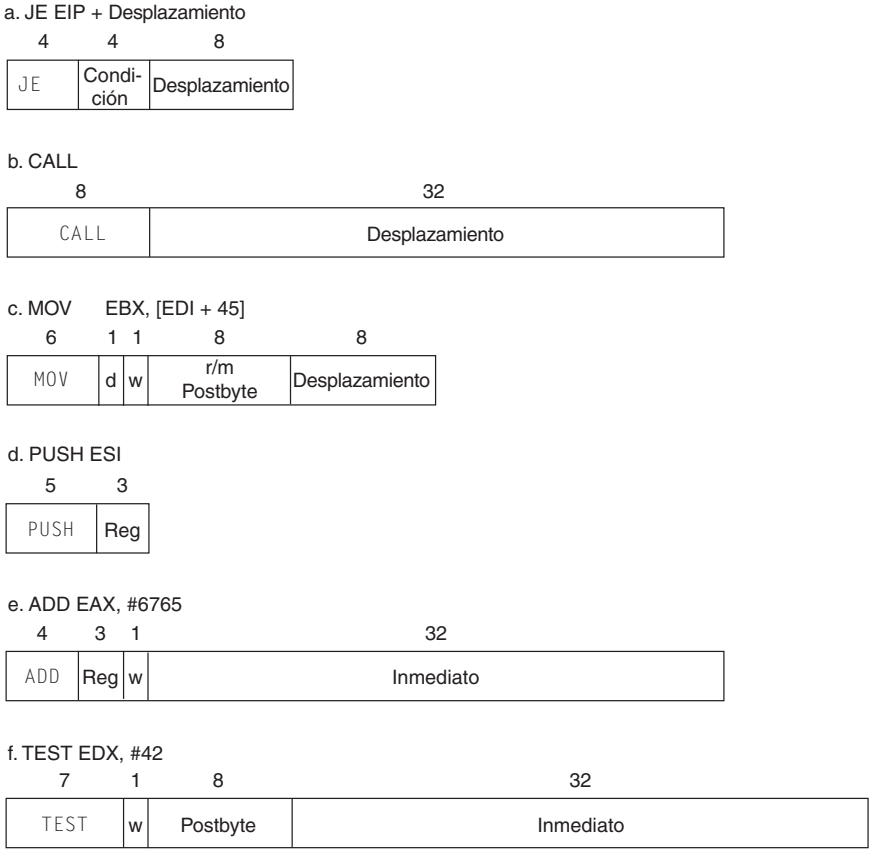


FIGURA 2.41 Formatos de instrucción típicos x86. La figura 2.42 muestra la codificación del postbyte. Muchas instrucciones contienen el campo `w` de 1 bit, que indica si la operación es un byte o una doble palabra. El campo `d` en `MOV` se utiliza en las instrucciones que pueden mover a o desde memoria y muestran la dirección del movimiento. La instrucción `ADD` requiere 32 bits para el campo inmediato porque en modos de 32 bits los immediatos son tanto de 8 bits como de 32 bits. El campo inmediato en `TEST` es de 32 bits de longitud porque no hay inmediatos de 8 bits para probar en modos de 32 bits. En general, las instrucciones pueden variar de 1 a 17 bytes de longitud. La longitud larga viene de prefijos adicionales de 1 byte, teniendo un inmediato de 4 bytes y una dirección de desplazamiento de 4 bytes, usando un código de operación de 2 bytes y utilizando un especificador del modo índice escalado, que agrega otro byte.

de Intel reconocen generalmente que el x86 es más difícil de construir que máquinas como MIPS, pero un mercado mayor significa que AMD e Intel pueden permitirse más recursos para ayudar a superar la complejidad añadida. Lo que le falta al x86 en estilo se compensa con la cantidad, haciéndolo bonito desde una perspectiva correcta.

El mérito excepcional es que los componentes arquitectónicos más frecuentemente usados en x86 no son demasiado difíciles de implementar, tal como AMD e Intel han demostrado rápidamente mejorando desde 1978 el funcionamiento de los programas de enteros. Para conseguir esas prestaciones, los compiladores deben evitar las partes de la arquitectura que son difíciles de poner en ejecución con rapidez.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*igual*	*igual*	*igual*	*igual*	*igual*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*dir que*	*dir que*	*dir que*	*dir que*	*dir que*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*campo*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	si+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURA 2.42 La codificación del primer especificador de la dirección del x86, "mod, registro, r/m". Las primeras cuatro columnas muestran la codificación del campo registro de 3 bits, que depende del bit w del código de operación y de si la máquina está en el modo de 16 bits (8086) o en el modo de 32 bits (80386). Las columnas restantes explican los campos mod y r/m. El significado del campo r/m de 3 bits depende del valor del campo mod de 2 bits y del tamaño de la dirección. Básicamente, los registros usados en el cálculo de la dirección se enumeran en la sexta y la séptima columnas, bajo mod = 0 y con mod = 1 que añade un desplazamiento de 8 bits y mod = 2 que añade un desplazamiento de 16 bits o de 32 bits, dependiendo del modo de direccionamiento. Las excepciones son r/m = 6 cuando mod = 1 o mod = 2 en modo de 16 bits selecciona BP más el desplazamiento; r/m = 5 cuando mod = 1 o mod = 2 en modo de 32 bits selecciona EBP más el desplazamiento; y r/m = 4 en modo de 32 bits cuando mod ≠ 3, donde (sib) significa utilizar el modo índice escalado mostrado en la figura 2.38. Cuando mod = 3, el campo r/m indica un registro, usando la misma codificación que el campo del registro combinado con el bit w.

2.18 Falacias y errores habituales

Falacia: instrucciones más potentes implican unas prestaciones mayores.

Una parte del potencial del Intel x86 son los prefijos que pueden modificar la ejecución de la instrucción siguiente. Un prefijo puede repetir la instrucción siguiente hasta que un contador, que se va decrementando, llegue. Así, para mover datos en memoria parece que la secuencia de instrucción natural es utilizar *move* con el prefijo de la repetición para realizar movimientos de memoria-a-memoria de 32 bits.

Un método alternativo, que utiliza las instrucciones estándares que tienen todos los computadores, es cargar los datos en los registros y después almacenar los registros de nuevo en memoria. Esta segunda versión del programa, con el código replicado para reducir la sobrecarga del lazo, copia aproximadamente 1.5 veces más rápido. Una tercera versión, que utiliza los registros de punto flotante en vez de los registros de enteros del x86, copia aproximadamente 2.0 veces más rápido que la instrucción compleja.

Falacia: escribir en lenguaje ensamblador para obtener prestaciones más altas.

Hace un tiempo, los compiladores para los lenguajes de programación producían secuencias de instrucciones sencillas; el aumento de la sofisticación de los compiladores implica que la distancia entre el código compilado y el código producido a mano se está acortando rápidamente. De hecho, para competir con los compiladores actuales el programador de lenguaje ensamblador necesita comprender a fondo los conceptos de los capítulos 4 y 5 (segmentación –*pipeline*– del procesador y jerarquía de memoria).

Esta batalla entre los compiladores y los codificadores del lenguaje ensamblador enmarca una situación en la cual los seres humanos están perdiendo terreno. Por ejemplo, C ofrece al programador una oportunidad para echarle una mano al compilador diciéndole qué variables se deberían mantener en registros frente a las volcadas en memoria. Cuando los compiladores eran pobres en la asignación de registros, este tipo de ayudas eran vitales para las prestaciones. De hecho, algunos libros de texto de C emplean bastante tiempo en dar ejemplos que utilizan con eficacia estas ayudas. Los compiladores C actuales generalmente ignoran tales ayudas porque el compilador hace un trabajo mejor en cuanto a la asignación que el programador.

Aunque el resultado de escribirlo a mano fuera un código más rápido, los peligros de escribir en lenguaje ensamblador son un mayor tiempo de codificación y depuración, la pérdida en portabilidad y la dificultad de mantener tal código. Uno de los pocos axiomas extensamente aceptados por la ingeniería del software es que la codificación lleva más tiempo si se escriben más líneas, y claramente se requieren muchas más líneas para escribir un programa en lenguaje ensamblador que en C o Java. Por otra parte, una vez codificado, el siguiente riesgo es que se convierta en un programa popular. Tales programas siempre viven un tiempo más largo que el esperado, y esto significa que alguien tendrá que poner al día el código al cabo de varios años y lo tendrá que hacer trabajar con las nuevas versiones de sistemas operativos y los nuevos modelos de máquinas. El escribir en lenguaje de alto nivel en vez hacerlo en lenguaje ensamblador no sólo permite a los compiladores futuros adaptar el código a las máquinas futuras, también hace más fácil mantener el software y permite que el programa funcione en más tipos de computadores.

Falacia: la importancia de la compatibilidad binaria comercial implica que los repertorios de instrucciones exitosos no cambian.

La compatibilidad binaria hacia atrás, a versiones anteriores del software, es sacrosanta, pero la figura 2.43 muestra que la arquitectura del x86 ha crecido significativamente. ¡La media es más de instrucción por mes en sus 30 años de vida!

Error: olvidar que las direcciones secuenciales de las palabras en máquinas con direccionamiento byte no se diferencian en uno.

Muchos programadores de lenguaje ensamblador han tenido que trabajar mucho sobre errores cometidos por suponer que la dirección de la palabra siguiente se puede encontrar incrementando la dirección en un registro, de uno en uno, en lugar del tamaño de la palabra en bytes. ¡El que avisa no es traidor!

Error: usar un puntero a una variable automática fuera de la definición de su procedimiento.

Un error común al trabajar con punteros es pasar un resultado de un procedimiento que incluye el puntero a una tabla declarada local a ese procedimiento. Siguiendo la disciplina de la pila en la figura 2.12, la memoria que contiene la tabla local será reutilizada tan pronto como el procedimiento retorne. Los punteros a las variables automáticas pueden conducir al caos.

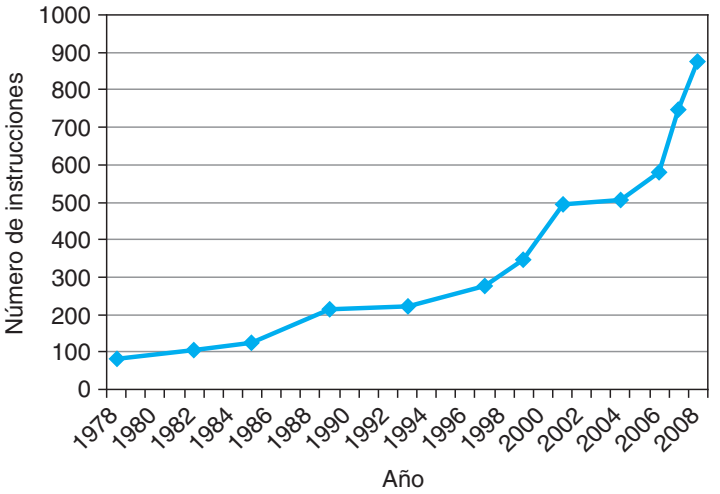


FIGURA 2.43 Aumento del repertorio de instrucciones x86 con el tiempo. Aunque hay una justificación técnica clara para alguna de estas extensiones, este aumento tan rápido también aumenta las dificultades que encuentran otros fabricantes para desarrollar procesadores compatibles.

Menos es más.
Robert Browning,
Andrea del Sarto, 1855

2.19 Conclusiones finales

Los dos principios del computador de programa almacenado (*stored-program*) son el uso de instrucciones que son indistinguibles de los números y el uso de la memoria alterable para los programas. Estos principios permiten que una sola máquina ayude a científicos ambientales, a consejeros financieros y a novelistas en sus especialidades. La selección de un repertorio de instrucciones que la máquina pueda entender exige un equilibrio delicado entre el número de instrucciones necesario para ejecutar un programa, el número de ciclos de reloj necesarios para cada instrucción y la velocidad del reloj. Cuatro principios de diseño guían a los autores de repertorios de instrucciones a la hora de encontrar este delicado equilibrio:

1. *La simplicidad favorece la regularidad.* La regularidad es el origen de muchas características del repertorio de instrucciones MIPS: mantiene todas las instrucciones de un mismo tamaño, siempre requiere tres registros de operandos para las instrucciones aritméticas y mantiene los campos de registros en el mismo lugar en cada formato de instrucción.

2. *Cuanto más pequeño, más rápido.* El deseo de velocidad es la razón por la que MIPS tiene 32 registros y no muchos más.

3. *Hacer rápido lo común (lo más frecuente)*. Ejemplos de hacer rápido acciones MIPS comunes incluyen el direccionamiento relativo al PC para las bifurcaciones o saltos condicionales y el direccionamiento inmediato para los operandos constante.

4. *El buen diseño exige buenos compromisos*. Un ejemplo para MIPS fue el compromiso entre permitir direcciones y constantes más grandes en instrucciones y mantener todas las instrucciones de la misma longitud.

Por encima de este nivel máquina está el lenguaje ensamblador, un lenguaje que los humanos podemos leer. El ensamblador lo traduce a números binarios que las máquinas puedan entender e incluso "amplían" el repertorio de instrucciones, creando instrucciones simbólicas que no están en el hardware. Por ejemplo, las constantes o las direcciones que son demasiado grandes están partidas en trozos del tamaño apropiado, las variaciones comunes de instrucciones tienen su propio nombre, y así sucesivamente. La figura 2.44 enumera las instrucciones MIPS que hemos estudiado hasta ahora, tanto las reales como las pseudoinstrucciones.

Cada categoría de instrucciones MIPS se asocia a las construcciones que aparecen en lenguajes de programación:

- Las instrucciones aritméticas se corresponden con las operaciones basadas en sentencias de asignación.

- Las instrucciones de transferencia de datos son más frecuentes cuando se trabaja con estructuras de datos como tablas o estructuras.

- Las bifurcaciones o saltos condicionales se utiliza en sentencias si (*if*) y en lazos.

- Los saltos incondicionales se utilizan en llamadas y retornos de procedimiento y para las sentencias *case/switch*.

Estas instrucciones no nacen iguales; la popularidad de unas cuantas domina a las otras. Por ejemplo, la figura 2.45 muestra la frecuencia de aparición de cada clase de instrucciones para SPEC2006. La distinta popularidad de unas instrucciones respecto a otras desempeña un papel importante en los capítulos sobre el camino de datos (*datapath*), control y segmentación.

Después de explicar la aritmética del computador en el capítulo 3, analizaremos más la arquitectura del repertorio de instrucciones MIPS.

Instrucciones MIPS	Nombre	Formato	Pseudo MIPS	Nombre	Formato
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
store byte	sb	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURA 2.44 El repertorio de instrucciones MIPS visto hasta ahora, con las instrucciones reales MIPS a la izquierda y las pseudoinstrucciones a la derecha. El apéndice B (sección B.10) describe la arquitectura completa MIPS. La figura 2.1 muestra más detalles de la arquitectura MIPS tratada en este capítulo. La información se encuentra también en las columnas 1 y 2 de la Tarjeta de Datos de Referencia MIPS, que se incluye con este libro.

Tipo de Instrucción	Ejemplos MIPS	Correspondencia con el lenguaje de alto nivel (HLL)	Frecuencia	
			Enteros	Pt. fl.
Aritmética	`add, sub, addi`	Operaciones en sentencias de asignación	24%	48%
Transferencia de datos	`lw, sw, lb, sb, lui`	Referencias a estructuras de datos, tales como tablas	36%	39%
Lógica	`and, or, nor, andi, ori, sll, srl`	Operaciones en sentencias de asignación	18%	4%
Bifurcación condicional	`beq, bne, slt, slti`	Sentencias si *(if)* y lazos	18%	6%
Salto incondicional	`j, jr, jal`	Llamadas a procedimientos, retornos y sentencias *case/switch*	3%	0%

FIGURA 2.45 Tipos de instrucción MIPS, ejemplos, correspondencia con construcciones de lenguaje de programación de alto nivel y porcentaje de instrucciones MIPS ejecutadas por categoría para el promedio del SPEC2006. La figura 3.26 en el capítulo 3 muestra el porcentaje de las instrucciones MIPS ejecutadas individualmente.

Perspectiva histórica y lecturas recomendadas

Esta sección examina la historia de las líneas maestras del repertorio de instrucciones a lo largo del tiempo, y damos un repaso de los lenguajes de programación y de los compiladores. Los ISAs incluyen arquitecturas con acumulador, arquitecturas con registros de propósito general, arquitecturas con pila y una breve historia del ARM y x86. También revisamos los temas polémicos de las arquitecturas de computador con lenguajes de alto nivel y de las arquitecturas de computador con repertorios de instrucciones reducidos. La historia de los lenguajes de programación incluye Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smaltalk, C++ y Java, y la historia de los compiladores incluye los hitos clave y a los pioneros que los alcanzaron. El resto de esta sección está en el CD.

2.21 Ejercicios

Contribución de John Oliver, Cal Poly, San Luis Obispo, con la colaboración de Nicole Kaiyan (University of Adelaide) y Milos Prvulovic (Georgia Tech)

El simulador MIPS, que es útil para estos ejercicios, se describe en el apéndice B. Aunque el simulador acepta pseudoinstrucciones, es mejor no utilizarlas en los ejercicios que piden desarrollar un código MIPS. El objetivo debe ser aprender el repertorio real de instrucciones de MIPS, y si se pide contar instrucciones, la cuenta debe reflejar el número real de instrucciones que se van ejecutar y no las pseudoinstrucciones.

Las pseudoinstrucciones pueden ser utilizadas en algunos casos (por ejemplo, la instrucción `la` cuando no se conoce el valor real en tiempo de ensamblaje). En

muchos casos, son convenientes y producen un código más legible (por ejemplo, las instrucciones `li` y `move`). Si se elige utilizar pseudoinstrucciones por estos motivos, por favor añádase un par de frases explicando qué pseudoinstrucciones se han usado y por qué.

Ejercicio 2.1

Los siguientes problemas abordan la traducción de C a MIPS. Suponga que las variables `g`, `h`, `i`, `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`f = g + h + i + j;`
b.	`f = g + (h + 5);`

2.1.1 [5]<2.2> ¿Cuál es el código MIPS para las instrucciones C de la tabla? Utilice el número mínimo posible de instrucciones ensamblador de MIPS.

2.1.2 [5]<2.2> ¿Cuántas instrucciones ensamblador de MIPS resultan de la traducción de las instrucciones C?

2.1.3 [5]<2.2> Si las variables `f`, `g`, `h`, `i` y `j` tienen valores 1, 2, 3, 4 y 5, respectivamente, ¿cuál es valor final de `f`?

Los siguientes problemas abordan la traducción de MIPS a C. Suponga que las variables `g`, `h`, `i` y `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`add  f, g, h`
b.	`addi  f, f, 1` `add  f, g, h`

2.1.4 [5]<2.2> ¿Cuál es la sentencia C equivalente para las instrucciones MIPS de la tabla?

2.1.5 [5]<2.2> Si las variables `f`, `g`, `h` e `i` tienen valores 1, 2, 3 y 4, respectivamente, ¿cuál es valor final de `f`?

Ejercicio 2.2

Los siguientes problemas abordan la traducción de C a MIPS. Suponga que las variables `g`, `h`, `i` y `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`f = f + f + i;`
b.	`f = g + (j + 2);`

2.2.1 [5]<2.2> ¿Cuál es el código MIPS para las instrucciones C de la tabla? Utilice el número mínimo posible de instrucciones ensamblador de MIPS.

2.2.2 [5]<2.2> ¿Cuántas instrucciones ensamblador de MIPS resultan de la traducción de las instrucciones C?

2.2.3 [5]<2.2> Si las variables `f`, `g`, `h` e `i` tienen valores 1, 2, 3 y 4, respectivamente, ¿cuál es valor final de `f`?

Los siguientes problemas abordan la traducción de MIPS a C. Suponga que las variables `g`, `h`, `i` y `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`add  f, f, h`
b.	`sub  f, $0, f` `addi  f, f, 1`

2.2.4 [5]<2.2> ¿Cuál es la sentencia C equivalente para las instrucciones MIPS de la tabla?

2.2.5 [5]<2.2> Si las variables `f`, `g`, `h` e `i` tienen valores 1, 2, 3 y 4, respectivamente, ¿cuál es valor final de `f`?

Ejercicio 2.3

Los siguientes problemas abordan la traducción de C a MIPS. Suponga que las variables `g`, `h`, `i` y `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`f = f + g + h + i + j + 2;`
b.	`f = g - (f + 5);`

2.3.1 [5]<2.2> ¿Cuál es el código MIPS para las instrucciones C de la tabla? Utilice el número mínimo posible de instrucciones ensamblador de MIPS.

2.3.2 [5]<2.2> ¿Cuántas instrucciones ensamblador de MIPS resultan de la traducción de las instrucciones C?

2.3.3 [5]<2.2> Si las variables `f`, `g`, `h`, `i` y `j` tienen valores 1, 2, 3, 4 y 5, respectivamente, ¿cuál es valor final de `f`?

Los siguientes problemas abordan la traducción de MIPS a C. Suponga que las variables `g`, `h`, `i` y `j` son conocidas y son enteros de 32 bits, tal como se declaran en C.

a.	`add  f, -g, h`
b.	`addi  h, f, 1` `sub  f, g, h`

2.3.4 [5]<2.2> ¿Cuál es la sentencia C equivalente para las instrucciones MIPS de la tabla?

2.3.5 [5]<2.2> Si las variables `f`, `g`, `h` e `i` tienen valores 1, 2, 3 y 4, respectivamente, ¿cuál es valor final de `f`?

Ejercicio 2.4

Los siguientes problemas abordan la traducción de C a MIPS. Suponga que las variables `f`, `g`, `h`, `i` y `j` se han asignado a los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4`, respectivamente. Suponga que las direcciones base de los vectores `A` y `B` están en los registros `$s6` y `$s7`, respectivamente.

a.	`f = g + h + B[4];`
b.	`f = g - A[B[4]];`

2.4.1 [10]<2.2, 2.3> ¿Cuál es el código MIPS para las instrucciones C de la tabla? Utilice el número mínimo posible de instrucciones ensamblador de MIPS.

2.4.2 [5]<2.2, 2.3> ¿Cuántas instrucciones ensamblador de MIPS resultan de la traducción de las instrucciones C?

2.4.3 [5]<2.2, 2.3> ¿Cuántos registros diferentes es necesario utilizar?

Los siguientes problemas abordan la traducción de MIPS a C. Suponga que las variables `f`, `g`, `h`, `i` y `j` se han asignado a los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4`, respectivamente. Suponga que las direcciones base de los vectores `A` y `B` están en los registros `$s6` y `$s7`, respectivamente.

a.	`add $s0, $s0, $s1`
	`add $s0, $s0, $s2`
	`add $s0, $s0, $s3`
	`add $s0, $s0, $s4`

b.	`lw $s0, 4($s6)`

2.4.4 [5]<2.2, 2.3> ¿Cuál es la sentencia C equivalente para las instrucciones MIPS de la tabla?

2.4.5 [5]<2.2, 2.3> Reescriba el código ensamblador para minimizar (si es posible) el número de instrucciones MIPS necesarias para llevar a cabo la misma operación de la tabla.

2.4.6 [5]<2.2, 2.3> ¿Cuántos registros diferentes es necesario utilizar tal como está escrito el código anterior? Con el nuevo código, ¿cuál es el número mínimo de registros necesario?

Ejercicio 2.5

En los siguientes problemas se estudian las operaciones de memoria en el contexto de un procesador MIPS. Los valores de un vector almacenado en memoria se muestran en la tabla.

a.

Dirección	Valor
12	1
8	6
4	4
0	2

b.

Dirección	Valor
16	1
12	2
8	3
4	4
0	5

2.5.1 [10]<2.2, 2.3> Para las posiciones de memoria mostradas en la tabla, escriba el código C que ordene los valores de menor a mayor, situando el menor valor en la posición de memoria más pequeña de las mostradas. Suponer que los datos se han representado en un vector de enteros de C de nombre `array`, que el procesador es direccionable por bytes y que una palabra está formada por 4 bytes.

2.5.2 [10]<2.2, 2.3> Para las posiciones de memoria mostradas en la tabla, escriba el código MIPS que ordene los valores de menor a mayor, situando el menor valor en la posición de memoria más pequeña de las mostradas. Utilice el número mínimo de instrucciones MIPS. Suponga que la dirección base de la tabla está en el registro `$s6`.

2.5.3 [5]<2.2, 2.3> ¿Cuántas instrucciones ensamblador de MIPS son necesarias para el ordenamiento? Si no se pudiese utilizar un campo inmediato en las instrucciones `lw` y `sw`, ¿cuántas instrucciones se necesitarían?

En los siguientes problemas se estudia la conversión de números hexadecimales a otros formatos.

a.	`0x12345678`
b.	`0xbeadf00d`

2.5.4 [5]<2.3> Convierta los número hexadecimales de la tabla a decimales.

2.5.5 [5]<2.3> Indique cómo se almacenarían los datos de la tabla en un procesador *little-endian* o uno *big-endian*. Suponga que los datos se almacenan a partir de la dirección 0.

Ejercicio 2.6

Los siguientes problemas abordan la traducción de C a MIPS. Suponga que las variables `f`, `g`, `h`, `i`, y `j` se han asignado a los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4`, respectivamente. Suponga que las direcciones base de los vectores `A` y `B` están en los registros `$s6` y `$s7`, respectivamente.

a.	`f = -g + h + B[1];`
b.	`f = A[B[g]+1];`

2.6.1 [10]<2.2, 2.3> ¿Cuál es el código MIPS para las instrucciones C de la tabla? Utilice el número mínimo posible de instrucciones ensamblador de MIPS.

2.6.2 [5]<2.2, 2.3> ¿Cuántas instrucciones ensamblador de MIPS resultan de la traducción de las instrucciones C?

2.6.3 [5]<2.2, 2,3> ¿Cuántos registros diferentes es necesario utilizar?

Los siguientes problemas abordan la traducción de MIPS a C. Suponga que las variables f, g, h, i y j se han asignado a los registros `$s0`, `$s1`, `$s2`, `$s3` y `$s4`, respectivamente. Suponga que las direcciones base de los vectores `A` y `B` están en los registros `$s6` y `$s7`, respectivamente.

a.	`add $s0, $s0, $s1`
	`add $s0, $s3, $s2`
	`add $s0, $s0, $s3`

b.	`addi $s6, $s6, -20`
	`add $s6, $s6, $s1`
	`lw $s0, 8($s6)`

2.6.4 [5]<2.2, 2.3> ¿Cuál es la sentencia C equivalente para las instrucciones MIPS de la tabla?

2.6.5 [5]<2.2, 2.3> Suponga que los valores contenidos en los registros `$s0`, `$s1`, `$s2` y `$s3` son, respectivamente, 10, 20, 30 y 40, que el registro `$s6` tiene el valor 256 y que en la memoria están los siguientes valores:

Dirección	Valor
256	100
260	200
264	300

Determine el valor en `$s0` al terminar el código ensamblador.

2.6.6 [10]<2.3, 2.5> Indique el valor de los campos op, rs y rt para cada instrucción MIPS. Indique el valor del campo inmediato para las instrucciones de tipo I, y del campo rd para las instrucciones de tipo R.

Ejercicio 2.7

En los siguientes problemas se estudia la conversión de números binarios con y sin signo a número decimales.

a.	1010 1101 0001 0000 0000 0000 0000 0010_{dos}
b.	1111 1111 1111 1111 1011 0011 0101 0011_{dos}

2.7.1 [5]<2.4> Suponiendo que los números de la tabla son enteros en complemento a 2, ¿qué números decimales representan?

2.7.2 [5]<2.4> Suponiendo que los números de la tabla son enteros sin signo, ¿qué números decimales representan?

2.7.3 [5]<2.4> ¿Qué números hexadecimales representan?

En los siguientes ejercicios se estudia la conversión de números decimales a binarios con y sin signo.

a.	2147483647_{diez}
b.	1000_{diez}

2.7.4 [5]<2.4> Convierta los números de la tabla en una representación binaria en complemento a 2.

2.7.5 [5]<2.4> Convierta los números de la tabla en una representación hexadecimal en complemento a 2.

2.7.6 [5]<2.4> Convierta los negativos de los números de la tabla en una representación hexadecimal en complemento a 2.

Ejercicio 2.8

En los siguientes problemas se estudia la extensión de signo y el desbordamiento. El contenido de los registros `$s0` y `$s1` son los valores mostrados en la tabla. Se le pedirá que haga algunas operaciones MIPS sobre estos registros y que muestre los resultados.

a.	`$s0` = $70000000_{dieciseis}$, `$s1` = $0x0FFFFFFF_{dieciseis}$
b.	`$s0` = $0x40000000_{dieciseis}$, `$s1` = $0x40000000_{dieciseis}$

2.8.1 [5]<2.4> Determine el valor de `$t0` después de la siguiente instrucción:

```
add $t0, $s0, $s1
```

¿Se ha producido desbordamiento?

2.8.2 [5]<2.4> Determine el valor de `$t0` después de la siguiente instrucción:

```
sub $t0, $s0, $s1
```

¿Se ha producido desbordamiento?

2.8.3 [5]<2.4> Determine el valor de `$t0` después de la siguiente instrucción:

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

¿Se ha producido desbordamiento?

En los siguientes problemas deberá realizar operaciones MIPS sobre los registros `$s0` y `$s1`. Dados los valores de `$s0` y `$s1` en cada una de las preguntas, diga si se produce desbordamiento.

a.	`add $s0, $s0, $s1`
b.	`sub $s0, $s0, $s1` `sub $s0, $s0, $s1`

2.8.4 [5]<2.4> Si los contenidos de los registros son `$s0` = 0x70000000 y `$s1` = 0x10000000, ¿se produce desbordamiento?

2.8.5 [5]<2.4> Si los contenidos de los registros son `$s0` = 0x40000000 y `$s1` = 0x20000000, ¿se produce desbordamiento?

2.8.6 [5]<2.4> Si los contenidos de los registros son `$s0` = 0x8FFFFFFF y `$s1` = 0xD0000000, ¿se produce desbordamiento?

Ejercicio 2.9

En la tabla se muestran varios valores para el registro `$s1`. Determine si se produce desbordamiento para ciertas operaciones.

a.	$\texttt{\$s1} = 2147483647_{\text{diez}}$
b.	$\texttt{\$s1} = \texttt{0xD0000000}_{\text{dieciseis}}$

2.9.1 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x70000000. ¿Se produce desbordamiento al hacer la operación add `$s0`, `$s0`, `$s1`?

2.9.2 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x80000000. ¿Se produce desbordamiento al hacer la operación sub `$s0`, `$s0`, `$s1`?

2.9.3 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x7FFFFFFF. ¿Se produce desbordamiento al hacer la operación add `$s0`, `$s0`, `$s1`?

En la tabla se muestran varios valores para el registro `$s1`. Determine si se produce desbordamiento para ciertas operaciones.

a.	`$s1 = 1010 1101 0001 0000 0000 0000 0000 0010`$_{dos}$
b.	`$s1 = 1111 1111 1111 1111 1011 0011 0101 0011`$_{dos}$

2.9.4 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x70000000. ¿Se produce desbordamiento al hacer la operación add `$s0`, `$s0`, `$s1`?

2.9.5 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x70000000. ¿Cuál es el resultado en hexadecimal de la operación add `$s0`, `$s0`, `$s1`?

2.9.6 [5]<2.4> Suponga que el contenido del registro `$s0` es `$s0` = 0x70000000. ¿Cuál es el resultado en base diez de la operación add `$s0`, `$s0`, `$s1`?

Ejercicio 2.10

La tabla muestra la codificación binaria de dos instrucciones. Tradúzcalas a ensamblador y averigüe qué instrucción MPS representan.

a.	`1010 1110 0000 1011 0000 0000 0000 0100`$_{dos}$
b.	`1000 1101 0000 1000 0000 0000 0100 0000`$_{dos}$

2.10.1 [5]<2.5> ¿Qué instrucciones representan los valores de la tabla?

2.10.2 [5]<2.5> ¿Qué tipo de instrucciones (tipo I, tipo R) representan los valores de la tabla?

2.10.3 [5]<2.5> Si los valores de la tabla fuesen datos en lugar de instrucciones, ¿qué número hexadecimal representan?

La tabla muestra dos instrucciones MIPS. Tradúzcalas y determine el formato.

a.	`add  $t0, $t0, $zero`
b.	`lw   $t1, 4($s3)`

2.10.4 [5]<2.4, 2.5> Indique la representación hexadecimal de las dos instrucciones.

2.10.5 [5]<2.5> ¿Qué tipo de instrucciones (tipo I, tipo R) son las instrucciones de la tabla?

2.10.6 [5]<2.5> ¿Cuál es representación hexadecimal del código de operación, y los campos rs y rt en esta instrucción? ¿Cuál es representación hexadecimal de los campos rd y func en las instrucciones tipo R? ¿Cuál es representación hexadecimal del campo inmediato en las instrucciones tipo I?

Ejercicio 2.11

La tabla muestra la representación del código de operación de dos instrucciones. Se le pedirá que las traduzca a ensamblador y averigüe qué instrucción MIPS representan.

a.	`0xAE0BFFFC`
b.	`0x8D08FFC0`

2.11.1 [5]<2.4, 2.5> ¿Cuál es la representación binaria de los números hexadecimales de la tabla?

2.11.2 [5]<2.4, 2.5> ¿Cuál es la representación decimal de los números hexadecimales de la tabla?

2.11.3 [5]<2.5> ¿Qué instrucciones representan los números hexadecimales de la tabla?

La tabla contiene los valores de varios campos de instrucciones MIPS. Se le pedirá que averigüe qué instrucciones son y que determine el formato MIPS.

a.	**op=0, rs=1, rt=2, rd=3, shamt=0, funct=32**
b.	**op=0x2B, rs=0x10, rt=0x5, const=0x4**

2.11.4 [5]<2.5> ¿Qué tipo de instrucción (tipo I, tipo R) representan los valores de la tabla?

2.11.5 [5]<2.5> ¿Cuáles son las instrucciones MIPS que representan los valores de la tabla?

2.11.6 [5]<2.4, 2.5> ¿Cuál es la representación binaria de estas instrucciones?

Ejercicio 2.12

La tabla contiene varias modificaciones que podrían hacerse a la arquitectura del repertorio de instrucciones de MIPS. Deberá investigar el impacto de estos cambios en el formato de las instrucciones.

a.	8 registros
b.	Constantes inmediatas de 10 bits

2.12.1 [5]<2.5> Si se cambia el repertorio de instrucciones del procesador MIPS, también debe cambiarse el formato de las instrucciones. Para los cambios sugeridos en la tabla, indique el tamaño en bits de los campos de una instrucción tipo R. ¿Cuál es el número de bits necesario para cada instrucción?

2.12.2 [5]<2.5> Si se cambia el repertorio de instrucciones del procesador MIPS, también debe cambiarse el formato de las instrucciones. Para los cambios sugeridos en la tabla, indique el tamaño en bits de los campos de una instrucción tipo I. ¿Cuál es el número de bits necesario para cada instrucción?

2.12.3 [5]<2.5, 2.10> ¿Por qué los cambios sugeridos en la tabla podrían reducir el tamaño de un programa ensamblador MIPS? ¿Por qué los cambios sugeridos en la tabla podrían aumentar el tamaño de un programa ensamblador MIPS?

La tabla contiene valores hexadecimales. Deberá averiguar qué instrucción MIPS representan y determinar el formato.

a.	`0x01090010`
b.	`0x8D090012`

2.12.4 [5]<2.5> ¿Cuál es la representación decimal de los valores de la tabla?

2.12.5 [5]<2.5> ¿Qué instrucción representan los números hexadecimales de la tabla?

2.12.6 [5]<2.4, 2.5> ¿Qué tipo de instrucción (tipo I, tipo R) representan los valores de la tabla? ¿Cuál es el valor de los campos op y rt?

Ejercicio 2.13

La tabla muestra el contenido de los registros `$t0` y `$t1`. Deberá hacer varias operaciones lógicas sobre estos registros.

a.	`$t0 = 0x55555555, $t1 = 0x12345678`
b.	`$t0 = 0xBEADFEED, $t1 = 0xDEADFADE`

2.13.1 [5]<2.6> Determine el valor final de `$t2` en la siguiente secuencia de instrucciones:

```
sll $t2, $t0, 4
or $t2, $t2, $t1
```

2.13.2 [5]<2.6> Determine el valor final de `$t2` en la siguiente secuencia de instrucciones:

```
sll $t2, $t0, 4
andi $t2, $t2, -1
```

2.13.3 [5]<2.6> Determine el valor final de `$t2` en la siguiente secuencia de instrucciones:

```
srl $t2, $t0, 3
andi $t2,  $t2, 0xFFEF
```

La tabla muestra varias operaciones lógicas de MIPS. Deberá encontrar el resultado de esas operaciones para ciertos valores de `$t0` y `$t1`.

a.	`sll $t2, $t0, 1` `or $t2, $t2, $t1`
b.	`srl $t2, $t0, 1` `andi $t2,  $t2, 0x00F0`

2.13.4 [5]<2.6> Suponiendo que `$t0` = 0x0000A5A5 y `$t1` = 00005A5A, determine el valor de $t2 después de las dos instrucciones de la tabla.

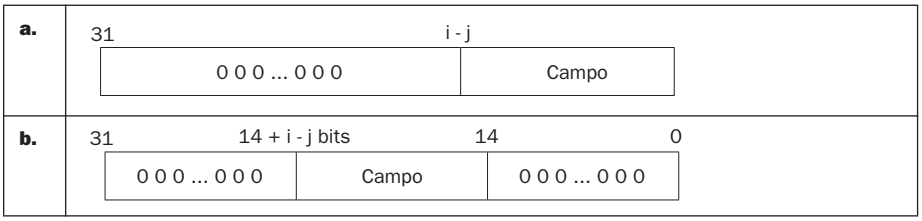
2.13.5 [5]<2.6> Suponiendo que `$t0` = 0xA5A50000 y `$t1` = A5A50000, determine el valor de $t2 después de las dos instrucciones de la tabla.

2.13.6 [5]<2.6> Suponiendo que `$t0` = 0xA5A5FFFF y `$t1` = A5A5FFFF, determine el valor de $t2 después de las dos instrucciones de la tabla.

Ejercicio 2.14

La siguiente figura muestra la posición de un grupo de bits en el registro `$t0`.

En los siguientes ejercicios deberá escribir instrucciones MIPS para extraer el grupo de bits "Campo" de `$t0` y situarlos en el registro `$t1` en las posiciones indicadas en la siguiente tabla.

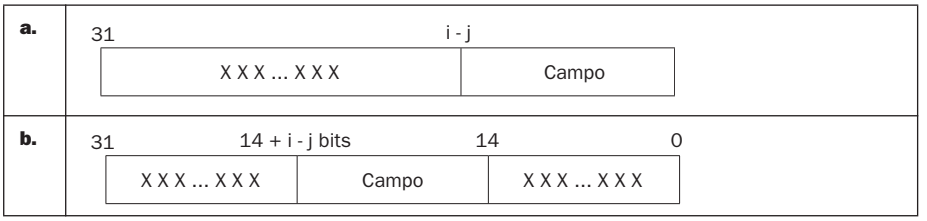


2.14.1 2.14.1 [20]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para los bits de "Campo" de `$t0`, para i = 22 y j = 5 y situarlo en `$t1` con el formato mostrado en la tabla.

2.14.2 2.14.2 [5]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para los bits de "Campo" de `$t0`, para i = 4 y j = 0 y situarlo en `$t1` con el formato mostrado en la tabla.

2.14.3 2.14.3 [5]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para los bits de "Campo" de `$t0`, para i = 3 y j = 28 y situarlo en `$t1` con el formato mostrado en la tabla.

En los siguientes ejercicios deberá escribir instrucciones MIPS para extraer los bits de "Campo" de `$t0` y situarlos en el registro `$t1` en las posiciones indicadas en la siguiente tabla. Los bits marcados como "`XXX`" no pueden cambiarse.



2.14.4 [20]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para bits de "Campo" de `$t0`, para i = 17 y j = 11 y situarlo en `$t1` con el formato mostrado en la tabla.

2.14.5 [5]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para bits de "Campo" de `$t0`, para i = 5 y j = 0 y situarlo en `$t1` con el formato mostrado en la tabla.

2.14.6 [5]<2.6> Encuentre la secuencia de instrucciones MIPS más corta para bits de "Campo" de `$t0`, para i = 31 y j = 29 y situarlo en `$t1` con el formato mostrado en la tabla.

Ejercicio 2.15

Para estos problemas, en la tabla se muestran algunas operaciones lógicas que no están en el repertorio de instrucciones de MIPS. ¿Como se implementan?

a.	`andn $t1, $t2, $t3  // operación AND bit a bit de $t2 y !$t3`
b.	`xnor $t1, $t2, $t3  // operación NOR exclusiva`

2.15.1 [5]<2.6> Las instrucciones lógicas de la tabla no están en el repertorio de instrucciones de MIPS. Determine el resultado si `$t2` = 0x00FFA5A5 y `$t3` = 0xFFFF003C.

2.15.2 [10]<2.6> Las instrucciones lógicas de la tabla no están en el repertorio de instrucciones de MIPS, pero pueden implementarse utilizando una o varias instrucciones MIPS. Encuentre el conjunto mínimo de instrucciones MIPS que pueden usarse en lugar de las instrucciones de la tabla.

2.15.3 [5]<2.6> Muestre la representación a nivel de bit de las instrucciones en la secuencia del ejercicio 2.15.2.

En la tabla se muestran varias instrucciones lógicas de C. En este ejercicio deberá evaluar e implementar estas instrucciones C utilizando instrucciones ensamblador de MIPS.

a.	`A = B & C[0]`
b.	`A = A ? B : C[0]`

2.15.4 [5]<2.6> La tabla muestra instrucciones de C que utilizan operadores lógicos. Si la posición de memoria de C[0] contiene el valor entero 0x00001234, y los valores enteros iniciales de `A` y `B` son 0x00000000 y 0x00002222, ¿cuál es el valor final de `A`?

2.15.5 [5]<2.6> Escriba la secuencia mínima de instrucciones ensamblador de MIPS que hacen la misma operación que las instrucciones C de la tabla.

2.15.6 [5]<2.6> Muestre la representación a nivel de bit de las instrucciones en la secuencia del ejercicio 2.15.5.

Ejercicio 2.16

La tabla muestra varios valores binarios del registro `$t0`. Dado el valor de `$t0` deberá determinar la salida de varias instrucciones de salto.

a.	1010 1101 0001 0000 0000 0000 0000 0010_{dos}
b.	1111 1111 1111 1111 1111 1111 1111 1111_{dos}

2.16.1 [5]<2.7> Suponga que el registro `$t0` tiene el valor mostrado en tabla y `$t1` tiene el valor

$0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{dos}$

¿Cuál es el valor de `$t2` después de las siguientes instrucciones?

```
        slt   $t2, $t0, $t1
        beq   $t2, $zero, ELSE
        j     DONE
ELSE:   addi  $t2, $zero, 2
DONE:
```

2.16.2 [5]<2.7> Suponga que el registro `$t0` tiene el valor mostrado en tabla y que se compara con el valor `X` en la siguiente instrucción MIPS. ¿Para qué valores de `X` (si hay alguno) `$t2` será igual a 1?

```
slti $t2, $t0, X
```

2.16.3 [5]<2.7> Suponga que el contador de programa (PC) toma el valor 0x00000020. ¿Es posible utilizar la instrucción MIPS salto incondicional (j) para poner el valor mostrado en la tabla en el PC? ¿Es posible utilizar la instrucción MIPS salto-si-igual (beq) para poner el valor mostrado en la tabla en el PC?

La tabla muestra varios valores binarios para el registro `$t0`. Dado el valor de `$t0`, deberá evaluar la salida de diferentes saltos.

a.	0x00001000
b.	0x20001400

2.16.4 [5]<2.7> Suponga que el registro `$t0` tiene el valor mostrado en tabla, ¿cuál es el valor de `$t2` después de las siguientes instrucciones?

```
        slt   $t2, $t0, $t0
        bne   $t2, $zero, ELSE
        j     DONE
ELSE:   addi  $t2, $t2, 2
DONE:
```

2.16.5 [5]<2.7> Suponga que el registro `$t0` tiene el valor mostrado en tabla, ¿cuál es el valor de `$t2` después de las siguientes instrucciones?

```
sll $t0, $t0, 2
slt $t2, $t0, $zero
```

2.16.6 [5]<2.7> Suponga que el contador de programa (PC) toma el valor 0x20000000. ¿Es posible utilizar la instrucción MIPS salto incondicional (j) para poner el valor mostrado en la tabla en el PC? ¿Es posible utilizar la instrucción MIPS salto-si-igual (`beq`) para poner el valor mostrado en la tabla en el PC?

Ejercicio 2.17

Para estos problemas se muestran varias instrucciones que no están en el repertorio de instrucciones de MIPS.

a.	`abs $t2, $t3` `# R[rd]=|R[rt]|`
b.	`sgt $t1, $t2, $t3` `# R[rd]=( R[rs] > R[rt]) ? 1:0`

2.17.1 [5]<2.7> La tabla muestra varias instrucciones que no están en el repertorio de instrucciones de MIPS y la descripción de cada instrucción. ¿Por qué no están incluidas en el conjunto de instrucciones de MIPS?

2.17.2 [5]<2.7> La tabla muestra varias instrucciones que no están en el repertorio de instrucciones de MIPS y la descripción de cada instrucción. Si estas instrucciones se implementasen en el conjunto de instrucciones MIPS, ¿cuál sería el formato de instrucción más apropiado?

2.17.3 [5]<2.7> Encuentre la secuencia de instrucciones MIPS más corta que realiza la misma operación que las instrucciones de la tabla.

La siguiente tabla contiene fragmentos de código ensamblador MIPS. Deberá evaluar cada uno de estos fragmentos y familiarizarse con las instrucciones de salto.

a.
```
LOOP:  slt  $t2, $0, $t1
       bne  $t2, $zero, ELSE
       j    DONE
ELSE:  addi $s2, $s2, 2
       subi $t1, $t1, 1
       j    LOOP
DONE:
```

b.
```
LOOP:  addi $t2, $0, 0xA
LOOP2: addi $s2, $s2, 2
       subi $t2, $t2, 1
       bne  $t2, $0, LOOP2
       subi $t1, $t1, 1
       bne  $t1, $0, LOOP
DONE:
```

2.17.4 [5]<2.7> En los lazos anteriores escritos en ensamblador de MIPS, suponga que el valor inicial del registro `$t1` es 10, si el valor inicial en `$s2` es 0, ¿cuál es el valor final en el registro `$s2`?

2.17.5 [5]<2.7> Escriba la rutina C equivalente a cada uno de los lazos de la tabla. Suponga que los registros `$s1`, `$s2`, `$t1` y `$t2` son los enteros `A`, `B`, `i` y `temp`, respectivamente.

2.17.6 [5]<2.7> En los lazos anteriores escritos en ensamblador de MIPS, suponga que el valor inicial del registro `$t1` es N. ¿Cuántas instrucciones MIPS se ejecutan?

Ejercicio 2.18

La siguiente tabla muestra fragmentos de código C. Deberá evaluar estos códigos C en ensamblador de MIPS.

a.
```
for (i=0; i<10, i++)
    a += b;
```

b.
```
While  (a < 10 ){
    D[a] = b + a;
    a + = 1;
}
```

2.18.1 [5]<2.7> Dibuje el gráfico de control de flujo del código C.

2.18.2 [5]<2.7> Utilizando el número mínimo de instrucciones, traduzca el código C a código ensamblador MIPS. Suponga que los valores `a`, `b`, `i`, y `j` están en los registros `$s0`, `$s1`, `$t0`, `$t1`, respectivamente, y que la dirección base del vector `D` está en el registro `$s2`.

2.18.3 [5]<2.7> ¿Cuántas instrucciones MIPS hay en la traducción del código C? Si el valor inicial de las variables `a` y `b` es 10 y 1, y todos los elementos de D valen 0, ¿cuántas instrucciones MIPS se ejecutan hasta completar el lazo?

La siguiente tabla contiene fragmentos de código ensamblador MIPS. Deberá evaluar cada uno de estos fragmentos y familiarizarse con las instrucciones de salto.

a.	`addi $t1, $0, 100`
	`LOOP: lw $s1, 0($s2)`
	`add $s2, $s2, $s1`
	`addi $s0, $s0, 4`
	`subi $t1, $t1, 1`
	`bne $t1, $0, LOOP`

b.	`addi $t1, $s0, 400`
	`LOOP: lw $s1, 0($s0)`
	`add $s2, $s2, $s1`
	`lw $s1, 4($s0)`
	`add $s2, $s2, $s1`
	`addi $s0, $s0, 8`
	`bne $t1, $s0, LOOP`

2.18.4 [5]<2.7> ¿Cuántas instrucciones MIPS se ejecutan?

2.18.5 [5]<2.7> Traduzca el código a C. Suponga que los enteros *i* y *result* están en los registros `$t1` y `$s2`, y que que la dirección base del entero `MemArray` está en el registro `$s0`.

2.18.6 [5]<2.7> Reescriba el código MIPS para reducir el número de instrucciones ejecutadas.

Ejercicio 2.19

La tabla muestra funciones en C. Suponga que la primera función de la tabla es la que se llama en primer lugar. Deberá traducir esta funciones a ensamblador del MIPS.

a.
```
int compare(int a, int b) {
    if (sub(a,b) >= 0)
        return 1;
    else
        return 0;
}
int sub(int a, int b){
    return a-b;
}
```

b.
```
int fib_iter(int a, int b, int n) {
    if (n == 0)
         return b;
    else
         return fib_iter(a+b, a, n-1);
}
```

2.19.1 [15]<2.8> Implemente el código de la tabla en ensamblador de MIPS. ¿Cuántas instrucciones MIPS se necesitan para ejecutar la función?

2.19.2 [5]<2.8> Los compiladores a menudo implementan las funciones *in-line*, es decir, el cuerpo de la función se copia en el espacio del programa, eliminando el sobrecoste de la llamada a la función. Implemente una versión *in-line* de la función de la tabla. ¿Cuál es la reducción del número de instrucciones MIPS necesarias para completar la función? Suponga que el valor inicial de `n` es 5.

2.19.3 [5]<2.8> Muestre el contenido de la pila después de cada llamada a la función. El valor inicial del puntero de pila es la dirección 0x7ffffffc. Siga las reglas de registros indicadas en la figura 2.11.

Los siguientes problemas hacen referencia a una función `f` que invoca a otra función `func`. La función `func` está ya compilada en otro módulo usando las reglas de invocación de MIPS de la figura 2.14. La declaración de `func` es "`int func(int a, int b);`". El código de `f` es:

a.
```
int f(int a, int b, int c) {
  return func(func(a,b),c);
}
```

b.
```
int f(int a, int b, int c) {
  return func(a,b)+func(b,c);
}
```

2.19.4 [10]<2.8> Traduzca la función `f` a ensamblador de MIPS utilizando las reglas de invocación de funciones de MIPS de la figura 2.14. Si se necesitan registros de `$t0` a `$t7`, utilice primero los de menor índice.

2.19.5 [5]<2.8> ¿Se puede usar la optimización *tail-call* en esta función? En caso negativo, explique por qué. En caso afirmativo, ¿cuál es la diferencia en el número de instrucciones de `f` con y sin optimización?

2.19.6 [5]<2.8> Justo antes de la finalización de la función `f` del problema 2.19.4, ¿qué se conoce de los contenidos de los registros `$t5`, `$s3`, `$ra` y `$sp`? Tenga en cuenta que se conoce la función `f` completa, pero de `func` sólo se conoce su declaración.

Ejercicio 2.20

Estos problemas abordan las llamadas recursivas a procedimientos. La tabla muestra un fragmento de código ensamblador que calcula el factorial de un número. Sin embargo, hay algunos errores en este fragmento que tendremos que localizar.

a.

```
FACT:  addi    $sp, $sp, -8
       sw      $ra, 4($sp)
       sw      $a0, 0($sp)
       slti    $t0, $a0, 1
       beq     $t0, $0, L1
       addi    $v0, $0, 1
       addi    $sp, $sp, 8
       jr      $ra

L1:    addi    $a0, $a0, -1
       jal     FACT
       lw      $a0, 4($sp)
       lw      $ra, 0($sp)
       addi    $sp, $sp, 8
       mul     $v0, $a0, $v0
       jr      $ra
```

b.

```
FACT:  addi    $sp, $sp, -8
       sw      $ra, 4($sp)
       sw      $a0, 0($sp)
       slti    $t0, $a0, 1
       beq     $t0, $0, L1
       addi    $v0, $0, 1
       addi    $sp, $sp, 8
       jr      $ra

L1:    addi    $t0, $t0, -1
       jal     FACT
       lw      $a0, 4($sp)
       lw      $ra, 0($sp)
       addi    $sp, $sp, 8
       mul     $v0, $a0, $v0
       jr      $ra
```

2.20.1 [5]<2.8> El código ensamblador de la tabla calcula el factorial de un número de entrada dado. La entrada entera se almacena en el registro `$a0` y el resultado se almacena en el registro `$v0`. Corrija los errores del código.

2.20.2 [10]<2.8> Reescriba el programa para que opere de forma no recursiva suponiendo que la entrada es 4. Utilice únicamente los registros `$s0` - `$s7`. Compare el número de instrucciones con los de la versión recursiva.

2.20.3 [5]<2.8> Indique el contenido de la pila después de cada llamada a la función, suponiendo que la entrada es 4.

Para los siguientes problemas, la tabla muestra un fragmento de código ensamblador que calcula un número de Fibonacci. Sin embargo, en este fragmento hay algunos errores y tendrá que localizarlos.

a.

```
FIB:  addi   $sp, $sp, -12
      sw     $ra, 0($sp)
      sw     $s1, 4($sp)
      sw     $a0, 8($sp)
      slti   $t0, $a0, 1
      beq    $t0, $0, L1
      addi   $v0, $a0, 0
      j      EXIT

L1:   addi   $a0, $a0, -1
      jal    FIB
      addi   $s1, $v0, $0
      addi   $a0, $a0, -1
      jal    FIB
      add    $v0, v0, $s1

EXIT: lw     $ra, 0($sp)
      lw     $a0, 8($sp)
      lw     $s1, 4($sp)
      addi   $sp, $sp, 12
      jr     $ra
```

b.

```
FIB:  addi   $sp, $sp, -12
      sw     $ra, 0($sp)
      sw     $s1, 4($sp)
      sw     $a0, 8($sp)
      slti   $t0, $a0, 1
      beq    $t0, $0, L1
      addi   $v0, $a0, 0
      j      EXIT

L1:   addi   $a0, $a0, -1
      jal    FIB
      addi   $s1, $v0, $0
      addi   $a0, $a0, -1
      jal    FIB
      add    $v0, v0, $s1

EXIT: lw     $ra, 0($sp)
      lw     $a0, 8($sp)
      lw     $s1, 4($sp)
      addi   $sp, $sp, 12
      jr     $ra
```

2.20.4 [5]<2.8> Este fragmento de código ensamblador calcula el Fibonacci de una entrada. La entrada entera se almacena en el registro `$a0` y el resultado se almacena en el registro `$v0`. Corrija los errores del código.

2.20.5 [10]<2.8> Reescriba el programa para que opere de forma no recursiva suponiendo que la entrada es 4. Utilice únicamente los registros `$s0` - `$s7`. Compare el número de instrucciones con las de la versión recursiva.

2.20.6 [5]<2.8> Indique el contenido de la pila después de cada llamada a la función, suponiendo que la entrada es 4.

Ejercicio 2.21

Suponga que la pila y el segmento estático de datos están vacíos y que los punteros de pila y global apuntan a las direcciones 0x7fff fffc y 0x1000 8000, respectivamente. Suponga que se usan las reglas de llamada de la figura 2.11 y que la entrada de las funciones se pasan a través de `$a0` y el valor de retorno se almacena en `$v0`. Suponga finalmente que la función `leaf` no puede utilizar registros temporales.

a.

```
main()
{
    leaf_function(1)
}
int leaf_function(int f) {
    int result;
    result=f+1;
    if (f>5)
        return result;
    leaf_function(result);
}
```

b.

```
int my_global = 100;
main()
{
   int x=10;
   int y =20;
   int z;
   z=my_function(x, my_global)
}
int my_function(int x, int y)
{
   return x-y;
}
```

2.21.1 [5]<2.8> Muestre los contenidos de la pila y del segmento estático de datos después de la llamada a cada función.

2.21.2 [5]<2.8> Escriba el código MIPS para el código C de la tabla.

2.21.3 [5]<2.8> Escriba el código MIPS para el código C de la tabla, suponiendo que la función `leaf` puede utilizar registros temporales (`$t0`, `$t1`, etc).

Para los siguientes problemas se utilizará la función escrita en ensamblador de MIPS de la tabla, que sigue las reglas de llamada de la figura 2.14.

a.
```
f: sub   $s0, $a0, $a3
   sll   $v0, $s0, 0x1
   add   $v0, $a2, $v0
   sub   $v0, $v0, $a1
   jr    $ra
```

b.
```
f: addi   $sp, $sp, 8
   sw     $ra, 4($sp)
   sw     $s0, 0($sp)
   move   $s0, $a2
   jal    g
   add    $v0, $v0, $s0
   lw     $ra, 4($sp)
   lw     $s0, 0($sp)
   addi   $sp, $sp, -8
   jr     $ra
```

2.21.4 [10]<2.8> En este código hay un error que viola las reglas de llamada de MIPS. Encuentre el error y corríjalo.

2.21.5 [10]<2.8> Traduzca el código a C. Suponga que los argumentos de las funciones del programa C se llaman `a`, `b`, `c`, etc.

2.21.6 [10]<2.8> Cuando se llama la función `f`, los registros `$a0`, `$a1`, `$a2` y `$a3` tienen los valores 1, 100, 1000 y 30, respectivamente. Determine el valor devuelto por la función. En caso de que haya una llamada a otra función `g` en `f`, suponga que el valor devuelto por `g` es siempre 500.

Ejercicio 2.22

Estos problemas estudian las conversiones a ASCII y Unicode. En la siguiente tabla se muestran algunas secuencias de caracteres.

a.	`A byte`
b.	`computer`

2.22.1 [5]<2.9> Traduzca esta secuencia a valores en bytes ASCII decimales.

2.22.2 [5]<2.9> Traduzca esta secuencia a Unicode de 16 bits (utilizando notación hexadecimal y el conjunto de caracteres Básico Latino).

La siguiente tabla muestra valores ASCII hexadecimales.

a.	`61 64 64`
b.	`73 68 69 66 74`

2.22.3 [5]<2.5, 2.9> Traduzca los valores ASCII hexadecimales a texto.

Ejercicio 2.23

En estos problemas se le pedirá que escriba un programa ensamblador MIPS que convierta secuencias de caracteres a números.

a.	**Secuencia de enteros decimales positivos**
b.	**Enteros hexadecimales en complemento a 2**

2.23.1 [10]<2.9> Escriba un programa ensamblador MIPS para convertir una secuencia de números ASCII en las condiciones mostradas en la tabla a un entero. La dirección de la secuencia de dígitos 0 a 9 está en el registro `$a0`. La secuencia termina con un `null`. El valor equivalente a esta secuencia de dígitos debe guardarse en el registros `$v0`. El programa debe poner el valor -1 en el registro `$v0` y parar si se encuentra algún carácter que no sea un dígito. Por ejemplo, si el registro `$a0` apunta a la secuencia de tres bytes 50_{diez}, 52_{diez}, 0_{diez} ("24" terminado con `null`), al final del programa el registro `$v0` debe tener el valor 24_{diez}.

Ejercicio 2.24

Suponga que el registro `$t1` tiene la dirección 0x1000 0000 y `$t2` la dirección 0x1000 0010.

a.	`lb $t0, 0($t1)` `sw $t0, 0($t2)`
b.	`lb $t0, 0($t1)` `sb $t0, 0($t2)`

2.24.1 [5]<2.9> Suponga que el contenido (hexadecimal) de la dirección 0x1000 0000 es:

1000 0000	12	34	56	78

Suponiendo que el valor inicial en la posición de memoria apuntada por `$t2` es 0xFFFF FFFF, ¿qué valor se almacena en la dirección apuntada por el registro `$t2`?

2.24.2 [5]<2.9> Suponga que el contenido (hexadecimal) de la dirección 0x1000 0000 es:

1000 0000	80	80	80	80

Suponiendo que el valor inicial en la posición de memoria apuntada por `$t2` es 0x0000 0000, ¿qué valor se almacena en la dirección apuntada por el registro `$t2`?

2.24.3 [5]<2.9> Suponga que el contenido (hexadecimal) de la dirección 0x1000 0000 es:

1000 0000	11	00	00	FF

Suponiendo que el valor inicial en la posición de memoria apuntada por `$t2` es 0x5555 5555, ¿qué valor se almacena en la dirección apuntada por el registro `$t2`?

Ejercicio 2.25

En estos problemas se estudia la utilización de constantes de 32 bits en MIPS, utilizando los valores binarios de la tabla.

a.	$1010\ 1101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0010_{dos}$
b.	$1111\ \ 1111\ 1111\ \ 1111\ 1111\ 1111\ 1111\ \ 1111_{dos}$

2.25.1 [10]<2.10> Escriba un código MIPS que cree las constantes de 32 bits de la tabla y las almacene en `$t1`.

2.25.2 [5]<2.6, 2.10> Si el valor actual del PC es 0x00000000, ¿se puede poner en el PC las direcciones de la tabla con una única instrucción de salto incondicional?

2.25.3 [5]<2.6, 2.10> Si el valor actual del PC es 0x00000600, ¿se puede poner en el PC las direcciones de la tabla con una única instrucción de salto condicional?

2.25.4 [5]<2.6, 2.10> Si el valor actual del PC es 0x00400600, ¿se puede poner en el PC las direcciones de la tabla con una única instrucción de salto condicional?

2.25.5 [10]<2.10> Suponiendo que el campo inmediato de una instrucción MIPS tiene 8 bits, escriba un código MIPS que cree las constantes de 32 bits de la tabla y las almacene en `$t1`. No utilice la instrucción `lui`.

Para los siguientes problemas, utilice el código ensamblador MIPS de la tabla.

a.	`lui $t0, 0x1234` `ori $t0, $t0, 0x5678`
b.	`ori $t0, $t0, 0x5678` `lui $t0, 0x1234`

2.25.6 [5]<2.6, 2.10> ¿Cuál es el valor en el registro `$t0` después de la secuencia de código de la tabla?

2.25.7 [5]<2.6, 2.10> Escriba el código C equivalente al código ensamblador de la tabla. Suponga que la mayor constante que se puede cargar en un registro entero de 32 bits es de 16 bits.

Ejercicio 2.26

En estos problemas deberá analizar el rango de las instrucciones de salto incondicional y condicional. Utilice los valores hexadecimales de la tabla.

a.	`0x00001000`
b.	`0xFFFC0000`

2.26.1 [10]<2.6, 2.10> Si el contenido del PC es 0x00000000, ¿cuántas instrucciones de salto condicional se necesitan para poner llegar a la dirección de la tabla?

2.26.2 [10]<2.6, 2.10> Si el contenido del PC es 0x00000000, ¿cuántas instrucciones de salto incondicional (no instrucciones de salto incondicional con registro) se necesitan para poner llegar a la dirección de la tabla?

2.26.3 [10]<2.6, 2.10> Para reducir el tamaño de los programas en ensamblador de MIPS, los diseñadores de MIPS decidieron reducir el tamaño del campo inmediato de las instrucciones tipo I de 16 a 8 bits. Si el contenido del PC es 0x00000000, ¿cuántas instrucciones de salto condicional se necesitan para poner llegar a la dirección de la tabla?

En los siguientes problemas tendrá que utilizar las modificaciones a la arquitectura del repertorio de instrucciones MIPS mostradas en la tabla.

a.	8 registros
b.	Campo de dirección/inmediato de 10 bits

2.26.4 [10]<2.6, 2.10> Si se modifica el repertorio de instrucciones del procesador, también hay que modificar el formato de las instrucciones. Para cada uno de los cambios de la tabla, ¿cuál es el impacto en el rango de direcciones de una instrucción `beq`? Suponga que las instrucciones siguen siendo de 32 bits y que cualquier cambio en la instrucciones tipo I aumentan/disminuyen solo el campo inmediato de la instrucción `beq`.

2.26.5 [10]<2.6, 2.10> Si se modifica el repertorio de instrucciones del procesador, también hay que modificar el formato de las instrucciones. Para cada uno de los cambios de la tabla, ¿cuál es el impacto en el rango de direcciones de una instrucción de salto incondicional? Suponga que las instrucciones siguen siendo de 32 bits y que cualquier cambio en la instrucciones tipo I aumentan/disminuyen solo el campo inmediato de la instrucción de salto incondicional.

2.26.6 [10]<2.6, 2.10> Si se modifica el repertorio de instrucciones del procesador, también hay que modificar el formato de las instrucciones. Para cada uno de los cambios de la tabla, ¿cuál es el impacto en el rango de direcciones de una instrucción de salto incondicional con registro? Suponga que las instrucciones siguen siendo de 32 bits.

Ejercicio 2.27

En los siguientes problemas deberá explorar los modos de direccionamiento de la arquitectura del conjunto de instrucciones MIPS mostrados en la tabla.

a.	Direccionamiento de registro
b.	Direccionamiento relativo al PC

2.27.1 [5]<2.10> En la tabla se muestran algunos modos de direccionamiento de la arquitectura del conjunto de instrucciones MIPS. Ponga un ejemplo de instrucción MIPS con el modo de direccionamiento de la tabla.

2.27.2 [5]<2.10> Para las instrucciones del problema 2.27.1, determine el tipo de formato de instrucción utilizado.

2.27.3 [5]<2.10> Indique las ventajas e inconvenientes de los modos de direccionamiento de la tabla. Escriba un código MIPS que ilustre estas ventajas e inconvenientes.

En los siguientes problemas deberá utilizar código ensamblador MIPS para explorar los pros y los contras del campo inmediato en las instrucciones tipo I.

a.	`0x00000000 lui $s0, 100` `0x00000004 ori $s0, $s0, 40`
b.	`0x00000100 addi $t0, $0, 0x0000` `0x00000104 lw $t1, 0x4000($t0)`

2.27.4 [15]<2.10> Muestre la representación a nivel de bit de las instrucciones de la tabla.

2.27.5 [10]<2.10> Si se reduce el tamaño del campo inmediato de las instrucciones tipo I y tipo J podemos reducir el número de bits necesarios para representar instrucciones. Reescriba el código MIPS de la tabla suponiendo que el campo inmediato de las instrucciones tipo I y tipo J fuese de 8 bits. No utilizar la instrucción `lui`.

2.27.6 [5]<2.10> ¿Cuántos ciclos adicionales se necesitan para ejecutar el código del problema 2.27.5 respecto al código de la tabla?

Ejercicio 2.28

A continuación se muestra el código MIPS para un bloqueo (lock).

```
try:    MOV     R3, R4
        MOV     R6, R7
        LL      R2, 0(R2)
        LL      R5, 0(R1)
        SC      R3, 0(R1)
        SC      R6, 0(R1)
        BEQZ    R3, try
        MOV     R4, R2
        MOV     R7, R5
```

2.28.1 [5]<2.11> ¿Cuántas instrucciones hay que ejecutar para cada chequeo y fallo del almacenamiento condicional?

2.28.2 [5]<2.11> Indique por qué puede fallar este código.

2.28.3 [15]<2.11> Reescriba el código para que funcione correctamente. Evite las condiciones de carrera.

La siguiente tabla tiene fragmentos de código y contenidos de registros. La notación "`($s1)`" muestra el contenido de la posición de memoria apuntada por el registro `$s1`. Las instrucciones se ejecutan en el ciclo indicado en un procesador paralelo con espacio de memoria compartida.

a.

Procesador 1	Procesador 2	Ciclo	Procesador 1		MEM	Procesador 2	
			$t1	$t0	$s1	$t1	$t0
		0	1	2	99	30	40
`ll $t1, 0($s1)`	`ll $t1, 0($s1)`	1					
`sc $t0, 0($s1)`		2					
	`sc $t0, 0($s1)`	3					

b.

Procesador 1	Procesador 2	Ciclo	Procesador 1			MEM	Procesador 2		
			$s4	$t1	$t0	$t1	$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	`try: add $t0,$0,$s4`	1							
`try: add $t0,$0,$s4`	`ll $t1, 0($s1)`	2							
`ll $t1, 0($s1)`		3							
`sc $t0, 0($s1)`		4							
`beqz $t0, try`	`sc $t0, 0($s1)`	5							
`add $s4, $0, $t1`	`beqz $t0, try`	6							

2.28.4 [5]<2.11> Rellene la tabla con el valor de los registros en cada ciclo.

Ejercicio 2.29

Los tres primeros problemas hacen referencia a una sección crítica de la forma

```
lock(lk);
operation
unlock(lk);
```

donde "operation" actualiza una variable compartida `shvar` usando una variable local (no compartida) `x`, como se muestra en la tabla.

	Operación
a.	`shvar=shvar+x;`
b.	`shvar=min(shvar,x);`

2.29.1 [10]<2.11> Escriba el código ensamblador MIPS para esta sección crítica suponiendo que la variable `lk` está en $a0, la dirección de `shvar` en `$a1` y el valor de `x` en `$a2`. La sección crítica no debe tener ninguna llamada a funciones, es decir, se deben incluir instrucciones MIPS para `lock()`, `unlock()`, `max()` y `min()`. Utilice instrucciones `ll/sc` para la implementación de `lock()`; `unlock()` es simplemente una instrucción de almacenamiento ordinaria.

2.29.2 [10]<2.11> Repita el problema 2.29.1, pero utilizando `ll/sc` para realizar una actualización atómica de `shvar` directamente, sin utilizar `lock( )` y `unlock( )`. En este caso no es necesaria la variable `lk`.

2.29.3 [10]<2.11> Compare las prestaciones del mejor-caso de los códigos de los problemas 2.29.1 y 2.29.2, suponiendo que cada instrucción necesita un ciclo para ejecutarse. Nota: mejor-caso significa que `ll/sc` siempre se ejecuta con éxito, el bloqueo está siempre disponible cuando queremos ejecutar `lock( )`, y si hay un salto se toma siempre la vía que necesita ejecutar el menor número de instrucciones.

2.29.4 [10]<2.11> Usando el código de 2.29.2 como ejemplo, explique qué ocurre cuando dos procesadores intentan acceder a la sección crítica al mismo tiempo, suponiendo que cada procesador ejecuta exactamente una instrucción por ciclo.

2.29.5 [10]<2.11> Explique por qué en el código del problema 2.29.2 el registro `$a1` tiene la dirección de la variable `shvar` y no el valor de esta variable, y el registro `$a2` contiene la variable x y no su dirección.

2.29.6 [10]<2.11> Podemos hacer la misma operación atómicamente en dos variables compartidas (`shvar1` y `shvar2`) en la misma sección crítica utilizando la aproximación de 2.29.1 (poniendo simplemente las dos actualizaciones entre las operaciones lock y su correspondiente unlock). Explique por qué no se puede hacer con la aproximación de 2.29.2, es decir, por qué no podemos utilizar `ll/sc` para acceder a ambas variables compartidas de forma que se garantice que las dos actualizaciones se van a ejecutar juntas como una única operación atómica.

Ejercicio 2.30

Las instrucciones pseudoensamblador no son parte del repertorio de instrucciones MIPS, pero aparecen frecuentemente en programas MIPS. La tabla muestra varias pseudoinstrucciones MIPS que serán traducidas a otras instrucciones MIPS.

a.	`move $t1, $t2`
b.	`beq $t1, small, LOOP`

2.30.1 [5]<2.12> Para cada pseudoinstrucción de la tabla, obtenga un código MIPS que haga la misma función. Se pueden utilizar registros temporales si es necesario. En la tabla, `large` hace referencia a un número de 32 bits y `small` a un número de 16 bits.

La tabla muestra varias pseudoinstrucciones MIPS que serán traducidas a otras instrucciones MIPS.

a.	`la $s0, v`
b.	`blt $a0, $v0, LOOP`

2.30.2 [5]<2.12> ¿Necesitan ser editadas durante la fase de enlazado. ¿Por qué?

Ejercicio 2.31

La tabla muestra algunos detalles del nivel de enlazado para dos procedimientos diferentes. En este ejercicio hará el papel del enlazador.

a.

	Procedimiento A				Procedimiento B			
Segmento de texto	**Dirección**	**Instrucción**		Segmento de texto	**Dirección**	**Instrucción**	
	0	`lw $a0, 0($gp)`			0	`sw $al, 0($gp)`	
	4	`jal 0`			4	`jal 0`	
	…	…			…	…	
Segmento de datos	0	(X)		Segmento de datos	0	(Y)	
	…	…			…	…	
Inform. de recolocación	**Dirección**	**Tipo instr.**	**Dependencia**	Inform. de recolocación	**Dirección**	**Tipo instr.**	**Dependencia**
	0	`lw`	X		0	Sw	
	4	`jai`	B		4	`jal`	
Tabla de símbolos	**Dirección**	**Símbolo**		Tabla de símbolos	**Dirección**	**Símbolo**	
	–	X			–	Y	
	–	B			–	A	

b.

	Procedimiento A				Procedimiento B			
Segmento de texto	**Dirección**	**Instrucción**		Segmento de texto	**Dirección**	**Instrucción**	
	0	`lui $at, 0`			0	`sw $a0, 0($gp)`	
	4	`ori $a0, $at 0`			4	`jmp 0`	
	8	`jal 0`			…	…	
	…	…			0x180	`jr $ra`	
					…	…	
Segmento de datos	0	(X)		Segmento de datos	0	(Y)	
	…	…			…	…	
Inform. de recolocación	**Dirección**	**Tipo instr.**	**Dependencia**	Inform. de recolocación	**Dirección**	**Tipo instr.**	**Dependencia**
	0	`lui`	X		0	`sw`	Y
	4	`ori`	X		4	`jmp`	FOO
	8	`jal`	B				
Tabla de símbolos	**Dirección**	**Símbolo**		Tabla de símbolos	**Dirección**	**Símbolo**	
	–	X			–	Y	
	–	B			0x180	FOO	

2.31.1 [5]<2.12> Enlace los ficheros objeto de la tabla para formar la cabecera de un fichero ejecutable. Suponga que el tamaño del texto y de datos del procedimiento A es 0x140 y 0x40, respectivamente, y que el tamaño del texto y de datos del procedimiento B es 0x300 y 0x50, respectivamente. Suponga también que la estrategia para colocación en memoria es la de la figura 2.13.

2.31.2 [5]<2.12> ¿Cuáles son las limitaciones en el tamaño de un ejecutable?

2.31.3 [5]<2.12> ¿Por qué podría tener problemas un ensamblador para implementar directamente las instrucciones de salto incondicional y salto condicional en el fichero objeto?

Ejercicio 2.32

En los tres primeros problemas de este ejercicio se supone que la función `swap` está definida con el siguiente código C, en lugar del código de la figura 2.24.

a.	`void swap(int v[], int k, int j){`
	`  int temp;`
	`  temp=v[k];`
	`  v[k]=v[j];`
	`  v[j]=temp;`
	`}`

b.	`void swap(int *p){`
	`  int temp;`
	`  temp=*p;`
	`  *p=*(p+1);;`
	`  *(p+1)=*p;`
	`}`

2.32.1 [10]<2.13> Traduzca esta función a código ensamblador de MIPS.

2.32.2 [5]<2.13> ¿Qué hay que cambiar en la función `sort`?

2.32.3 [5]<2.13> Si estuviese ordenando bytes de 8 bits en lugar de palabras de 32 bits, ¿en qué cambiaría el código ensamblador de MIPS del problema 2.32.1?

Para los tres problemas restantes se supone que la función `sort` de la figura 2.27 se cambia de la siguiente forma:

a.	Usar registros s en lugar de registros t
b.	Usar la instrucción bltz (branch on less than zero) en lugar de slt y bne en la etiqueta `for2tst`

2.32.4 [5]<2.13> ¿Este cambio afecta al código para almacenamiento y recuperación de registros de la figura 2.27?

2.32.5 [10]<2.13> Al ordenar un vector de 10 elementos que ya ha sido ordenado, ¿cuántas instrucciones de más (o de menos) se ejecutan como resultado de este cambio?

2.32.6 [10]<2.13> Al ordenar un vector de 10 elementos que ya ha sido ordenado en orden descendente (contrario al orden creado por `sort( )`), ¿cuántas instrucciones de más (o de menos) se ejecutan como resultado de este cambio?

Ejercicio 2.33

Los problemas de este ejercicio hacen uso de la siguiente función:

a.

```
int find(int a[], int n, int x){
   int i;
   for (i=0; i!=n; i++)
      if (a[i]==x)
         return i;
   return -1;
}
```

b.

```
int count(int a[], int n, int x){
   int res=0;
   int i;
   for (i=0; i!=n; i++)
      if (a[i]==x)
         res=res+1;
   return res;
}
```

2.33.1 [10]<2.14> Traduzca esta función a código ensamblador de MIPS.

2.33.2 [10]<2.14> Traduzca esta función a código C basado en punteros.

2.33.3 [10]<2.14> Traduzca tu código C basado en punteros a código ensamblador de MIPS.

2.33.4 [5]<2.14> Compare el peor-caso en el número de instrucciones ejecutadas en las iteraciones que no son las últimas del lazo del código del problema 2.33.1 con el del código basado en punteros del problema 2.33.2. Nota: se considera el peor-caso como aquel en el que se toma la salida más larga en las instrucciones de salto condicional; por ejemplo, en caso de una sentencia if, el resultado de la comprobación de la condición es tal que se toma el camino con más instrucciones. Sin embargo, si el resultado de la comprobación de la condición llevase a salir del lazo, entonces se considera que se toma la salida que nos mantiene en el lazo.

2.33.5 [5]<2.14> Compare el número de registros temporales (registros t) que se necesitan en el código del problema 2.33.1 con los que se necesitan en el código basado en punteros de 2.33.3.

2.33.6 [5]<2.14> Si los registros `$t0-$t7` y `$a0-$a7` en las reglas de llamada de MIPS fuesen guardados por el llamado como los registros `$s0 - $s7`, ¿qué cambiaría en la respuesta del problema 2.33.4?

Ejercicio 2.34

La siguiente tabla muestra código ensamblador de ARM. En los siguientes problemas tendremos que traducirlo a código MIPS.

a.	`MOV r0, #10 ; iniciar el contador del lazo a 10`
	`LOOP: ADD r0, r1 ; sumar r1 a r0`
	`SUBS r0, 1 ; decrementar el contador`
	`BNE LOOP ; se repite el lazo si Z=0`

b.	`ROR r1, r2, #4 ; r1 = r23:0 concatenado con r231:4`

2.34.1 [5]<2.16> Traduzca este código ensamblador de ARM a código ensamblador de MIPS. Suponga que los registros r0, r1 y r2 de ARM tienen los mismos valores que los registros `$s0`, `$s1` y `$s2` de MIPS. Utilice registros temporales de MIPS (`$t0`, etc.) si es necesario.

2.34.2 [5]<2.16> Muestre los campos de bits que representan las instrucciones ARM para las instrucciones de la tabla.

La siguiente tabla muestra código ensamblador de MIPS. En los siguientes problemas tendremos que traducirlo a código de ARM.

a.	`slt $t0, $s0, $s1` `blt $t0, 0, FARAWAY`
b.	`add $s0, $s1, $s2`

2.34.3 [5]<2.16> Traduzca este código a código ensamblador de ARM.

2.34.4 [5]<2.16> Muestre los campos de bits que representan el código ensamblador de ARM.

Ejercicio 2.35

El procesador ARM tiene unos modos de direccionamiento que no están soportados en el MIPS. Los siguientes problemas exploran estos modos de direccionamiento.

a.	`LDR r0, [r1] ; r0=memoria[r1]`
b.	`LDMIA r0, [r1, r2, r4] ; r1=memoria[r0], r2=memoria[r0+4],` `; r4=memoria[r0+8]`

2.35.1 [5]<2.16> Identifique el tipo de modo de direccionamiento de las instrucciones ensamblador de ARM de la tabla.

2.35.2 [5]<2.16> Para las instrucciones ensamblador de ARM de la tabla, escriba una secuencia de instrucciones ensamblador de MIPS que realicen la misma transferencia de datos.

En los siguientes problemas se comparan códigos escritos con los conjuntos de instrucciones de ARM y de MIPS. La tabla muestra fragmentos de códigos escritos con instrucciones de ARM.

a.	`LDR r0, =Table1 ; carga dirección base de una tabla`
	`LDR r1, #100 ; inicia el contador del lazo`
	`EOR r2, r2, r2 ; borra r2`
	`ADDLP: LDR r4, [r0] ; accede al primer operando`
	`ADD r2, r2, r4 ; suma a r2`
	`ADD r0, r0, #4 ; incremento para siguiente elemento de la tabla`
	`SUBS r1, r1, #1 ; decrementa el contador del lazo`
	`BNE ADDLP ; si contador de lazo !=0, salto a ADDLP`

b.	`ROR r1, r2, #4 ; r1 = r23:0 concatenado con r231:4`

2.35.3 [10]<2.16> Escriba una rutina en código ensamblador de MIPS equivalente al código ensamblador ARM de la tabla.

2.35.4 [5]<2.16> ¿Cuál es el número total de instrucciones ensamblador de ARM del código? ¿Cuál es el número total de instrucciones ensamblador de MIPS?

2.35.5 [5]<2.16> Suponga que el CPI medio de la rutina MIPS es igual al CPI medio de la rutina ARM, y que frecuencia del procesador MIPS es 1.5 veces la frecuencia del procesador ARM. Determine cuántas veces es más rápido el procesador ARM que el MIPS.

Ejercicio 2.36

El procesador ARM tiene una forma interesante de incluir constantes inmediatas. En este ejercicio se investigan estas diferencias. La tabla muestra instrucciones ARM.

a.	`ADD r3, r2, r1, LSL #3 ; r3=r2 + (r1 << 3)`
b.	`ADD r3, r2, r1, ROR #3 ; r3=r2 + (r1, rotado 3 bits a la derecha)`

2.36.1 [5]<2.16> Escriba un código MIPS equivalente al código ARM de la tabla.

2.36.2 [5]<2.16> Suponiendo que el contenido del registro R1 es 8, reescriba el código MIPS para minimizar el número de instrucciones ensamblador MIPS.

2.36.3 [5]<2.16> Suponiendo que el contenido del registro R1 es 0x06000000, reescriba el código MIPS para minimizar el número de instrucciones ensamblador MIPS.

La siguiente tabla tiene instrucciones MIPS.

a.	`addi $r3, r2, 0x1`
b.	`addi $r3, r2, 0x8000`

2.36.4 [5]<2.16> Escriba un código ARM equivalente al código MIPS de la tabla.

Ejercicio 2.37

Este ejercicio explora las diferencias entre los conjuntos de instrucciones MIPS y x86. La tabla contiene código ensamblador x86.

a.	`mov edx, [esi+4*ebx]`

b.	`START: mov ax, 00101100b`
	`mov cx, 00000011b`
	`mov bx, 11110000b`
	`and ax, bx`
	`or ax, cx`

2.37.1 [10]<2.17> Escriba un pseudocódigo para la rutina de la tabla.

2.37.2 [10]<2.17> Escriba un código MIPS equivalente a la rutina de la tabla.

La siguiente tabla contiene código ensamblador x86.

a.	`mov edx, [esi+4*ebx]`
b.	`mov eax, 0x12345678`

2.37.3 [5]<2.17> Indique el tamaño de los campos de bits que representan cada una de las instrucciones de la tabla. Suponga que la etiqueta `MY_FUNCTION` es una constante de 32 bits.

2.37.4 [10]<2.17> Escriba instrucciones MIPS equivalentes.

Ejercicio 2.38

El repertorio de instrucciones x86 incluye el prefijo REP, que hace que la instrucción se repita un número dado de veces o hasta que se satisfaga una condición. Los tres primeros problemas utilizan las siguientes instrucciones x86:

	Instrucción	Interpretación
a.	`REP MOVSB`	`Repetir hasta que ECX =0:` `Mem8[EDI]=Mem8[ESI], EDI=EDI+1, ESI=ESI+1, ECX=ECX-1`
b.	`REP MOVSD`	`Repetir hasta que ECX =0:` `Mem32[EDI]=Mem32[ESI], EDI=EDI+4, ESI=ESI+4, ECX=ECX-1`

2.38.1 [5]<2.17> ¿Cuál es la utilización típica de esta instrucción?

2.38.2 [5]<2.17> Escriba un código MIPS equivalente al de la tabla, suponiendo que `$a0`, `$a1`, `$a2` y `$a3` corresponden a ECX, EDI, ESI y EAX, respectivamente.

2.38.3 [5]<2.17> Si la instrucción x86 necesita un ciclo para leer la memoria, un ciclo para escribir en memoria y un ciclo para cada actualización de registro, y si MIPS necesita un ciclo por instrucción, ¿cuál es la aceleración al utilizar la instrucción x86 en lugar del código MIPS equivalente si ECX es muy grande? Suponga que los ciclo de reloj de MIPS y del x86 son iguales.

Los siguientes tres problemas de este ejercicio trabajan con la siguiente función, dada en C y en ensamblador de x86. Para cada instrucción x86 se muestra su longitud en el formato de instrucciones de longitud variable y la interpretación (lo que hace la instrucción). Obsérvese que la arquitectura x86 tiene muy pocos registros en comparación con la arquitectura MIPS y, en consecuencia, las reglas de llamada de x86 ponen todos los argumentos en la pila. El valor de retorno se devuelve al que hizo la llamada en el registro EAX.

	Código C	Código x86

a.	`int f(int a, int b){`
	`  return a+b;`
	`}`
		`f: push %ebp ; 1B, lleva %ebp a la pila`
		`mov %esp, %ebp ; 2B, mueve %esp a %ebp`
		`mov 0xc(%ebp),%eax ; 3B, carga 2º argumento en %aex`
		`add 0x8(%ebp),%eax ; 3B, suma 1º argumento a %eax`
		`pop %ebp ; 1B, restaura %ebp`
		`ret ; vuelta de la rutina`

b.	`void f(int*a, int *b){`
	`  *a=*a+*b;`
	`  *b=*a;`
	`}`
		`f: push %ebp ; 1B, lleva %ebp a la pila`
		`mov %esp, %ebp ; 2B, mueve %esp a %ebp`
		`mov 8(%ebp), %eax ; 3B, carga 1º argumento en %aex`
		`mov 12(%ebp), %ecx ; 3B, carga 2º argumento en %ecx`
		`mov (%eax), %edx ; 2B, carga *a en %edx`
		`add (%ecx), %edx ; 2B, suma *b a %edx`
		`mov %edx, (%eax) ; 2B, almacena %edx en *a`
		`mov %edx, (%ecx) ; 2B, almacena %edx en *b`
		`pop %ebp ; 1B, restaura %ebp`
		`ret ; vuelta de la rutina`

2.38.4 [5]<2.17> Traduzca esta función a ensamblador de MIPS. Compare el tamaño (número de bytes de memoria de instrucciones) del código MIPS con el código x86.

2.38.5 [5]<2.17> Si el procesador tiene la capacidad de ejecutar dos instrucciones por ciclo, debe ser capaz de leer dos instrucciones consecutivas en cada ciclo. Explique cómo se puede hacer esto en MIPS y en x86.

2.38.6 [5]<2.17> Si cada instrucción MIPS necesita un ciclo para ejecutarse y cada instrucción x86 un ciclo más un ciclo adicional por cada lectura o escritura de memoria que tenga que hacer en la instrucción, ¿cuál es la aceleración al utilizar x86 en lugar de MIPS? Suponga que el ciclo del reloj es el mismo en ambos casos y que la ejecución se realiza por la vía más corta de la función (es decir, en cada lazo se hace una única iteración y la salida de cada sentencia if es la que lleva hacia el retorno de la función). Observe que la instrucción ret del x86 lee la dirección de retorno de la pila.

Ejercicio 2.39

La siguiente tabla muestra el CPI de varios tipos de instrucciones.

	Aritméticas	Carga/almacenamiento	Salto
a.	2	10	3
b.	1	10	4

2.39.1 [5]<2.18> Suponga un programa con la siguiente cuenta de instrucciones:

	Instrucciones (en millones)
Aritméticas	500
Carga/almacenamiento	300
Salto	100

Determine el tiempo de ejecución en un procesador con los CPI de la tabla y una frecuencia de 1.5 GHz.

2.39.2 [5]<2.18> Suponga que se añaden nuevas y más potentes instrucciones aritméticas al conjunto de instrucciones. El uso de estas nuevas instrucciones nos permite reducir el número de instrucciones aritméticas del programa en un 25%, a costa de un incremento del 10% en el tiempo de ciclo de la señal de reloj. ¿Es una buena decisión de diseño? ¿Por qué?

2.39.3 [5]<2.18> Suponga que se encuentra una forma de duplicar las prestaciones de las instrucciones aritméticas. ¿Cuál es la aceleración que se obtiene para nuestro procesador? ¿Qué sucede si se encuentra una forma de mejorar las prestaciones de las instrucciones aritméticas en un factor 10?

La siguiente tabla muestra el porcentaje de ejecución de instrucciones.

	Aritméticas	Carga/almacenamiento	Salto
a.	60%	20%	20%
b.	80%	15%	5%

2.39.4 [5]<2.18> Dados los valores de la tabla y suponiendo que una instrucción aritmética tarda 2 ciclos, una de carga/almacenamiento 6 ciclos y una de salto 3 ciclos, determine el CPI medio.

2.39.5 [5]<2.18> Si queremos mejorar las prestaciones un 25%, ¿cuántos ciclos debería tardar las instrucciones aritméticas, en media, si no se mejoran ni las instrucciones de salto ni las de carga/almacenamiento?

2.39.6 [5]<2.18> Si queremos mejorar las prestaciones un 50%, ¿cuántos ciclos debería tardar las instrucciones aritméticas, en media, si no se mejoran ni las instrucciones de salto ni las de carga/almacenamiento?

Ejercicio 2.40

Los tres primeros problemas de este ejercicio utilizan la siguiente función ensamblador MIPS. Desafortunadamente, el programador ha cometido el error de suponer que MIPS se basa en el direccionamiento de palabras, cuando en realidad se basa en direccionamiento de bytes.

a.

```
: int f(int a[], int n, int x);
f:  move  $v0, $zero      ; ret=0
    move  $t0, $zero      ; i=0
L:  add   $t1, $t0, $a0   ; &(a[i])
    lw    $t1, 0($t1)     ; read a[i]
    bne   $t1, $a2, S     ; if (a[i]==x)
    addi  $v0, $v0, 1     ; ret++
S:  addi  $t0, $t0, 1     ; i++
    bne   $t0, $a1, L     ; repeat if i!=n
    jr    $ra             ; return ret
```

b.

```
: void f(int *a, int *b, int n);
f: move  $t0, $a0          ; p=a
   move  $t1, $a1          ; q=b
   add   $t2, $a2, $a0     ; &(a[n])
L: lw    $t3, 0($t0)       ; read *p
   lw    $t4, 0($t1)       ; read *q
   add   $t3, $t3, $t4     ; *p+*q
   sw    $t3, 0($t0)       ; *p=*p+*q
   addi  $t0, $t0, 1       ; p=p+1
   addi  $t1, $t1, 1       ; rq=q+1
   bne   $t0, $t2, L       ; repeat if p!=&(a[n]
   jr    $ra               ; return
```

Observe que ";" significa que el resto de la línea es un comentario.

2.40.1 [5]<2.18> Para que los accesos a palabras de memoria en la arquitectura MIPS (instrucciones sw y lw) se ejecuten correctamente las direcciones deben estar alineadas, es decir, los dos bits menos significativos de la dirección deben ser cero. En caso contrario, se produce una excepción "bus error". Explique cómo afecta este requerimiento a la ejecución de la función de la tabla.

2.40.2 [5]<2.18> Si "a" es un puntero que apunta al comienzo de un vector de elementos de 1 byte y si reeplazamos lw y sw por lb (carga de un byte) y sb (almacenamiento de un byte), respectivamente, ¿funcionaría correctamente la función? Nota: lb lee un byte de memoria, hace extensión de signo y lo carga en el registro destino; sb almacena el byte menos significativo del registro en memoria.

2.40.3 [5]<2.18> Cambie el código para que funcione correctamente con enteros de 32 bits.

Los siguientes tres problemas hacen uso de un programa que reserva memoria para un vector, da valores a los elementos del vector, llama a la función sort de la figura 2.27 e imprime en pantalla el vector. La función principal del programa es la siguiente (en C y código MIPS).

Código principal en C	Versión MIPS del código principal

`main(){`
`  int *v;`
`  int n=5;`
`  v=my_alloc(5);`
`  my_init(v,n);`
`sort(v,n);`
`.`
`.`
`.`
	`main:`
	`li  $s0,5`
	`move  $a0, $s0`
	`jal  my_alloc`
	`move  $s1, $v0`
	`move  $a0, $s1`
	`move  $a1, $s0`
	`jal  my_init`
	`move  $a0, $s1`
	`move  $a1, $s0`
	`jal  sort`

La función `my_alloc` se define en la tabla. Observe que el programador ha cometido el error de utilizar un puntero a una variable automática `arr` fuera de la función en la que se ha definido.

`my_alloc` **en C**	**Código MIPS para** `my_alloc`

`int *my_alloc(int n){`
`  int arr[n];`
`  return arr;`
`}`
	`my_alloc:`
	`  addu  $sp, $sp, -4  ; puntero de pila`
	`  sw  $fp, 0($sp)  ; $fp a la pila`
	`  move  $fp, $sp  ; guarda $sp en $fp`
	`  sll  $t0, $a0, 2  ; se necesitan 4*n bytes`
	`  sub  $sp, $sp, $t0 ; espacio para arr`
	`  move  $v0, $sp  ; dirección de retorno de arr`
	`  move  $sp, $fp  ; restaura $sp desde $fp`
	`  lw  $fp, 0($sp)  ; Extrae $fp`
	`  addiu $sp, $sp, 4  ; de la pila`
	`  jr  $ra`

A continuación se muestra la función `my_init` en código MIPS.

a.

```
my_init:
   move    $t0, $zero     ; i=0
   move    $t1, $a0
L: sw      $zero, 0($t1)  ; v[i]=0
   addiu   $t1, $t1, 4
   addiu   $t0, $t0, 1    ; i=i+1
   bne     $t0, $a1, L    ; until i==n
   jr      $ra
```

b.

```
my_init:
   move    $t0, $zero     ; i=0
   move    $t1, $a0
L: sub     $t2, $a1, $t0
   sw      $t2, 0($t1)    ; a[i]=n-i
   addiu   $t1, $t1, 4
   addiu   $t0, $t0, 1    ; i=i+1
   bne     $t0, $a1, L    ; until i==n
   jr      $ra
```

2.40.4 [5]<2.18> ¿Cuál es el contenido (valores de los cinco elementos) del vector `v` antes de la ejecución de la instrucción "`jal sort`" en el código principal?

2.40.5 [5]<2.18, 2.13> ¿Cuál es el contenido del vector `v` antes de que se entre por primera vez en el lazo más externo de la función `sort`? Suponga que los valores en los registros `$sp`, `$s0`, `$s1`, `$s2` y `$s3` son 0x10000, 20, 40, 7 y 1, respectivamente, al comienzo del código principal (antes de la ejecución de "`li $s0, 5`").

2.40.6 [5]<2.18, 2.13> ¿Cuál es el contenido del vector de 5 elementos apuntado por `v` justo después de que "`jal sort`" devuelva el control al código principal?

Respuestas a las autoevaluaciones

$2.2, página 80: MIPS, C, Java.
$2.3, página 87: 2) muy lento.
$2.4, página 93: 3) -8_{diez}.
$2.5, página 101: 4) `sub $s2, $s0, $s1`.
$2.6, página 104: Ambos. AND con una máscara de unos pondrá ceros en todas parte excepto en el campo deseado e indicado por la máscara. Desplazando a la izquierda el número de posiciones correcto elimina los bits a la izquierda del campo. Desplazando a la derecha el número de posiciones correcto sitúa el campo en los bits más a derecha de la palabra. Observe que la operación AND deja el campo donde ya estaba originalmente, y que los desplazamientos trasladan el campo a los bits más a la derecha de la palabra.
$2.7, página 111: I. Todos son correctos. II. 1).
$2.8, página 122: Ambos son verdad.
$2.9, página 127: I. 2) II. 3).
$2.10, página 136: I. 4) +-128K. II. 6) un bloque de 256M. III.4) `sll`.
$2.11, página 139: Ambos son verdad.
$2.12, página 148: 4) Independencia de la máquina.

3

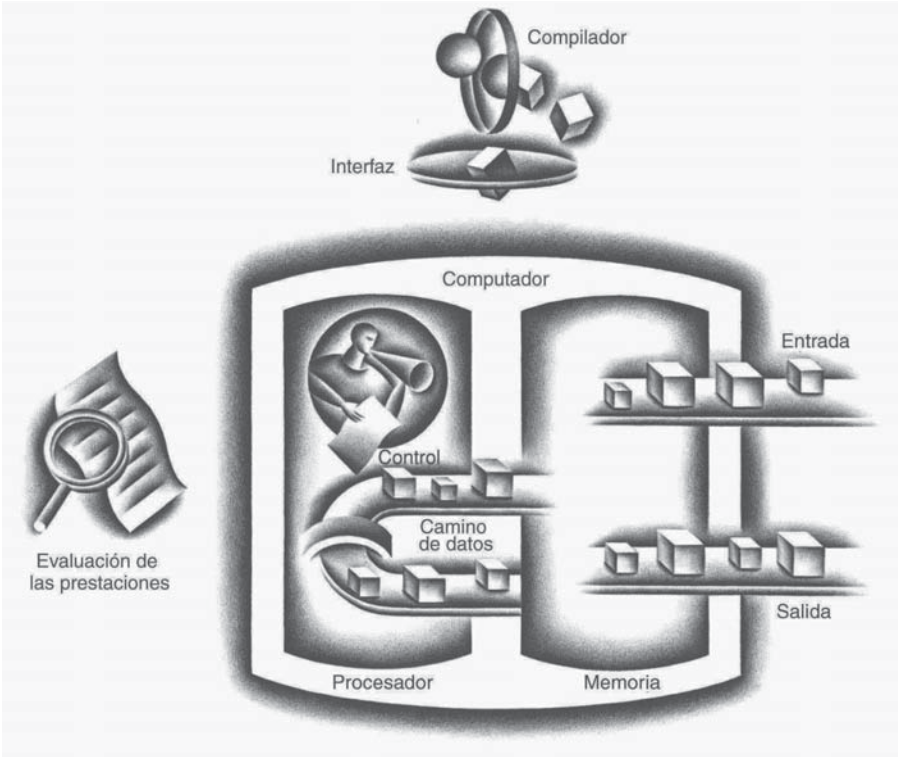
Aritmética para computadores

La precisión numérica es el alma verdadera de la ciencia

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco tipos de componentes de un computador



3.1 Introducción

Las palabras del computador están formadas por bits; de este modo, las palabras se pueden representar como números binarios. Aunque los números naturales 0, 1, 2, … se pueden representar tanto en forma decimal como binaria, ¿qué podemos decir de los otros números que usamos comúnmente? Por ejemplo:

- ¿Qué podemos decir de las fracciones y los números reales?
- ¿Qué pasa si una operación genera un número más grande de lo que se puede representar?
- Y detrás de todas estas cuestiones hay un misterio: ¿cómo multiplica o divide números el hardware?

El objetivo de este capítulo es desvelar este misterio, incluyendo la representación de números, los algoritmos aritméticos, el hardware que implementa estos algoritmos y las implicaciones de todo ello en los repertorios de instrucciones. Estas revelaciones pueden incluso explicar peculiaridades de los computadores con las que ya haya tropezado.

Resta: la compañera tramposa de la suma

DAVID LETTERMAN et al., Núm.10, Los diez mejores cursos para atletas en una factoría de fútbol, *Book of Top Ten Lists*, 1990.

3.2 Suma y resta

La suma es justo lo que se esperaría de ella en los computadores. Los dígitos se suman bit a bit de derecha a izquierda, pasando los arrastres o acarreos (*carries*) al próximo dígito de la izquierda, tal y como se hace a mano. La resta usa la suma: el operando apropiado simplemente se niega antes de ser sumado.

Suma y resta binaria

EJEMPLO

Sumemos 6_{diez} a 7_{diez} en binario y luego restemos 6_{diez} de 7_{diez} en binario.

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{dos} = 7_{diez} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{dos} = 6_{diez} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{dos} = 13_{diez} \end{array}$$

Sobre los cuatro bits de la derecha recae toda la acción; la figura 3.1 muestra las sumas y los acarreos. Los acarreos se muestran entre paréntesis, con flechas que muestran cómo se pasan.

Restar 6_{diez} de 7_{diez} se puede hacer directamente:

RESPUESTA

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{dos} & = 7_{diez} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{dos} & = 6_{diez} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dos} & = 1_{diez} \end{array}$$

o vía suma, usando la representación en complemento a dos de -6_{diez}:

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{dos} & = 7_{diez} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{dos} & = -6_{diez} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{dos} & = 1_{diez} \end{array}$$

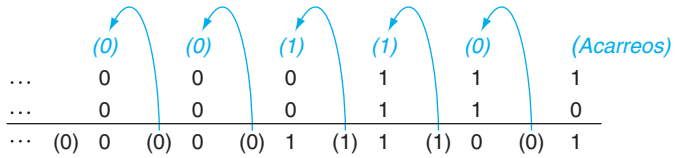


FIGURA 3.1 Suma binaria, mostrando los acarreos de derecha a izquierda. El bit situado más a la derecha suma 1 a 0, cuyo resultado es 1 y el acarreo producido por este bit es 0. Por tanto, la operación para el segundo bit de la derecha es 0 + 1 + 1. Esto genera un 0 para este bit de suma y un acarreo de 1. El tercer bit es la suma de 1 + 1 + 1, resultando un acarreo de 1 y un bit de suma de 1. El cuarto bit es 1 + 0 + 0, lo que da un 1 de suma sin acarreo.

Recordemos que el desbordamiento ocurre cuando el resultado de una operación no puede ser representado con el hardware disponible, en este caso una palabra de 32 bits. ¿Cuándo puede producirse desbordamiento en la suma? Cuando se suman operandos de signo diferente no puede producirse desbordamiento. La razón es que la suma no puede ser mayor que uno de los operandos. Por ejemplo, –10 + 4 = –6. Puesto que los operandos caben en 32 bits y la suma no es mayor que ninguno de los operandos, la suma debe caber en 32 bits también. Por lo tanto, no puede producirse desbordamiento cuando se suman operandos positivos y negativos.

Existen restricciones similares para la aparición de desbordamiento en la resta, pero es justo el principio contrario: cuando el signo de los operandos es el *mismo*, no se puede producir desbordamiento. Para verlo, recuerde que $x - y = x + (-y)$, porque restamos negando el segundo operando y entonces sumamos. Así, cuando restamos operandos del mismo signo, al final acabamos *sumando* operandos de signo *diferente*. Del párrafo anterior, sabemos que en este caso tampoco se puede producir desbordamiento.

Aunque hemos visto cuándo no se puede producir desbordamiento en la suma y la resta, aún no hemos respondido a la pregunta de cómo detectarlo cuando sí ocurre. Claramente, la suma o resta de dos números de 32 bits puede dar un resul-

tado que necesite 33 bits para ser representado completamente. La falta del bit 33º bit significa que cuando ocurre desbordamiento el bit de signo toma el *valor* del resultado en lugar del signo correcto del resultado. Puesto que necesitamos justo un bit extra, sólo el bit de signo puede estar incorrecto. El desbordamiento se produce cuando se suman dos números positivos y la suma es negativa, o viceversa. Esto significa que se ha producido un acarreo saliente en el bit de signo.

El desbordamiento se produce en la resta cuando restamos un número negativo de un número positivo y obtenemos un resultado negativo, o cuando restamos un número positivo de uno negativo y obtenemos un resultado positivo. Esto significa que se ha producido un acarreo en el bit de signo. La figura 3.2 muestra las combinaciones de las operaciones, operandos y resultados que producen desbordamiento.

Acabamos de ver cómo se detecta el desbordamiento para números en complemento a dos en un computador. ¿Qué pasa con los números sin signo? Los números sin signo se usan comúnmente para direcciones de memoria donde los desbordamientos se ignoran.

El diseñador del computador debe, por tanto, proporcionar una manera de ignorar el desbordamiento en algunos casos y reconocerlo en otros. La solución de MIPS es tener dos tipos de instrucciones aritméticas para reconocer las dos alternativas:

- Suma (`add`), suma inmediato (`addi`) y resta (`sub`) provocan excepciones cuando hay desbordamiento.
- Suma sin signo (`addu`), suma inmediato sin signo (`addiu`) y resta sin signo (`subu`) *no* causan excepciones cuando hay desbordamiento.

Puesto que C ignora los desbordamientos, los compiladores de C de MIPS siempre generarán la versión sin signo de las instrucciones aritméticas `addu`, `addiu` y `subu` sin importar el tipo de las variables. Los compiladores Fortran de MIPS, sin embargo, escogen las instrucciones aritméticas apropiadas, dependiendo del tipo de los operandos.

Operación	Operando A	Operando B	Resultado que indica desbordamiento
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURA 3.2 Condiciones de desbordamiento para la suma y la resta.

Unidad Aritmética y Lógica (ALU): hardware para hacer sumas y restas y habitualmente operaciones lógicas como AND y OR.

El  **apéndice C** describe el hardware para realizar sumas y restas. Recibe el nombre de **Unidad Aritmética y Lógica, ALU**.

El diseñador del computador debe decidir cómo manejar los desbordamientos aritméticos. Aunque algunos lenguajes como C ignoran los desbordamientos entre enteros, lenguajes como Ada y Fortran requieren que se le notifique al programa el desbordamiento. El programador o el entorno de programación deben decidir qué hacer cuando ocurre desbordamiento.

MIPS detecta el desbordamiento mediante una **excepción**, también llamada **interrupción** en muchos computadores. Una excepción o interrupción es esencialmente una llamada a procedimiento no prevista. La dirección de la instrucción que produjo desbordamiento se guarda en un registro, y el computador salta a una dirección predefinida para invocar a la rutina apropiada para esta excepción. La dirección interrumpida se guarda para que en algunas situaciones el programa pueda continuar después de ejecutar cierto código de correción. (La sección 4.9 cubre las excepciones con más detalle; los capítulos 5 y 6 describen otras situaciones donde pueden ocurrir excepciones e interrupciones.)

MIPS incluye un registro llamado *contador de programa de excepción* (*exception program counter*, EPC) para almacenar la dirección de la instrucción que ha causado la excepción. La instrucción *mover desde el control del sistema* (*move from system control,* `mfc0`) se usa para copiar el EPC a un registro de propósito general para que el software MIPS tenga la opción de volver a la instrucción que produjo la excepción con una instrucción de salto a través de registro.

Interfaz hardware software

Excepción (también llamada interrupción): evento no previsto que interrumpe la ejecución del programa; se usa para detectar desbordamiento.

Interrupción: excepción que viene del exterior del procesador. (Algunas arquitecturas usan el término interrupción para todas las excepciones).

Aritmética para multimedia

Dado que todos los microprocesadores de computadores de sobremesa tienen su propio monitor gráfico, y que la disponibilidad de transistores ha aumentado, ha sido inevitable la incorporación hardware de apoyo para operaciones gráficas.

En muchos sistemas, se utilizaron originalmente 8 bits para representar cada uno de los tres colores básicos más otros 8 bits para la posición del píxel. Posteriormente, con la utilización de micrófonos y altavoces para teleconferencia y videojuegos se pensó en añadir también apoyo para el sonido. Para audio se necesitan más de 8 bits, pero con16 bits es suficiente.

Todos los microprocesadores incluyen apoyo para que los datos tipo byte y media palabra ocupen menos espacio en memoria (véase sección 2.9), pero puesto que las operaciones aritméticas con datos de este tamaño en programas con enteros son muy infrecuentes, el apoyo disponible no va más allá de la transferencia de datos. Los arquitectos se dieron cuenta de que muchas aplicaciones de gráficos y audio podrían hacer las operaciones sobre vectores de datos de esos tamaños. Un procesador podría hacer operaciones simultáneas sobre vectores de ocho operandos de 8 bits, cuatro operandos de 16 bits o dos operandos de 32 bits, simplemente rompiendo la cadena de propagación del acarreo de 64 bits. El coste de esta partición es pequeño. Estas extensiones se llamaron extensiones vectoriales o SIMD, de instrucción única, múltiples datos (*single instruction, multiple data*) (véase la sección 2.17 y el capítulo 7).

Otra característica, que no está disponible en los procesadores de propósito general, son las operaciones con *saturación*. Saturación quiere decir que cuando se produce desbordamiento en una operación, se fuerza a que el resultado sea el mayor número positivo o el menor número negativo (el más negativo), en lugar del resultado de una operación módulo como en complemento a 2. La saturación es necesaria en operaciones de audio o gráficos. Por ejemplo, la utilización del mando de volumen en una radio sería muy frustrante si cuando estamos subiendo volumen llega un momento en que éste desciende repentinamente a un nivel muy bajo. Un mando de volumen con saturación pararía al nivel máximo aunque siguiésemos intentando subirlo con el mando. En la figura 3.3 se muestran las operaciones aritmética y lógicas habituales en muchas extensiones multimedia de los repertorios de instrucciones modernos.

Categoría de la instrucción	Operandos
Suma/resta sin signo	Ocho de 8 bits o cuatro de 16 bits
Suma/resta con saturación	Ocho de 8 bits o cuatro de 16 bits
Max/min/mínimo	Ocho de 8 bits o cuatro de 16 bits
Media	Ocho de 8 bits o cuatro de 16 bits
Desplazamiento a la derecha/izquierda	Ocho de 8 bits o cuatro de 16 bits

FIGURA 3.3 Resumen del apoyo multimedia para computadores de sobremesa.

Extensión: MIPS puede saltar al producirse desbordamiento, pero a diferencia de muchos otros computadores no hay saltos condicionales para detectar el desbordamiento. Una secuencia de instrucciones MIPS puede descubrirlo. Para la suma con signo, la secuencia es la siguiente (véase el apartado *Extensión* sobre las instrucciones lógicas del capítulo 2 para la definición de las instrucciones `xor`):

```
addu   $t0, $t1,   $t2              # $t0 = suma, pero sin detectar
xor    $t3, $t1,   $t2              # comprueba si los signos son distintos
slt    $t3, $t3,   $zero            # $t3 = 1 si los signos difieren
bne    $t3, $zero, No_desbordamiento # los signos de $t1 y $t2 ≠, no hay desbordamiento
xor    $t3, $t0,   $t1              # los signos son iguales, ¿el de la suma también?
                                    # $t3 negativo si el signo de la suma es distinto
slt    $t3, $t3,   $zero            # $t3 = 1 si el signo de la suma es diferente
bne    $t3, $zero, Desbordamiento   # los tres signos ≠; saltar a desbordamiento
```

Para la suma sin signo (`$t0 = $t1 + $t2`), la comprobación es

```
addu   $t0, $t1,   $t2                 #  $t0 = suma
nor    $t3, $t1,   $zero               #  $t3 = NOT $t1
                                       #  (comp a 2 - 1: 2^32 - $t1 - 1)
sltu   $t3, $t3    $t2                 #  (2^32 - $t1 - 1) < $t2
                                       #  ⇒ 2^32 - 1 < $t1 + $t2
bne    $t3  $zero, Desbordamiento      #  si(2^32-1 < $t1 + $t2) ir a desbordamiento
```

Resumen

El punto principal de esta sección es que, independientemente de la representación, el tamaño finito de palabra de los computadores significa que las operaciones aritméticas pueden producir resultados que son demasiado largos para caber en ese tamaño de palabra fijo. Es fácil detectar desbordamiento en números sin signo, aunque éstos se ignoran casi siempre porque los programas no necesitan detectar desbordamiento en la aritmética de direcciones, el uso más común de los números naturales. El complemento a dos presenta un reto mayor, pero dado que algunos sistemas software requieren la detección de desbordamiento, hoy en día todos los computadores tienen una manera de detectarlo.

La creciente popularidad de las aplicaciones multimedia ha llevado a instrucciones aritméticas que utilizan operaciones más estrechas (con menos bits) que pueden operar fácilmente en paralelo.

Autoevaluación

Algunos lenguajes de programación permiten aritmética entera en complemento a dos sobre variables declaradas tipo "byte y medio". ¿Qué instrucciones MIPS se deberían usar?

1. Carga con `lbu`, `lhu`; aritmética con `add`, `sub`, `mult`, `div`; y almacenamiento usando `sb`, `sh`.
2. Carga con `lb`, `lh`; aritmética con `add`, `sub`, `mult`, `div`; y almacenamiento usando `sb`, `sh`.
3. Carga con `lb`, `lh`; aritmética con `add`, `sub`, `mult`, `div`, usando `and` para enmascarar el resultado a 8 o 16 bits después de cada operación; y almacenamiento usando `sb`, `sh`.

Extensión: En el texto precedente dijimos que se copia el EPC en un registro con `mfc0` y entonces retornamos al código interrumpido con un salto por registro. Esto conduce a una pregunta interesante: puesto que primero se debe transferir el EPC a un registro para usarlo con un salto por registro, ¿cómo puede el salto por registro volver al código interrumpido y restaurar los valores originales de *todos* los registros? O se restauran los registros antiguos en primer lugar, destruyendo consecuentemente el valor de retorno del EPC que se colocó en un registro para usar en el salto por registro, o se restauran todos los registros excepto el que guarda la dirección de retorno de manera que se pueda realizar el salto –¡lo que significa que una excepción cambiaría un registro en cualquier momento durante la ejecución de un programa! –. Ninguna de las dos opciones es aceptable.

Para rescatar al hardware de este problema, los programadores MIPS se han puesto de acuerdo en reservar los registros `$k0` y `$k1` para el sistema operativo; estos registros no se restauran en las excepciones. Así como los compiladores MIPS evitan usar el registro `$at` de manera que el ensamblador pueda usarlo como registro temporal (véase la sección *Interfaz hardware software* en la sección 2.10), los compiladores también se abstienen de usar los registros `$k0` y `$k1` para que estén disponibles para el sistema operativo. Las rutinas de excepciones colocan la dirección de retorno en uno de estos registros y entonces usan un salto por registro para restaurar la dirección de la instrucción.

Extensión: La velocidad de la suma se puede aumentar determinado el acarreo de entrada a los bits de mayor peso cuánto antes. Hay varios esquemas de anticipación de acarreo de forma que, en el peor caso, el retraso es proporcional al log2 del número de

bits del sumador. La propagación de estas señales anticipadas es más rápida porque atraviesan menos puertas, pero la anticipación del acarreo real necesita muchas más puertas. El esquema más popular es *carry lookahead*, que se describe en la sección C.6 del apéndice C en el CD.

La multiplicación es un fastidio, la división es un mal; la regla de tres me deja perplejo, y los ejercicios me enloquecen.

Anónimo, manuscrito de la época de la reina Elisabeth, 1570

3.3 Multiplicación

Ahora que hemos completado la explicación de la suma y la resta, estamos preparados para construir la más fastidiosa operación de multiplicación.

Pero primero revisemos la multiplicación de números decimales manual para recordar los pasos y los nombres de los operandos. Por razones que aparecerán claras pronto, limitamos este ejemplo decimal al uso de los dígitos 0 y 1. Multiplicar 1000_{diez} por 1001_{diez}:

```
Multiplicando                    1000diez
Multiplicador          x         1001diez
                                 ------
                                  1000
                                 0000
                                0000
                               1000
                               -------
Producto                       1001000diez
```

El primer operando se llama *multiplicando* y el segundo *multiplicador*. El resultado final se llama *producto*. Tal y como podrá recordar, el algoritmo que se aprende en la enseñanza primaria es tomar los dígitos del multiplicador de uno en uno de derecha a izquierda, multiplicar el multiplicando por cada dígito del multiplicador y desplazar el producto parcial un dígito a la izquierda de los productos parciales anteriores.

La primera observación es que el número de dígitos del producto es considerablemente mayor que el número de dígitos tanto en el multiplicando como el multiplicador. De hecho, si ignoramos los bits de signo, la longitud de una multiplicación de un multiplicando de n bits y un multiplicador de m bits es de $n + m$ bits. Esto es, $n + m$ bits son necesarios para representar todos los posibles productos. Por ello, como la suma, la multiplicación debe tener en cuenta el desbordamiento, porque frecuentemente queremos un producto de 32 bits como resultado de multiplicar dos números de 32 bits.

En este ejemplo restringimos los dígitos decimales a 0 y 1. Con sólo dos alternativas, cada paso de la multiplicación es simple:

1. Colocar una copia del multiplicando (1 × multiplicando) en el lugar adecuado si el dígito del multiplicador es 1, o
2. Colocar un 0 (0 × multiplicando) en el lugar correcto si el dígito es 0.

Aunque el ejemplo decimal anterior hacía uso sólo de 0 y 1, la multiplicación de números binarios debe usar 0 y 1 siempre, y por eso ofrece sólo estas dos opciones siempre.

Una vez que hemos visto las bases de la multiplicación, tradicionalmente el próximo paso ha sido proporcionar el hardware de multiplicación altamente optimizado. Rompemos con la tradición, convencidos de que conseguirá un mejor entendimiento, viendo la evolución del hardware de multiplicación y el algoritmo a lo largo de múltiples generaciones. Por ahora, asumamos que estamos multiplicando sólo números positivos.

Versión secuencial del algoritmo y el hardware de multiplicación

Este diseño imita el algoritmo que todos nosotros aprendimos en primaria; el hardware se muestra en la figura 3.4. Hemos dibujado el hardware de manera que los datos fluyan de arriba hacia abajo para que se parezca más al método de lápiz y papel.

Supongamos que el multiplicador está en el registro de 32 bits "Multiplicador" y que el registro de 64 bits "Producto" tiene un valor inicial igual a 0. Del ejemplo de lápiz y papel anterior, está claro que necesitaremos mover el multiplicando a la izquierda un dígito cada paso para que pueda ser añadido a los productos intermedios. A lo largo de 32 pasos un multiplicando de 32 bits se movería 32 bits a la izquierda. Por esto, necesitamos un registro multiplicando de 64 bits, cuyo valor inicial es el multiplicando de 32 bits a la derecha y 0s en la mitad izquierda. Este registro se desplaza un bit a la izquierda en cada paso para alinear el multiplicando con la suma que está siendo acumulada en el registro "Producto" de 64 bits.

La figura 3.5 muestra los tres pasos básicos necesarios para cada bit. El bit menos significativo del multiplicador (Multiplicador0) determina si el multiplicando se suma al registro producto. El desplazamiento a la izquierda del paso 2 tiene el efecto de mover los operandos intermedios a la izquierda, justo como cuando se multi-

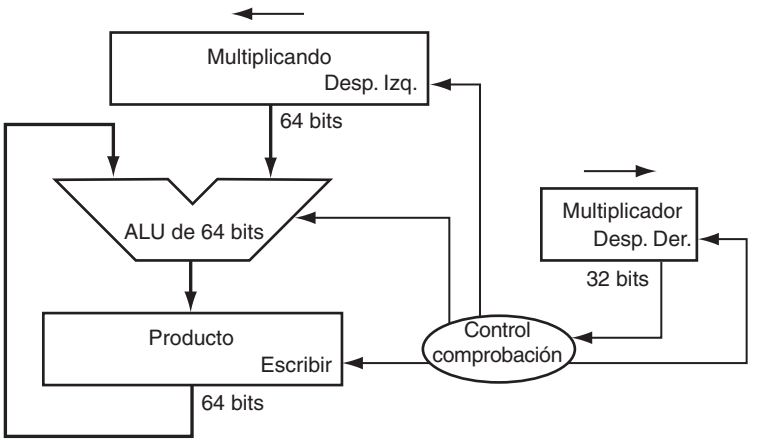


FIGURA 3.4 Primera versión del hardware de multiplicación. El registro multiplicando, la ALU y el registro producto tienen un ancho de 64 bits, y sólo el multiplicador contiene 32 bits. (El apéndice C describe la ALU.) El multiplicando de 32 bits empieza en la mitad derecha del registro multiplicando y se desplaza a la izquierda 1 bit en cada paso. El multiplicador se desplaza en la dirección opuesta a cada paso. El algoritmo empieza con el producto iniciado a cero. El control decide cuándo desplazar los registros multiplicando y multiplicador y cuándo escribir nuevos valores en el registro producto.

plica a mano. El desplazamiento a la derecha en el paso 3 nos da el siguiente bit del multiplicador que se debe examinar en la siguiente iteración. Estos tres pasos se repiten 32 veces para obtener el producto. Si cada paso tarda un ciclo de reloj, este algoritmo requeriría al menos 100 ciclos de reloj para multiplicar 2 números de 32 bits. La importancia relativa de las operaciones aritméticas como la multiplicación varía según el programa, pero la suma y la resta pueden ser en cualquier parte entre

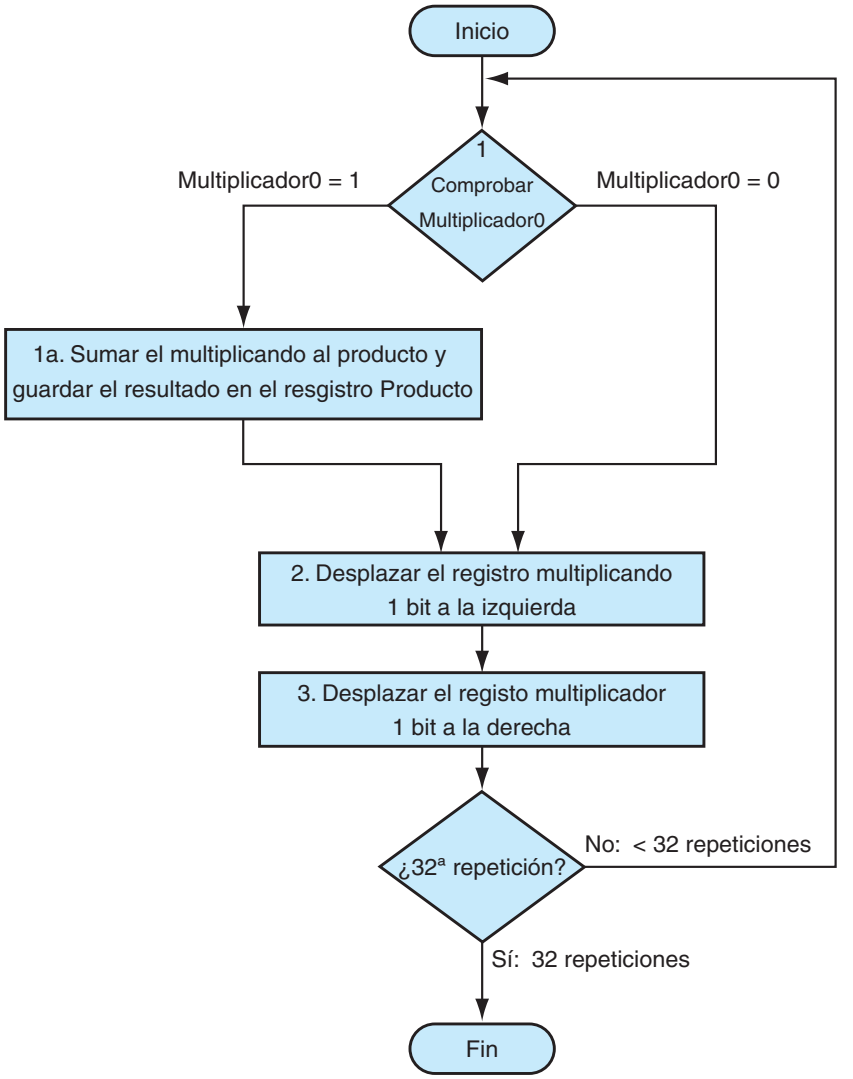


FIGURA 3.5 Primer algoritmo de multiplicación, que utiliza el hardware mostrado en la figura 3.4. Si el bit menos significativo del multiplicador es 1, se suma el multiplicando al producto. Si no, se va al siguiente paso. En los dos próximos pasos, se desplaza el multiplicando a la izquierda y el multiplicador a la derecha. Estos tres pasos se repiten 32 veces.

5 y 100 veces más frecuentes que la multiplicación. Consecuentemente, en muchas aplicaciones, las multiplicaciones pueden tardar varios ciclos de reloj sin afectar significativamente a las prestaciones. Sin embargo, la ley de Amdahl (véase la sección 1.8) nos recuerda que incluso una frecuencia moderada para una operación lenta puede limitar las prestaciones.

Este algoritmo y su hardware se pueden refinar fácilmente para que tarde un ciclo de reloj en cada paso. El aumento de velocidad se obtiene al realizar las operaciones en paralelo: el multiplicador y multiplicando se desplazan mientras el multiplicando se suma al producto si el bit del multiplicador es 1. El hardware sólo tiene que asegurarse de que comprueba el bit correcto del multiplicador y de que obtiene la versión del multiplicando antes del desplazamiento. El hardware se suele optimizar aún más partiendo por la mitad el tamaño del sumador y los registros al fijarse dónde hay trozos no usados de los registros y del sumador. La figura 3.6 muestra el hardware modificado.

Interfaz hardware software

En el caso de multiplicaciones por constantes, es posible reemplazar el cálculo aritmético por desplazamientos. Algunos compiladores reemplazan multiplicaciones por constantes cortas por una serie de desplazamientos y sumas. Puesto que en base 2 un bit a la izquierda representa un número dos veces mayor, desplazar los bits a la izquierda tiene el mismo efecto que multiplicar por una potencia de 2. Tal y como se mencionó en el capítulo 2, casi todos los compiladores implementan la optimización de reducción de esfuerzo que supone sustituir una multiplicación por una potencia de 2 por un desplazamiento a la izquierda.

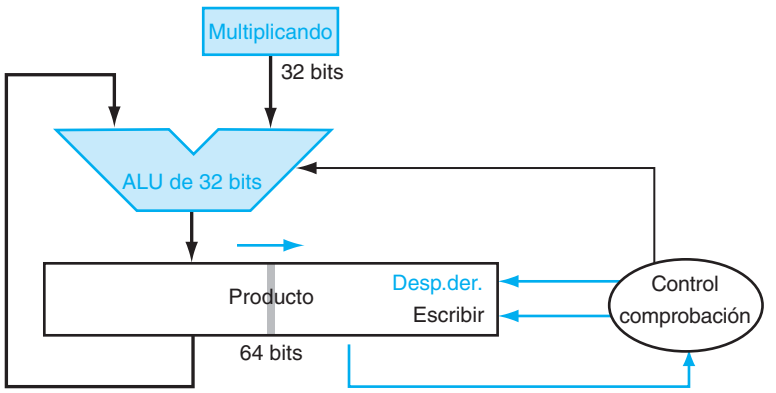


FIGURA 3.6 Versión refinada del hardware de multiplicación. Compárela con la primera versión de la figura 3.4. El registro multiplicando, la ALU y el registro multiplicador tienen un ancho de 32 bits, y sólo el registro producto sigue siendo de 64 bits. Ahora el producto se desplaza a la derecha. El registro multiplicador separado también ha desaparecido. En su lugar, el multiplicador se almacena en la mitad derecha del registro del producto. Estos cambios se destacan en color. El registro Producto debería tener, en realidad, 65 bits para almacenar el acarreo de salida del sumador, pero se muestra con sólo 64 bits para destacar la evolución de la figura 3.4.

EJEMPLO

Algoritmo de multiplicación

Usando números de 4 bits para ahorrar espacio, multiplicar $2_{diez} \times 3_{diez}$, o $0010_{dos} \times 0011_{dos}$.

RESPUESTA

La figura 3.7 muestra el valor de cada registro para cada paso, etiquetados de acuerdo a la figura 3.5, con el valor final de $0000\ 0110_{dos}$ o 6_{diez}. Se usa el color para indicar los valores del registro que cambian en cada paso, y el bit rodeado de un círculo es el bit examinado para determinar la operación de cada paso.

Multiplicación con signo

Hasta el momento multiplicado números positivos. La forma más fácil de entender cómo multiplicar números con signo es convertir primero el multiplicador y multiplicando a números positivos y después recordar los signos originales. Los algoritmos deberían entonces ejecutar 31 iteraciones, dejando los signos fuera del cálculo. Tal y como aprendimos en primaria, debemos negar el producto sólo si los signos originales no son iguales.

Es evidente que el último algoritmo funcionará correctamente con números con signo siempre que recordemos que los números con los que estamos tratando tienen infinitos dígitos, y que sólo los estamos representando con 32 bits. Por tanto, los pasos de desplazamiento necesitarían extender el signo del producto para números con signo. Cuando el algoritmo acaba, la palabra inferior tendría el producto de 32 bits.

Iteración	Paso	Multiplicador	Multiplicando	Producto
0	Valores iniciales	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcando	0011	0000 0010	0000 0010
	2: Desplazar Mcando a la izq.	0011	0000 0100	0000 0010
	3: Desplazar Mdor a la der.	000(1)	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Desplazar Mcando a la izq.	0001	0000 1000	0000 0110
	3: Desplazar Mdor a la der.	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ ninguna operación	0000	0000 1000	0000 0110
	2: Desplazar Mcando a la izq.	0000	0001 0000	0000 0110
	3: Desplazar Mdor a la der.	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Desplazar Mcando a la izq.	0000	0010 0000	0000 0110
	3: Desplazar Mdor a la der.	0000	0010 0000	0000 0110

FIGURA 3.7 Ejemplo de multiplicación usando el algoritmo de la figura 3.6. El bit examinado para determinar el próximo paso está rodeado de un círculo de color.

Multiplicación más rápida

La ley de Moore ha suministrado tantos recursos que los diseñadores de hardware pueden ahora construir un hardware de multiplicación mucho más rápido. Que el multiplicando se vaya a sumar o no se sabe al principio de la multiplicación examinando cada uno de los 32 bits del multiplicador. Es posible realizar multiplicaciones más rápidas proporcionando un sumador de 32 bits para cada bit del multiplicador: una entrada es la operación lógica AND del multiplicando con un bit del multiplicador y la otra es la salida del sumador anterior.

Una forma directa para conseguir esto sería conectando las salidas de los sumadores a las entradas del sumador situadas a su izquierda, formando una pila de 32 sumadores. Otra alternativa para organizar estos 32 sumadores es un árbol paralelo, tal como se muestra en la figura 3.8. En lugar de un retraso igual al tiempo necesario para hacer 32 sumas, el retraso es $\log_2(32)$ o el tiempo necesario para realizar cinco sumas de 32 bits. La figura 3.8 muestra cómo se realiza esta conexión más rápida.

De hecho, la multiplicación puede ser todavía más rápida que cinco sumas si se utilizan *sumadores de acarreo almacenado* (*carry save adders*) (véase la sección C.6 en el **apéndice C** en el CD) y además, como se puede segmentar fácilmente, este diseño es capaz de soportar varias multiplicaciones al mismo tiempo (véase el capítulo 4).

Multiplicación en MIPS

MIPS proporciona un par de registros de 32 bits separados para almacenar el producto de 64 bits, llamados *Hi* y *Lo*. Para realizar un producto con o sin signo correcto, MIPS tiene dos instrucciones: multiplicar (`mult`) y multiplicar sin signo (`multu`). Para buscar el producto de 32 bits entero, el programador puede usar *moves from lo* (`mflo`) (mover desde lo). El ensamblador MIPS genera una pseudoinstrucción para multiplicar que especifica tres registros de propósito general, generando las instrucciones `mflo` y `mfhi` para colocar el producto en registros.

Resumen

La multiplicación se realiza con un hardware sencillo de suma y desplazamiento, derivado del método de lápiz y papel aprendido en la escuela primaria. Los compiladores incluso usan instrucciones de desplazamiento para multiplicaciones por potencias de dos.

Interfaz hardware software

Las dos instrucciones de multiplicar de MIPS ignoran el desbordamiento, por lo que es responsabilidad del software comprobar si el producto es demasiado grande para caber en un registro de 32 bits. No hay desbordamiento si *hi* es 0 para `multu` o es igual al signo replicado 32 veces para `mult`. Se puede usar la instrucción *move from hi* (`mfhi`) (mover desde hi) para transferir *hi* a un registro de propósito general para comprobar el desbordamiento.

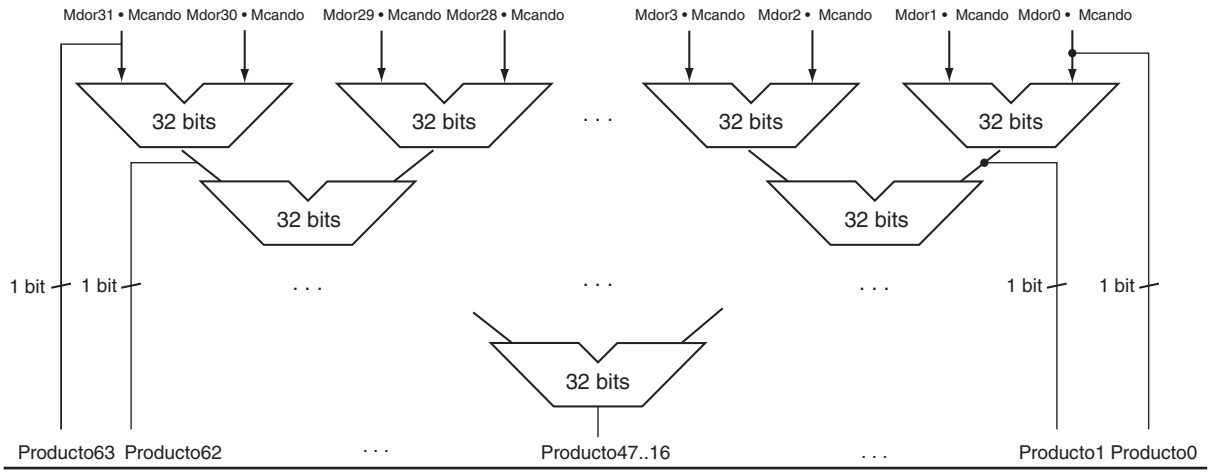


FIGURA 3.8 Hardware para multiplicación rápida. En lugar de usar 31 veces un único sumador de 32 bits, este hardware "desenrolla el lazo" para usar 31 sumadores y los organiza para minimizar el retraso.

Divide et impera.
"Divide y vencerás" en latín, antigua máxima política citada por Maquiavelo, 1532.

3.4 División

La operación recíproca de la multiplicación es la división, una operación que es menos frecuente y más peculiar. Incluso ofrece la oportunidad de realizar una operación no válida matemáticamente: dividir por 0.

Comenzamos con un ejemplo de división larga usando números decimales para recordar los nombres de los operandos y el algoritmo de la división de la escuela primaria. Por razones similares a las de la sección anterior, limitamos los dígitos decimales a 0 y 1. El ejemplo es dividir $1\,001\,010_{diez}$ entre 1000_{diez}:

$$
\begin{array}{rll}
 & 1001_{diez} & \text{Cociente} \\
\text{Divisor } 1000_{diez} & \overline{)\,1001010_{diez}} & \text{Dividendo} \\
 & -1000 & \\
 & \overline{10} & \\
 & 101 & \\
 & 1010 & \\
 & \overline{-1000} & \\
 & 10_{diez} & \text{Resto}
\end{array}
$$

Los dos operandos (**dividendo** y **divisor**) y el resultado (**cociente**) de la división van acompañados por un segundo resultado denominado **resto**. Esta es otra forma de expresar la relación entre los componentes:

$$\text{Dividendo} = \text{cociente} \times \text{divisor} + \text{resto}$$

donde el resto es menor que el divisor. A veces los programas utilizan la instrucción de dividir sólo para obtener el resto, ignorando el cociente.

El algoritmo de la división de la escuela primaria intenta ver cuántas veces se puede restar un número, creando un dígito del cociente en cada intento. Nuestro ejemplo decimal, cuidadosamente seleccionado, sólo usa los números 0 y 1, de manera que es fácil determinar cuántas veces cabe el divisor en la parte del dividendo: es o 0 veces o 1 vez. Los números binarios contienen sólo 0 o 1, así que la división binaria está restringida a estas dos posibilidades, razón por la cual se simplifica la división binaria.

Supongamos que ambos (dividendo y divisor) son positivos y, por lo tanto, el cociente y el resto son no negativos. Los operandos de la división y ambos resultados son valores de 32 bits, y por ahora ignoraremos el signo.

Dividendo: número que ha de dividirse por otro.
Divisor: número por el cual se divide el dividendo.
Cociente: resultado primario de la división; número que cuando se multiplica por el divisor y se le suma el resto da como resultado el dividendo.
Resto: resultado secundario de la división; número que cuando se suma al producto del cociente y el divisor da como resultado el dividendo.

Algoritmo y hardware de división

La figura 3.9 muestra el hardware que imita nuestro algoritmo de la escuela primaria. Empezamos con un registro Cociente de 32 bits con valor inicial 0. Cada iteración del algoritmo necesita mover el divisor a la derecha un dígito, de manera que empezamos con el divisor situado en la mitad izquierda del registro Divisor de 64 bits y, en cada paso, lo desplazamos un bit a la derecha para alinearlo con el dividendo. El valor inicial del registro Resto es el dividendo.

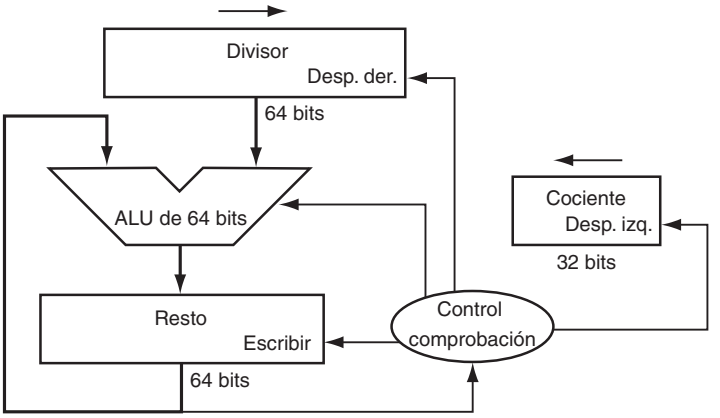


FIGURA 3.9 Primera versión del hardware de división. El registro Divisor, la ALU y registro Resto tienen un ancho de 64 bits, y sólo el registro Cociente es de 32 bits. El divisor de 32 bits empieza en la mitad izquierda del registro Divisor y se desplaza 1 bit a la derecha en cada iteración. El valor inicial del resto es el el dividendo. El control decide cuándo desplazar los registros Divisor y Cociente y cuándo escribir el nuevo valor en el registro Resto.

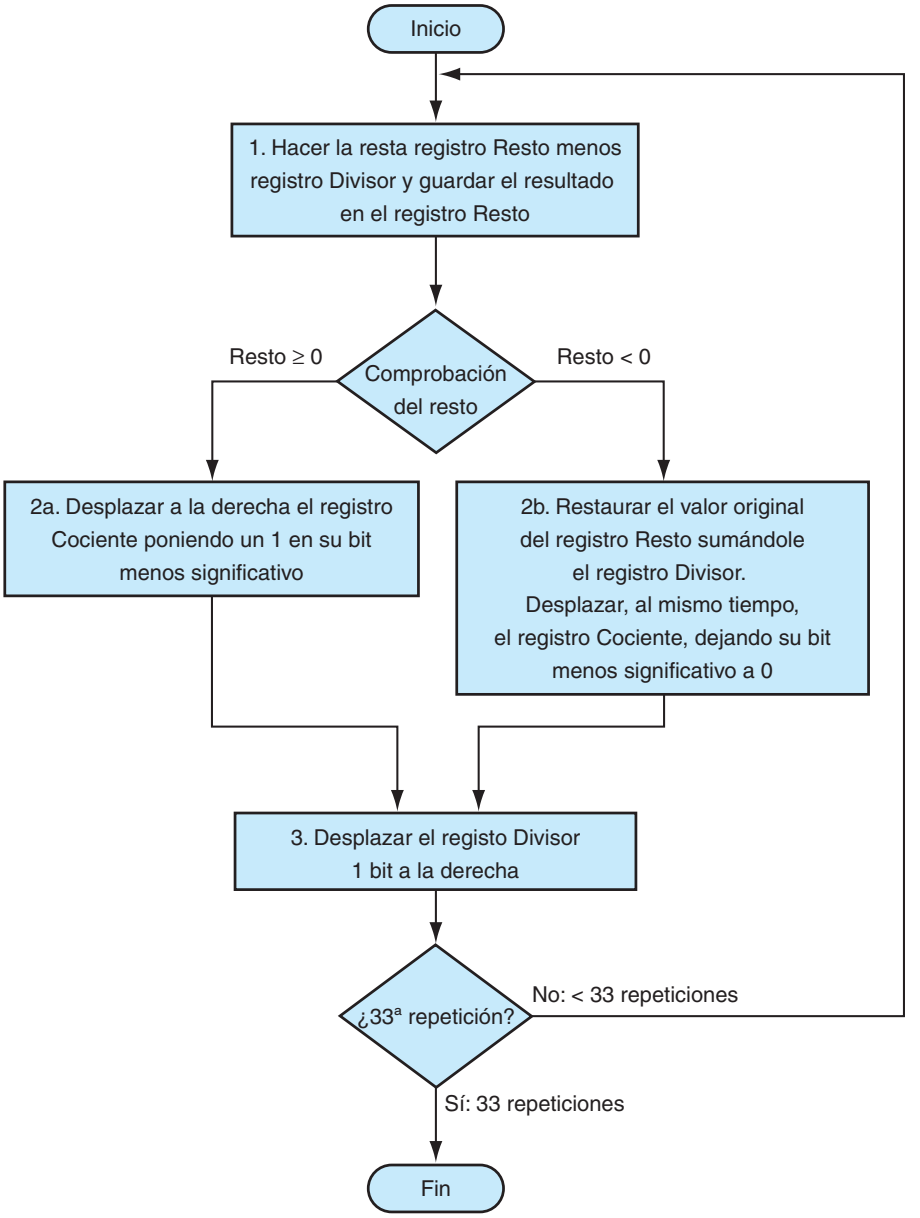


FIGURA 3.10 Un algoritmo de división que utiliza el hardware de la figura 3.9. Si el resto es positivo, el divisor cabía en el dividendo, de manera que el paso 2a genera un 1 en el cociente. Un resto negativo después del paso 1 significa que el divisor no cabía en el dividendo, de manera que el paso 2b genera un 0 en el cociente y suma el divisor al resto, invirtiendo la resta del paso 1. El desplazamiento final, en el paso 3, alinea el divisor adecuadamente, con relación al dividendo, para la siguiente iteración. Estos pasos se repiten 33 veces.

La figura 3.10 muestra los tres pasos del primer algoritmo de división. A diferencia de los humanos, el computador no es suficientemente inteligente para conocer por anticipado si el divisor es menor que el dividendo. Primero debe restar el divisor en el paso 1; recuerde que así es cómo hacíamos la comparación en la instrucción *set on less than* (activar si menor que). Si el resultado es positivo, el divisor es menor o igual que el dividendo, así que generamos un 1 en el cociente (paso 2a). Si el resultado es negativo, el próximo paso es restaurar el valor original añadiendo el divisor al resto y generar un 0 en el cociente (paso 2b). El divisor se desplaza a la derecha y volvemos a iterar de nuevo. El resto y el cociente se encontrarán en sus registros homónimos tras completar todas las iteraciones.

Un algoritmo de división

EJEMPLO

Usando una versión de 4 bits del algoritmo para ahorrar páginas, intentamos dividir 7_{diez} entre 2_{diez}, o $0000\ 0111_{dos}$ entre 0010_{dos}.

RESPUESTA

La figura 3.11 muestra el valor de cada registro para cada uno de los pasos, con el cociente igual a 3_{diez} y el resto a 1_{diez}. Observe que la comprobación en el paso 2 de si el resto es positivo o negativo comprueba simplemente que el bit de signo del registro resto es un 0 o un 1. El requisito sorprendente de este algoritmo es que necesita $n + 1$ pasos para obtener el cociente y el resto correctos.

Este algoritmo y su hardware se pueden refinar para ser más rápidos y baratos. Se obtiene una mayor rapidez desplazando los operandos y el cociente al mismo tiempo que se realiza la resta. Este refinamiento divide en dos el tamaño del sumador y los registros observando dónde están las porciones no usadas de los registros y el sumador. La figura 3.12 muestra el hardware modificado.

División con signo

Hasta ahora hemos ignorado los números con signo en la división. La solución más sencilla es recordar los signos del divisor y el dividendo y entonces negar el cociente si los signos no son iguales.

Extensión: La única complicación de la división con signo es que debemos determinar también el signo del resto. Recordemos que la siguiente ecuación debe cumplirse siempre:

Dividendo = cociente x divisor + resto

Para comprender cómo se determina el signo del resto, fijémonos en el ejemplo de dividir todas las combinaciones de $\pm 7_{diez}$ entre $\pm 2_{diez}$. El primer caso es fácil:

+7 ÷ +2: cociente = +3, resto = +1

Iteración	Paso	Cociente	Divisor	Resto
0	Valores iniciales	0000	0010 0000	0000 0111
1	1: Resto = Resto – Div	0000	0010 0000	1110 0111
	2b: Resto < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Desp. Div a la derecha	0000	0001 0000	0000 0111
2	1: Resto = Resto – Div	0000	0001 0000	1111 0111
	2b: Resto < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Desp. Div a la derecha	0000	0000 1000	0000 0111
3	1: Resto = Resto – Div	0000	0000 1000	1111 1111
	2b: Resto < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Desp. Div a la derecha	0000	0000 0100	0000 0111
4	1: Resto = Resto – Div	0000	0000 0100	0000 0011
	2a: Resto ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Desp. Div a la derecha	0001	0000 0010	0000 0011
5	1: Resto = Resto – Div	0001	0000 0010	0000 0001
	2a: Resto ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Resto Div a la derecha	0011	0000 0001	0000 0001

FIGURA 3.11 Ejemplo de división usando el algoritmo de la figura 3.10. El bit examinado para determinar el próximo paso está rodeado por un círculo en color.

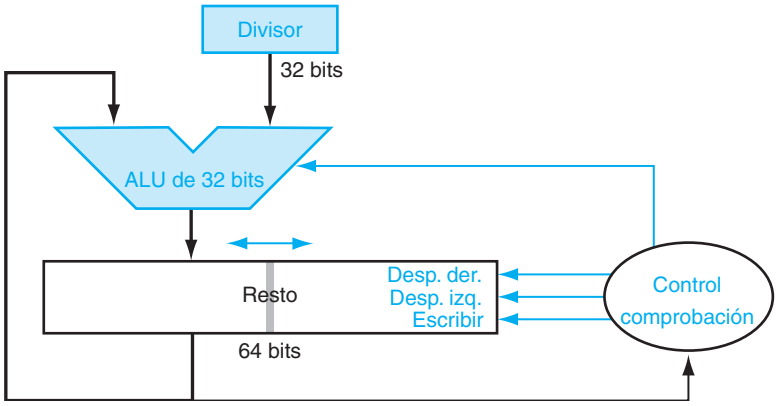


FIGURA 3.12 Una versión mejorada del hardware de división. El registro Divisor, la ALU y el registro Cociente tienen todos un ancho de 32 bits, y sólo el registro Resto sigue siendo de 64 bits. Comparado con la figura 3.10, el número de bits de la ALU y el registro Divisor se ha dividido a la mitad y el resto se desplaza a la izquierda. Esta versión también combina el registro Cociente con la mitad derecha del registro Resto. (Como en la figura 3.6, el registro Resto debería tener 65 bits, para asegurarnos de que el acarreo de la suma no se pierde.)

Comprobando los resultados:

$7 = 3 \times 2 + (+1) = 6 + 1$

Si cambiamos el signo del dividendo, el cociente debe cambiar también:

–7 ÷ +2: cociente = –3

Reescribiendo nuestra fórmula básica para calcular el resto:

Resto = (dividendo – cociente × divisor) = –7 – (–3 × +2) = –7 – (–6) = –1

Así pues,

–7 ÷ +2: cociente = –3, resto = –1

Comprobando los resultados de nuevo:

–7 = –3 × 2 + (–1) = –6 – 1

La razón por la que la respuesta no es un cociente de –4 y un resto de +1, que también cumplirían la fórmula, es que el valor absoluto del cociente cambiaría dependiendo de los signos del dividendo y del divisor. Claramente, si

$-(x \div y) \neq (-x) \div y$

la programación sería aún un reto mayor. Este comportamiento anómalo se evita siguiendo la regla de que el dividendo y el resto deben tener los mismos signos, sin importar cuáles son los signos del divisor y el cociente.

Calculamos las otras combinaciones siguiendo la misma regla:

+7 ÷ –2: cociente = –3, resto = +1
–7 ÷ –2: cociente = +3, resto = –1

De esta forma, el algoritmo de la división con signo correcto niega el cociente si los signos de los operandos son opuestos y hace que el signo del resto no nulo sea igual al del dividendo.

División más rápida

En la multiplicación usamos muchos sumadores para acelerarla, pero no podemos utilizar el mismo truco para la división. La razón es que necesitamos saber el signo de la diferencia antes de que podamos ejecutar el siguiente paso del algoritmo, mientras que con la multiplicación podíamos calcular los 32 productos parciales inmediatamente.

Existen técnicas para producir más de un bit del cociente por bit. La técnica de la *división SRT* intenta adivinar varios bits del cociente por paso, usando una tabla de consulta basada en los bits superiores del dividendo y del resto. Utiliza los pasos siguientes para corregir las suposiciones incorrectas. Un valor típico hoy en día es 4 bits. La clave está en adivinar el valor que hay que restar. En la división binaria solamente hay una única elección. Estos algoritmos usan 6 bits del resto y 4 bits del divisor para indexar una tabla que determina el cálculo para cada paso.

La precisión de este método rápido depende de que se tenga valores apropiados en la tabla de consulta. La falacia de la página 276 de la sección 3.8 muestra qué puede pasar si la tabla no es correcta.

División en MIPS

De las figuras 3.6 y 3.12, ya habrá observado que se puede usar el mismo hardware secuencial tanto para multiplicar como para dividir. El único requisito es un registro de 64 bits que pueda desplazarse a la izquierda o a la derecha y una ALU de 32 bits que sume o reste. Por ello, MIPS usa los registros de 32 bits *hi* y *lo*

tanto para multiplicar como para dividir. Como podemos esperar del algoritmo anterior, al terminar la instrucción de división, *Hi* contiene el resto y *Lo* contiene el cociente.

Para manejar tanto enteros con signo como sin signo, MIPS tiene dos instrucciones: *dividir* (`div`) y *dividir sin signo* (`divu`). El ensamblador MIPS permite especificar tres registros de las instrucciones de dividir, generando las instrucciones `mflo` y `mfhi` para colocar el resultado deseado en un registro de propósito general.

Resumen

El hardware común para soportar la multiplicación y la división permite que MIPS proporcione un par de registros de 32 bits que se usan tanto para multiplicar como para dividir. La figura 3.13 resume los elementos añadidos a la arquitectura MIPS en las dos últimas secciones.

Interfaz hardware software

Las instrucciones de división de MIPS ignoran el desbordamiento, de manera que el software debe determinar si el cociente es demasiado grande. Además del desbordamiento, la división también puede producir un cálculo impropio: la división por 0. Algunos computadores distinguen entre estos dos sucesos anómalos. El software MIPS debe comprobar el divisor para descubrir la división por 0, así como el desbordamiento.

Extensión: Un algoritmo de la división aún más rápido no suma el divisor inmediatamente para restaurar si el resto es negativo. Simplemente suma el dividendo al resto desplazado en el próximo paso puesto que $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. Este algoritmo sin restauración, que tarda un ciclo por paso, se examina con más detalle en los ejercicios; el algoritmo que hemos visto aquí se llama división *con restauración*. Un tercer algoritmo que no almacena el resultado de la resta cuando es negativo es el llamado algoritmo de división *sin representación* (*nonperforming division*), que de media tiene un tercio menos de operaciones aritméticas.

La velocidad no lleva a ninguna parte si estás en la dirección equivocada

Proverbio americano

3.5 Punto flotante

Además de los enteros con y sin signo, los lenguajes de programación soportan números con decimales, que en matemáticas se llaman *reales*. Veamos algunos ejemplos de números reales:

$3.14159265\ldots_{\text{diez}}$ (π)

$2.71828\ldots_{\text{diez}}$ (e)

$0.000000001_{\text{diez}}$ o $1.0_{\text{diez}} \times 10^{-9}$ (segundos en un nanosegundo)

$3\ 155\ 760\ 000_{\text{diez}}$ o $3.15576_{\text{diez}} \times 10^{9}$ (segundos en un siglo normal)

Lenguaje ensamblador del MIPS

Categoria	Instruccción	Ejemplo		Significado	Comentarios
Aritmética	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Tres operandos; desbordamiento detectado
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Tres operandos; desbordamiento detectado
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constante; desbordamiento detectado
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Tres operandos; desbordamiento no detectado
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Tres operandos; desbordamiento no detectado
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constante; desbordamiento no detectado
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Usado para copiar Exception PC más otros registros especiales
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	Producto con signo de 64 bits en Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 8.0 pt $s3	Producto sin signo de 64 bits en Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = cociente, Hi = resto
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Cociente y resto sin signo
	move from Hi	mfhi	$s1	$s1 = Hi	Para obtener copia de Hi
	move from Lo	mflo	$s1	$s1 = Lo	Para obtener copia de Lo
Transferencia de datos	load word	lw	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Palabra de memoria a registro
	store word	sw	$s1,100($s2)	Memory[$s2 +100] = $s1	Palabra de memoria a registro
	load half unsigned	lhu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Media palabra de memoria a registro
	store half	sh	$s1,100($s2)	Memory[$s2 + 100] = $s1	Media palabra de registro a memoria
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte de memoria a registro
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1	Byte de registro a memoria
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Carga una constante en los 16 bits superiores
Lógicas	and	and	$s1,$s2,$s3	$s1 = $s2 & $s3	Operandos en tres registros; AND bit a bit
	or	or	$s1,$s2,$s3	$s1 = $s2 | $s3	Operandos en tres registros; OR bit a bit
	nor	nor	$s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Operandos en tres registros; NOR bit a bit
	and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100	ANDbit a bit con constante t
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	OR bit a bit con constante
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Desplazamiento por una constante a la izquierda
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Desplazamiento por una constante a la derecha
Salto condicional	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Comprobación de igualdad; salto relativo al PC
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Comprobación de desigualdad; salto relativo al PC
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Comparación menor que; complemento a dos
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Comparación < constante; complemento a dos
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Comparación menor que; sin signo
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Comparación < constante; sin signo
Salto incondicional	jump	j	2500	go to 10000	Salto a la dirección destino
	jump register	jr	$ra	go to $ra	Para switch, procedimiento de retorno
	jump and link	jal	2500	$ra = PC + 4; go to 10000	Para llamada a procedimiento

FIGURA 3.13 Arquitectura MIPS presentada hasta ahora. La memoria y los registros de la arquitectura MIPS no se incluyen por razón de espacio, pero esta sección añadió los registros *Hi* y *Lo* para soportar la multiplicación y la división. El lenguaje máquina de MIPS se muestra en la tarjeta de referencia resumen MIPS en la contraportada de este libro.

En el último ejemplo, observe que el número no representa una mantisa pequeña, pero es mayor de lo que podemos representar con un entero con signo de 32 bits. La notación alternativa para los dos últimos números se llama **notación científica**, que tiene un único dígito a la izquierda del decimal. Un número en notación científica que no tenga 0s al principio se llama un número **normalizado**, que es la forma habitual de escribirlo. Por ejemplo, $1.0_{diez} \times 10^{-9}$ está en notación científica normalizada, pero $0.1_{diez} \times 10^{-8}$ y $10.0_{diez} \times 10^{-10}$ no lo están.

Notación científica: notación que expresa los números con un único dígito a la izquierda del punto decimal.
Normalizado: número en notación en punto flotante que no tiene 0s por delante.

De la misma forma que representamos números decimales en notación científica, también podemos representar números binarios en notación científica:

$$1.0_{dos} \times 2^{-1}$$

Para mantener un número binario en forma normalizada, necesitamos una base que podamos incrementar y decrementar por el número exacto de bits que deben ser desplazados para tener un dígito no nulo a la izquierda del punto decimal. Solamente la base 2 cumple este requisito. Puesto que la base no es 10, también necesitamos un nuevo nombre para el punto decimal; una buena opción es *punto binario*.

Punto flotante: aritmética del computador que representa números en los cuales el punto binario no es fijo.

La aritmética del computador que soporta tales números se llama **punto flotante** porque representa números en los cuales el punto binario no es fijo, como ocurre con los enteros. El lenguaje de programación C usa el nombre *float* para estos números. Igual que en la notación científica, los números se representan con un único dígito no cero a la izquierda del punto binario. En binario, la forma es

$$1.xxxxxxxxx_{dos} \times 2^{yyyy}$$

(Aunque el computador también representa el exponente en base 2 al igual que el resto del número, para simplificar la notación mostramos el exponente en decimal).

Una notación científica estándar para reales en forma normalizada ofrece tres ventajas: simplifica el intercambio de datos que incluyen números en punto flotante; simplifica los algoritmos de la aritmética en punto flotante al saber que los números estarán siempre en este formato; e incrementa la precisión de los números que se pueden almacenar en una palabra, puesto que los 0s precedentes innecesarios se reemplazan por dígitos significativos a la derecha del punto binario.

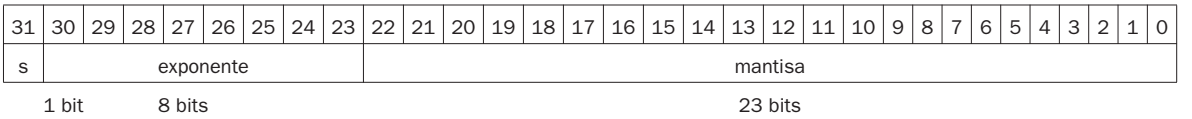
Representación en punto flotante

Un diseñador de una representación en punto flotante debe encontrar un compromiso entre el tamaño de la **mantisa** y el tamaño del **exponente** porque una longitud fija de palabra significa que se debe tomar un bit de uno para añadirlo al otro. Este compromiso es entre precisión y rango: incrementar el tamaño de la mantisa mejora la precisión de la mantisa, mientras que incrementar el tamaño del exponente incrementa el rango de números que se pueden representar. Tal y como nos recuerda nuestra directriz de diseño del capítulo 2, un buen diseño implica un buen compromiso.

Mantisa: valor, generalmente entre 0 y 1, situado en el campo mantisa.
Exponente: en el sistema de representación numérica de la aritmética en punto flotante, el valor que se coloca en el campo de exponente.

Los números en punto flotante frecuentemente son múltiplos del tamaño de una palabra. A continuación se muestra la representación de un número en punto flotante MIPS, donde *s* es el signo del número en punto flotante (un 1 significa negativo), el *exponente* es el valor del campo exponente de 8 bits (incluido el signo

del exponente), y la *mantisa* es el número de 23 bits. Esta representación se llama *signo y magnitud*, puesto que el signo tiene un bit separado del resto del número.



En general, los números en punto flotante son habitualmente de la forma

$$(-1)^S \times F \times 2^E$$

F implica el valor del campo mantisa y E el del campo exponente; la relación exacta entre estos campos se explicará pronto. (Veremos en breve que MIPS hace algo un poco más sofisticado).

Estos tamaños escogidos de exponente y mantisa dan a la aritmética del computador MIPS un rango extraordinario. Fracciones tan pequeñas como $2.0_{diez} \times 10^{-38}$ y números casi tan grandes como $2.0_{diez} \times 10^{38}$ se pueden representar en un computador. Extraordinario sí, pero no infinito, por lo que aún es posible que haya números demasiado grandes para el computador. Por tanto, en la aritmética de punto flotante, al igual que en la de enteros, pueden ocurrir interrupciones de desbordamiento. Obsérvese que aquí **desbordamiento** significa que el exponente es demasiado grande para ser representado en el campo exponente.

Desbordamiento (punto flotante): situación en la cual un exponente positivo es demasiado grande para caber en el campo exponente.

El punto flotante presenta también un nuevo tipo de suceso excepcional. Así como los programadores querrán saber cuándo han calculado un número que es demasiado grande para ser representado, también querrán saber si la mantisa que están calculando se ha hecho tan pequeña que no se puede representar; ambos sucesos podrían hacer que un programa diese respuestas incorrectas. Para distinguirlo del desbordamiento, se le llama **desbordamiento a cero (*underflow*)**. Esta situación ocurre cuando el exponente negativo es demasiado grande para caber en el campo del exponente.

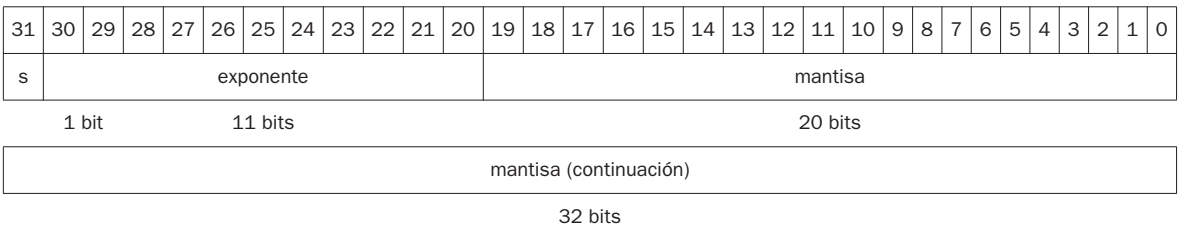
Desbordamiento a cero (*underflow*) (punto flotante): situación en la cual un exponente negativo es demasiado grande para caber en el campo exponente.

Una forma de reducir la posibilidad de desbordamiento a cero o desbordamiento es ofrecer otro formato que tiene un exponente mayor. En C este número se llama *double*, y las operaciones con *doubles* se llaman aritmética en punto flotante de **precisión doble**, mientras que el nombre del formato anterior es punto flotante de **precisión simple**.

Precisión doble: valor en punto flotante representado en dos palabras de 32 bits.

Precisión simple: valor en punto flotante representado en una palabra de 32 bits.

La representación de un número en punto flotante de precisión doble ocupa dos palabras MIPS, como se muestra más abajo, donde *s* sigue siendo el signo del número, *exponente* es el valor del campo del exponente de 11 bits y *mantisa* es el número de 52 bits del campo mantisa.



La precisión doble de MIPS permite números casi tan pequeños como $2.0_{diez} \times 10^{-308}$ y casi tan grandes como $2.0_{diez} \times 10^{308}$. Aunque la precisión doble incrementa el rango del exponente, su principal ventaja es la mayor precisión a causa de la mantisa más grande.

Estos formatos van más allá de MIPS. Son parte del *estándar en punto flotante IEEE 754*, que se utiliza en casi todos los computadores creados desde 1980. Este estándar ha mejorado enormemente tanto la portabilidad de los programas de punto flotante como la calidad de la aritmética de los computadores.

Para empaquetar aún más bits en la parte significativa, el IEEE 754 hace implícito el valor 1 para el primer bit de los números binarios normalizados. Por esto, el número tiene en realidad 24 bits de longitud en precisión simple (un 1 implícito u oculto y una mantisa de 23 bits) y de 53 bits en precisión doble (1 + 52). Para ser precisos, usamos el término *la parte significativa* para representar los números de 24 o 53 bits, que es 1 más la parte fraccionaria, y *mantisa* cuando nos referimos al número de 23 o 52 bits. Puesto que 0 no tiene un primer 1, al valor 0 se le reserva un exponente con valor 0 de manera que el hardware no le añadirá un primer 1.

Así, $00\ldots00_{dos}$ representa el 0; la representación del resto de los números usa la forma anterior con el 1 oculto añadido:

$$(-1)^S \times (1 + \text{mantisa}) \times 2^E$$

donde los bits de la mantisa representan un número entre 0 y 1, y E especifica el valor del campo exponente, que se detallará en breve. Si numeramos los bits de la mantisa *de izquierda a derecha* s1, s2, s 3, . . . , el valor es

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

La figura 3.14 muestra las codificaciones de los números en punto flotante IEEE 754. Otras características del IEEE 754 son los símbolos especiales para representar sucesos inusuales. Por ejemplo, en lugar de interrumpir ante una división por 0, el software puede forzar el resultado a un patrón de bits representando $+\infty$ o $-\infty$; el mayor exponente se reserva para estos símbolos especiales. Cuando el programador imprime los resultados, el programa imprimirá un símbolo de infinito. (Para los expertos en matemáticas, el propósito del infinito es dar completitud topológica a los reales).

Precisión simple		Precisión doble		Objeto representado
Exponente	Mantisa	Exponente	Mantisa	
0	0	0	0	0
0	Distinto de cero	0	Distinto de cero	± número no normalizado
1–254	cualquiera	1–2046	cualquiera	± número en punto flotante
255	0	2047	0	± infinito
255	Distinto de cero	2047	Distinto de cero	NaN (*Not a Number*)

FIGURA 3.14 Codificación IEEE 754 para números en punto flotante. Un bit de signo independiente determina el signo. Los números desnormalizados se describen en la extensión de la página 270. Esta información se encuentra también en la columna 4 de la Tarjeta de Datos de Referencia de MIPS, que e incluye con el libro.

El IEEE 754 tiene incluso un símbolo para el resultado de operaciones inválidas, tales como 0/0 o restar infinito de infinito. Este símbolo es *NaN*, por "No un Número" (*Not a Number*, en inglés). El propósito de *NaN* es permitir a los programadores posponer varias comprobaciones y decisiones en el programa para cuando sea conveniente.

Los diseñadores del IEEE 754 también querían una representación que se pudiese procesar fácilmente mediante comparaciones de enteros, especialmente para ordenar. Este deseo es el que hace que el signo esté en el bit más significativo, permitiendo una comprobación rápida de *menor que*, *mayor que*, o *igual a 0*. (Es un poco más complicado que una simple ordenación de enteros, puesto que esta notación es esencialmente de signo y magnitud en lugar de complemento a dos).

Colocar el exponente antes que la mantisa también simplifica la ordenación de números en punto flotante mediante instrucciones de comparación de enteros, puesto que los números con un exponente mayor aparentan ser mayores que los números con exponentes menores, siempre que ambos exponentes tengan el mismo signo.

Los exponentes negativos plantean una dificultad a la ordenación simplificada. Si usamos complemento a dos o cualquier otra notación en la cual los exponentes negativos tengan un 1 en el bit más significativo del campo del exponente, un exponente negativo parecerá un número mayor. Por ejemplo, $1.0_{dos} \times 2^{-1}$ se representaría como

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Recuerde que el primer 1 está implícito en la mantisa). El valor $1.0_{dos} \times 2^{+1}$ parecería un número menor.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

La notación deseable debe por tanto representar el exponente más negativo como $00 \ldots 00_{dos}$, y el más positivo como $11 \ldots 111_{dos}$. Esta convención se llama notación sesgada (*biased notation*), siendo el sesgo (o desplazamiento) el número que se resta a la representación normal, sin signo, para determinar el valor real.

IEEE 754 usa un sesgo de 127 para la precisión simple, de manera que –1 se representa por el patrón de bits del valor $-1 + 127_{diez}$, o $126_{diez} = 0111\ 1110_{dos}$, y +1 se representa por 1 + 127, o $128_{diez} = 1000\ 0000_{dos}$. El sesgo del exponente para precisión doble es 1023. El exponente sesgado significa que el valor representado por un número en punto flotante es realmente

$$(-1)^S \times (1 + \text{mantisa}) \times 2^{(\text{exponente-sesgo})}$$

El rango de los números en precisión simple va desde un número tan pequeño como el

$\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{dos} \times 2^{-126}$

hasta uno tan grande como el

$\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_{dos} \times 2^{+127}$

Veamos la representación.

Representación en punto flotante

EJEMPLO

Mostrar la representación binaria IEEE 754 del número -0.75_{diez} en precisión simple y doble

RESPUESTA

El número -0.75_{diez} también es

$$-3/4_{diez} \text{ o } -3/2^2{}_{diez}$$

También se representa por la mantisa binaria

$$-11_{dos}/2^2{}_{diez} \text{ o } -0.11_{dos}$$

En notación científica, este valor es

$$-0.11_{dos} \times 2^0$$

y en notación científica normalizada es

$$-1.1_{dos} \times 2^{-1}$$

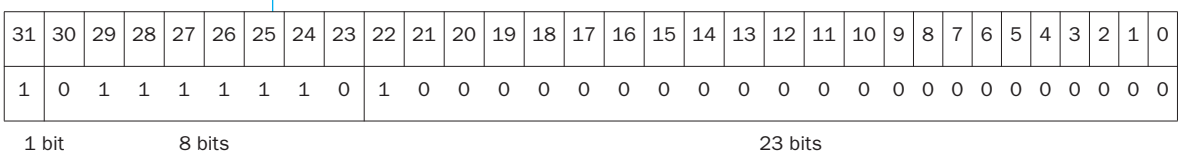
La representación general para un número en simple precisión es

$$(-1)^S \times (1 + \text{mantisa}) \times 2^{(\text{exponente} - 127)}$$

Cuando restamos el sesgo 127 del exponente de $-1.1_{dos} \times 2^{-1}$, el resultado es

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{dos}) \times 2^{(126 - 127)}$$

La representación binaria en precisión simple de -0.75_{diez} entonces es



La representación en precisión doble es

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\text{dos}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Ahora vamos a mostrar cómo ir en la otra dirección.

Conversión de binario a decimal en punto flotante

EJEMPLO

¿Qué número decimal está representado por este número en punto flotante de precisión simple?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

RESPUESTA

El bit de signo es 1, el campo de exponente contiene 129, y el campo mantisa contiene $1 \times 2^{-2} = 1/4$, o 0.25. Usando la ecuación básica,

$$\begin{aligned}(-1)^S \times (1 + \text{mantisa}) \times 2^{(\text{exponente} - \text{sesgo})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

En las próximas secciones daremos los algoritmos para la suma y la multiplicación en punto flotante. Esencialmente usan las operaciones enteras correspondientes sobre las mantisas, pero es necesario un mantenimiento extra para manipular los exponentes y normalizar el resultado. En primer lugar damos una deducción intuitiva de los algoritmos en decimal, y a continuación una versión binaria más detallada en las figuras.

Extensión: En un intento de incrementar el rango sin eliminar bits de la mantisa, algunos computadores anteriores al estándar IEEE 754 usaban una base diferente de la binaria. Por ejemplo, los grandes computadores IBM 360 y 370 usaban la base 16. Puesto que cambiar el exponente de IBM en 1 significa desplazar la mantisa 4 bits, los números en la base "normalizada" 16 pueden tener hasta 3 bits de comienzo a 0. De ahí que los dígitos hexadecimales significan que pueden eliminarse hasta 3 bits, lo que conduce a sorprendentes problemas en la precisión en la aritmética en punto flotante, como se muestra en la sección 3.6. Los últimos grandes computadores IBM admiten el estándar IEEE 755, así como el formato hexadecimal.

Suma en punto flotante

Sumemos a mano números en notación científica para ilustrar los problema que se encuentran en la suma en punto flotante: $9.999_{diez} \times 10^{1} + 1.610_{diez} \times 10^{-1}$. Suponga que sólo podemos almacenar cuatro dígitos decimales de la mantisa y dos dígitos decimales del exponente.

Paso 1. Para ser capaces de sumar estos números correctamente, debemos alinear el punto decimal del número que tiene el menor exponente. Por ello necesitamos una forma del número menor, $1.610_{diez} \times 10^{-1}$, que coincida con el mayor exponente. Para obtenerla nos fijamos que hay varias representaciones de un número en punto flotante no normalizado en notación científica:

$$1.610_{diez} \times 10^{-1} = 0.1610_{diez} \times 10^{0} = 0.01610_{diez} \times 10^{1}$$

El número de la derecha es la versión que deseamos, puesto que su exponente coincide con el exponente del número mayor, $9.999_{diez} \times 10^{1}$. Así, el primer paso desplaza la mantisa del número menor a la derecha hasta que el exponente corregido coincida con el del número mayor. Pero podemos representar sólo cuatro dígitos decimales, así que, después de desplazar, el número es en realidad:

$$0.016_{diez} \times 10^{-1}$$

Paso 2. A continuación viene la suma de las mantisas:

$$\begin{array}{rr} & 9.999_{diez} \\ + & 0.016_{diez} \\ \hline & 10.015_{diez} \end{array}$$

La suma es $10.015_{diez} \times 10^{1}$.

Paso 3. Esta suma no está normalizada en notación científica, de manera que necesitamos ajustarla:

$$10.015_{\text{diez}} \times 10^1 = 1.0015_{\text{diez}} \times 10^2$$

De este modo, después de la suma puede que tengamos que desplazar la suma para ponerla en forma normalizada, ajustando el exponente de manera apropiada. Este ejemplo muestra el desplazamiento a la derecha, pero si un número fuese positivo y el otro negativo, podría ser posible que la suma tuviese muchos ceros al comienzo, lo que requeriría desplazamientos a la izquierda. Siempre que el exponente se incrementa o decrementa, debemos comprobar el desbordamiento o el desbordamiento a cero —es decir, debemos asegurarnos de que el exponente aún cabe en su campo—.

Paso 4. Puesto que supusimos que la mantisa sólo podía tener 4 dígitos (excluyendo el signo), debemos redondear el número. En nuestro algoritmo de la escuela primaria, las reglas de redondeo truncan el número por el dígito que se redondea si el valor del dígito que está su derecha es de 0 a 4, y suman 1 al dígito que se redondea si el que está a su derecha vale de 5 a 9. El número

$$1.0015_{\text{diez}} \times 10^2$$

se redondea a 4 dígitos en la mantisa a

$$1.002_{\text{diez}} \times 10^2$$

puesto que el cuarto dígito a la derecha del punto decimal estaba entre 5 y 9. Observe que si hubiésemos tenido mala suerte al redondear, tal como sumar 1 a una cadena de 9s, la suma ya no estaría normalizada y tendríamos que realizar de nuevo el paso 3.

La figura 3.15 muestra el algoritmo para la suma binaria en punto flotante que sigue este ejemplo en decimal. Los pasos 1 y 2 son similares al ejemplo mostrado: se ajusta la mantisa del número con el exponente menor y después se suman las dos mantisas. El paso 3 normaliza el resultado, forzando una comprobación del desbordamiento y del desbordamiento a cero. La comprobación del desbordamiento y del desbordamiento a cero depende de la precisión de los operandos. Recuerde que el patrón de bits todos a cero en el exponente está reservado y se usa para la representación del cero en punto flotante. También, el patrón de bits todos a uno en el exponente está reservado para indicar valores y situaciones que están fuera del ámbito normal del los números en punto flotante (véase la extensión de la página 270). Así, para precisión simple, el máximo exponente es 127, y el mínimo exponente es –126. Los límites para precisión doble son 1023 y –1022.

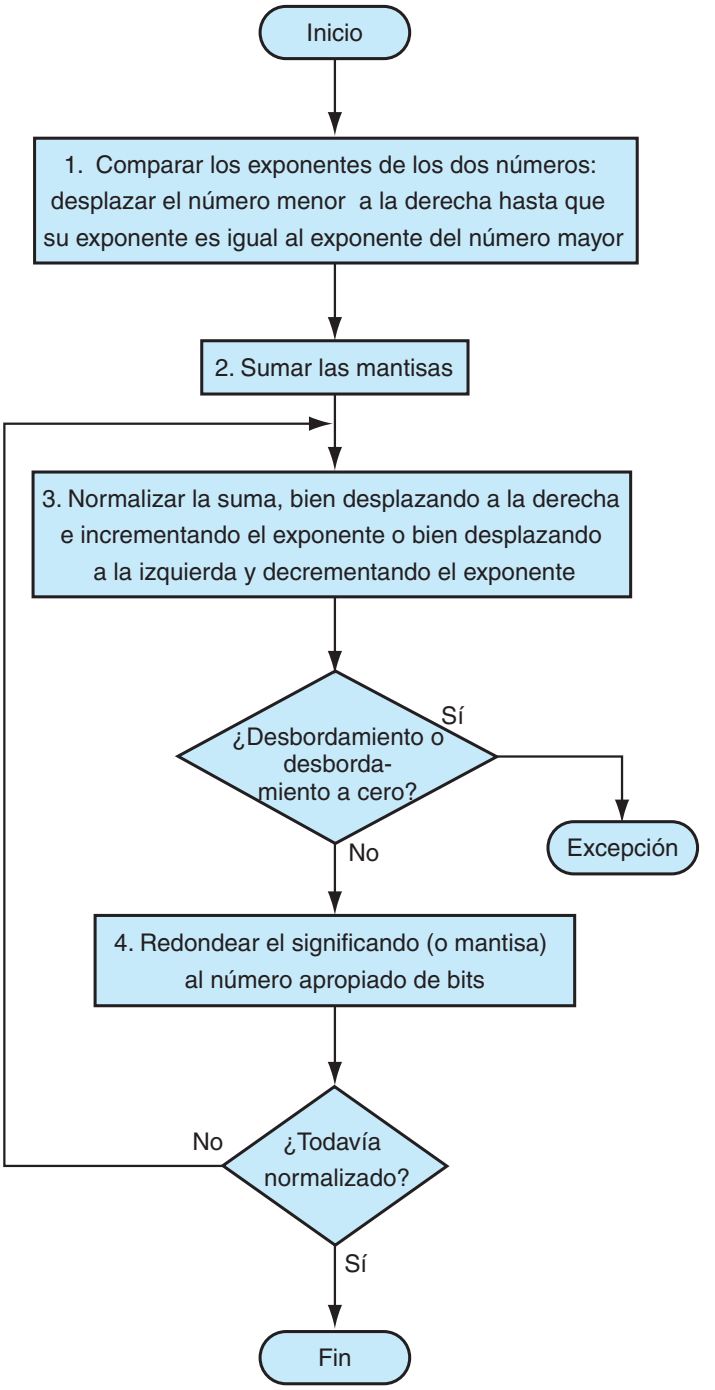


FIGURA 3.15 Suma en punto flotante. Lo normal es ejecutar los pasos 3 y 4 una sola vez, pero si el redondeo provoca la desnormalización de la suma, hay que repetir el paso 3.

Suma binaria en punto flotante

EJEMPLO

Intentar sumar los números 0.5_{diez} y -0.4375_{diez} en binario usando el algoritmo de la figura 3.15.

RESPUESTA

Primero examinemos la versión binaria de los dos números en notación científica normalizada, suponiendo que se tienen 4 bits de precisión:

$$\begin{aligned} 0.5_{diez} &= 1/2_{diez} &&= 1/2^1{}_{diez} \\ &= 0.1_{dos} &&= 0.1_{dos} \times 2^0 &&= 1.000_{dos} \times 2^{-1} \\ -0.4375_{diez} &= -7/16_{diez} &&= -7/2^4{}_{diez} \\ &= -0.0111_{dos} &&= -0.0111_{dos} \times 2^0 &&= -1.110_{dos} \times 2^{-2} \end{aligned}$$

Ahora, sigamos el algoritmo:

Paso 1. El significando (o mantisa) del número con el exponente menor ($-1.11_{dos} \times 2^{-2}$) se desplaza a la derecha hasta que su exponente coincide con el del número mayor:

$$-1.110_{dos} \times 2^{-2} = -0.111_{dos} \times 2^{-1}$$

Paso 2. Sumamos los significandos:

$$1.000_{dos} \times 2^{-1} + (-0.111_{dos} \times 2^{-1}) = 0.001_{dos} \times 2^{-1}$$

Paso 3. A continuación, normalizamos la suma, comprobando el desbordamiento o el desbordamiento a zero:

$$\begin{aligned} 0.001_{dos} \times 2^{-1} &= 0.010_{dos} \times 2^{-2} &&= 0.100_{dos} \times 2^{-3} \\ &= 1.000_{dos} \times 2^{-4} \end{aligned}$$

Puesto que $127 \geq -4 \geq -126$, no hay desbordamiento ni desbordamiento a cero. (El exponente sesgado sería $-4 + 127$, o 123, que está entre 1 y 254, que son los exponentes sesgados no reservados menor y mayor, respectivamente).

Paso 4. Redondeamos la suma:

$$1.000_{dos} \times 2^{-4}$$

La suma ya cabe exactamente en 4 bits, de manera que no hay cambios en los bits debido al redondeo.

La suma es entonces:

$$\begin{aligned} 1.000_{dos} \times 2^{-4} &= 0.0001000_{dos} &&= 0.0001_{dos} \\ &= 1/2^4{}_{diez} &&= 1/16_{diez} &&= 0.0625_{diez} \end{aligned}$$

Esta suma es el resultado esperado de sumar 0.5_{diez} a -0.4375_{diez}

Muchos computadores dedican hardware para ejecutar las operaciones en punto flotante lo más rápido posible. La figura 3.16 esboza la organización básica del hardware para la suma en punto flotante.

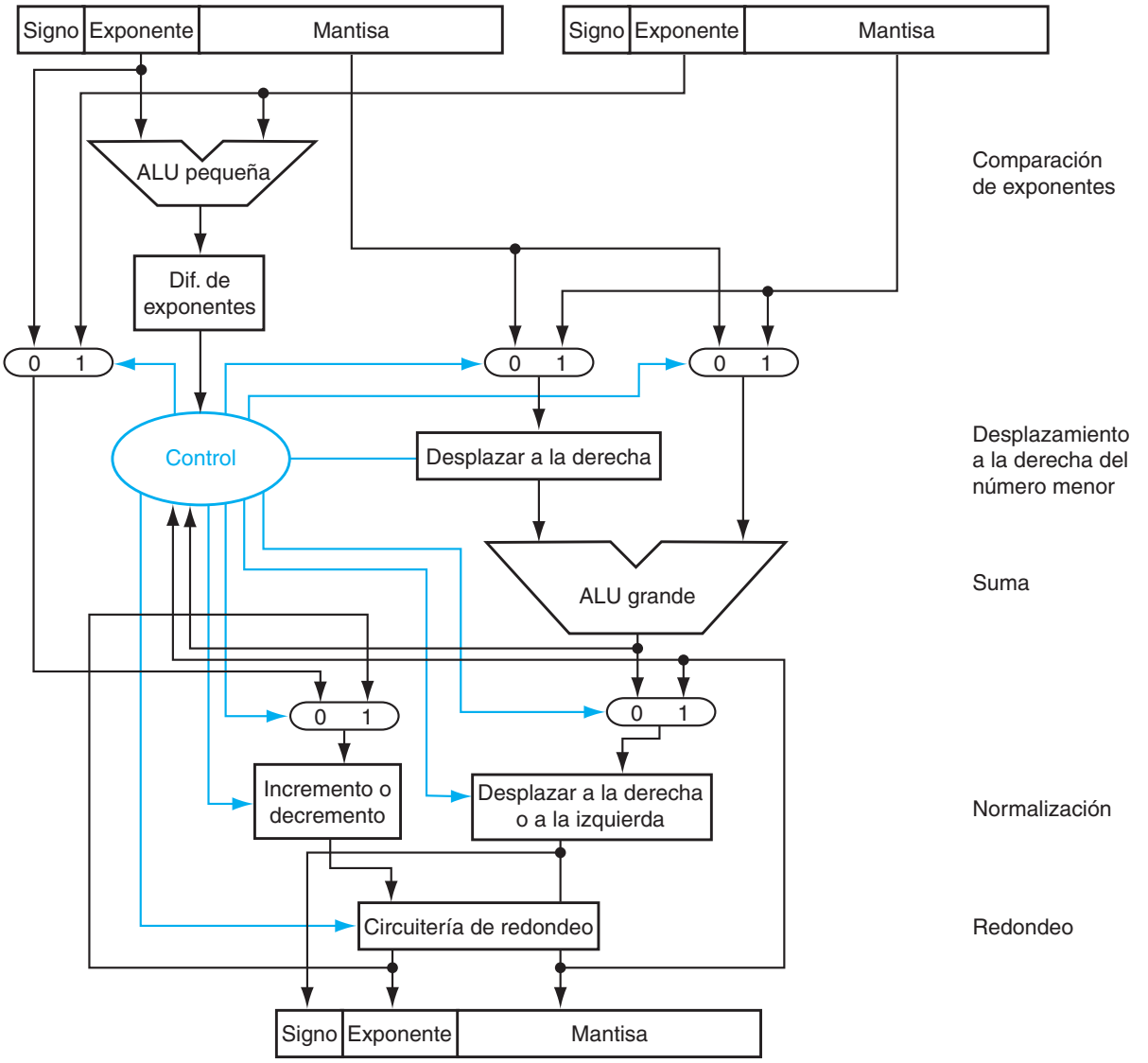


FIGURA 3.16 Diagrama de bloques de una unidad aritmética dedicada a la suma en punto flotante. Los pasos de la figura 3.15 corresponden a cada bloque, de arriba abajo. Primero, el exponente de un operando se resta del otro usando una pequeña ALU para determinar cuál es mayor y por cuánto. Esta diferencia controla los tres multiplexores; de izquierda a derecha, seleccionan el mayor exponente, el significando del número menor y el significando del número mayor. El significando del menor se desplaza a la derecha, y se suman los significandos en la ALU grande. El paso de normalización desplaza la suma a la izquierda o derecha e incrementa o decrementa el exponente. El redondeo genera el resultado final, el cual puede requerir una nueva normalización para obtener el resultado final.

Multiplicación en punto flotante

Ahora que hemos explicado la suma en punto flotante, veamos la multiplicación. Empezamos multiplicando números decimales en notación científica a mano: $1.110_{diez} \times 10^{10} \times 9.200_{diez} \times 10^{-5}$. Suponga que sólo podemos almacenar cuatro cifras en la mantisa y dos en el exponente.

Paso 1. A diferencia de la suma, calculamos el exponente del producto simplemente sumando los exponentes de ambos operandos:

$$\text{Nuevo exponente} = 10 + (-5) = 5$$

Veamos cómo funcionaría esto con los exponentes sesgados, para asegurarnos de que se obtiene el mismo resultado: 10 + 127 = 137, y –5 + 127 = 122, de manera que

$$\text{Nuevo exponente} = 137 + 122 = 259$$

Este resultado es demasiado grande para el campo exponente de 8 bits, por lo que algo falla. El problema está en el sesgo porque estamos sumando los sesgos junto con los exponentes:

$$\text{Nuevo exponente} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

De acuerdo con esto, para obtener la suma sesgada correcta cuando sumamos numeros sesgados, debemos restar el sesgo a la suma:

$$\text{Nuevo exponente} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

y 5 es, además, el resultado que calculamos inicialmente.

Paso 2. Después viene la multiplicación de los significandos:

$$\begin{array}{r} 1.110_{diez} \\ \times\ 9.200_{diez} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{diez} \end{array}$$

Hay tres cifras a la derecha del punto decimal para cada operando, de manera que el punto decimal está colocado seis cifras desde la derecha en el significando del producto:

$$10.212000_{diez}$$

Suponiendo que sólo se pueden almacenar tres cifras a la derecha del punto decimal, el producto es 10.212×10^5.

Paso 3. Este producto no está normalizado, de manera que tenemos que normalizarlo:

$$10.212_{diez} \times 10^5 = 1.0212_{diez} \times 10^6$$

Así, después de la multiplicación, el producto puede verse desplazado una cifra a la derecha para ponerlo en forma normalizada, sumando 1 al exponente. En este punto, podemos comprobar si hay desbordamiento o desbordamiento a cero. El desbordamiento a cero se puede producir si los dos operandos son pequeños –es decir, si ambos tienen exponentes negativos grandes–.

Paso 4. Como supusimos que el significando tenía sólo 4 cifras (excluyendo el signo), debemos redondear el número. El número

$$1.0212_{diez} \times 10^6$$

se redondea a cuatro cifras en el significando, dando

$$1.021_{diez} \times 10^6$$

Paso 5. El signo del producto depende de los signos de los operandos originales. Si ambos son el mismo, el signo es positivo; en caso contrario, es negativo. De aquí que el producto sea

$$+1.021_{diez} \times 10^6$$

El signo del resultado de la suma en el algoritmo de sumar estaba determinado por la suma de los significandos, pero en la multiplicación el signo del producto se determina a partir de los signos de los operandos.

De nuevo, como muestra la figura 3.17, la multiplicación de números binarios en punto flotante es muy similar a los pasos que acabamos de describir. Empezamos calculando el nuevo exponente del producto sumando los exponentes sesgados, asegurándonos de restar un sesgo para obtener el resultado correcto. Después se multiplican los significandos, seguido por un paso de normalización opcional. Se comprueba que el tamaño del exponente no presente desbordamiento o desbordamiento a cero y, después, se redondea el producto. Si el redondeo conduce a una normalización adicional, tenemos que comprobar de nuevo el tamaño del exponente. Finalmente, se actualiza el bit de signo a 1 si el signo de los operandos era diferente (producto negativo) o a 0 si tenían el mismo (producto positivo).

EJEMPLO

Multiplicación decimal en punto flotante

Multiplicar los números 0.5_{diez} y -0.4375_{diez}, usando los pasos de la figura 3.17.

En binario, la tarea es multiplicar $1.000_{dos} \times 2^{-1}$ por $-1.110_{dos} \times 2^{-2}$.

RESPUESTA

Paso 1. Sumar los exponentes sin sesgo:

$$-1 + (-2) = -3$$

o, usando las representaciones sesgadas:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2)+(127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Paso 2. Multiplicar los significandos:

$$\begin{array}{r} 1.000_{dos} \\ \times\ 1.110_{dos} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{dos} \end{array}$$

El producto es $1.110000_{dos} \times 2^{-3}$, pero necesitamos ajustarlo a cuatro bits, así que será $1.110_{dos} \times 2^{-3}$.

Paso 3. Ahora comprobamos el producto para asegurarnos de que está normalizado, y después comprobamos si hay desbordamiento o desbordamiento a cero. El producto ya está normalizado y, puesto que $127 \geq -3 \geq -126$, no hay ningún desbordamiento. (Usando la representación sesgada, $254 \geq 124 \geq 1$, de manera que el exponente quepa).

Paso 4. El redondeo del producto no produce ningún cambio:

$$1.110_{dos} \times 2^{-3}$$

Paso 5. Puesto que los signos de los operandos originales difieren, el signo del producto es negativo. Por lo tanto, el producto es

$$-1.110_{dos} \times 2^{-3}$$

Si lo convertimos a decimal para comprobar el resultado:

$$\begin{aligned} -1.110_{dos} \times 2^{-3} &= -0.001110_{dos} = -0.00111_{dos} \\ &= -7/2^{5}{}_{diez} = -7/32_{diez} = -0.21875_{diez} \end{aligned}$$

El producto de 0.5_{diez} y -0.4375_{diez} es efectivamente -0.21875_{diez}.

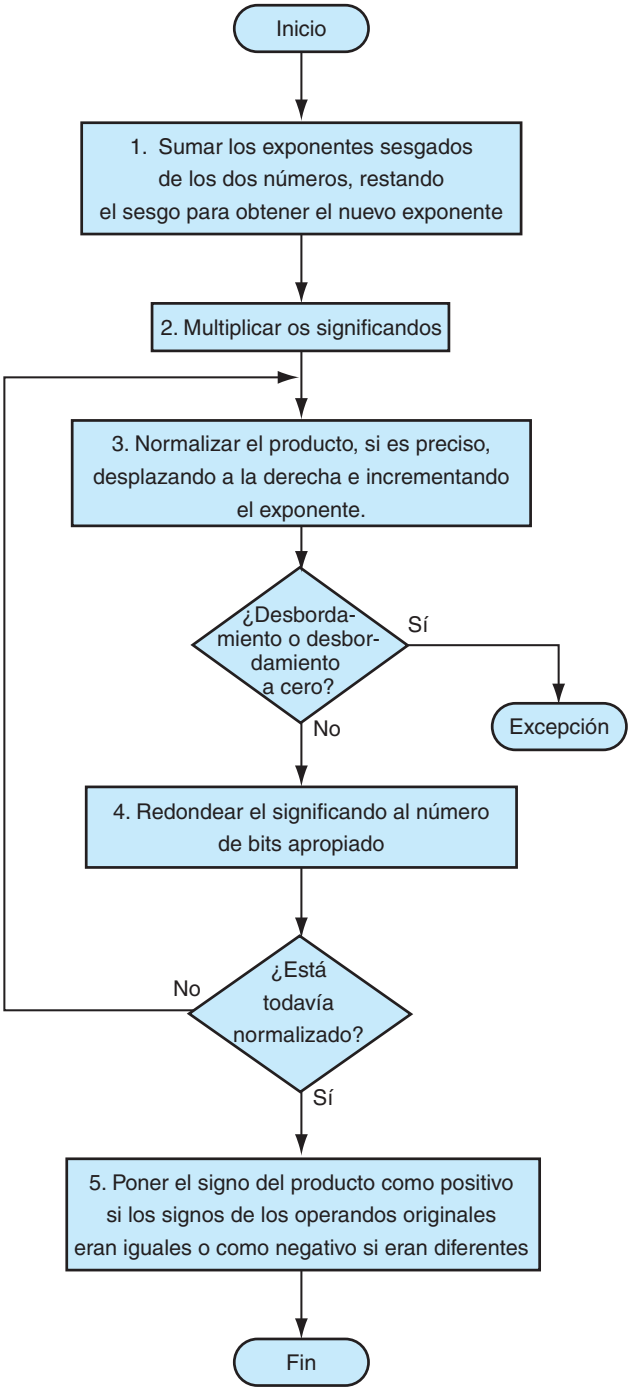


FIGURA 3.17 Multiplicación en punto flotante. Lo normal es ejecutar los pasos 3 y 4 una sola vez, pero si el redondeo provoca que el resultado no esté normalizado, se debe repetir el paso 3.

Instrucciones de punto flotante en MIPS

MIPS soporta los formatos IEEE 754 de precisión simple y doble con las siguientes instrucciones:

- Suma en punto flotante, precisión simple (`add.s`) y doble (`add.d`)
- Resta en punto flotante, precisión simple (`sub.s`) y doble (`sub.d`)
- Multiplicación en punto flotante, precisión simple (`mul.s`) y doble (`mul.d`)
- División en punto flotante, precisión simple (`div.s`) y doble (`div.d`)
- Comparación en punto flotante, precisión simple (`c.×.s`) y doble (`c.×.d`), donde × puede ser igual (`eq`), diferente (`neq`), menor (`lt`), menor o igual (`le`), mayor (`gt`), mayor o igual (`ge`)
- Salto condicional en punto flotante, verdadero (`bclt`) o falso (`bclf`)

La comparación en punto flotante actualiza un bit a cierto o verdadero, dependiendo de la condición de comparación, y el salto condicional en punto flotante decide si salta o no, dependiendo de la condición.

Los diseñadores de MIPS decidieron añadir registros en punto flotante separados, llamados `$f0`, `$f1`, `$f2`, . . . —usados tanto para precisión simple como para precisión doble–. De ahí que incluyeran instrucciones de carga y almacenamiento separadas para los registros de punto flotante: `lwc1` y `swc1`. Los registros base para las transferencias de datos en punto flotante siguen siendo los registros de enteros. El código MIPS para cargar dos números en precisión simple desde memoria, y almacenar el resultado de la suma podría ser de la siguiente manera:

```
lwc1  $f4,x($sp)  # número de 32-bit punto flotante en F4
lwc1  $f6,y($sp)  # Carga un número de 32-bit punto
                  # flotante en F6
add.s $f2,$f4,$f6 # F2 = F4 + F6 precisión simple
swc1  $f2,z($sp)  # Almacena número de 32-bit punto
                  # flotante desde F2
```

Un registro en precisión doble es en realidad una pareja de registros de precisión simple par-impar, que utiliza el registro par como nombre.

La figura 3.18 resume la parte de la arquitectura MIPS relativa al punto flotante vista en este capítulo, con los elementos añadidos para soportar punto flotante resaltados en color. De manera similar a la figura 2.19 en el capítulo 2, la figura 3.19 muestra la codificación de estas instrucciones.

Operandos en punto flotante del MIPS

Nombre	Ejemplo	Comentarios
32 registros de punto flotante	`$f0, $f1, $f2, . . . , $f31`	Registros de punto flotante. Se usan por pares para números de precisión doble.
2^{30} palabras de memoria	Memory[0], Memory[4], . . . , Memory[4294967292]	Accesible sólo por instrucciones de transferencia de datos. El MIPS utiliza direcciones de byte, de modo que la dirección de dos palabras consecutivas difieren en 4. La memoria mantiene estructuras de datos, como tablas y registros almacenados, como en el caso de los que se salvan en llamadas a procedimientos.

Lenguaje ensamblador de punto flotante del MIPS

Categoría	Instrucción	Ejemplo	Significado	Comentarios
Aritmética	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	suma en punto flotante (precisión simple)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	resta en punto flotante (precisión simple)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	multiplicación en punto flotante (precisión simple)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	división en punto flotante (precisión simple)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	suma en punto flotante (precisión doble)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	resta en punto flotante (precisión doble)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	multiplicación en punto flotante (precisión doble)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	división en punto flotante (precisión doble)
Transferencia de datos	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	dato de 32 bits a registro de punto flotante
	store word copr. 1	`swc1 $f1,100($s2)`	`Memory[$s2 + 100] = $f1`	dato de 32 bits a memoria
Salto condicional	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	salto relativo al PC si condición
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	salto relativo al PC si no condición
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	comparación de menor, en punto flotante, precisión simple
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	comparación de menor, en punto flotante, precisión doble

Lenguaje máquina de punto flotante del MIPS

Nombre	Formato	Ejemplo						Comentarios
`add.s`	R	17	16	6	4	2	0	`add.s $f2,$f4,$f6`
`sub.s`	R	17	16	6	4	2	1	`sub.s $f2,$f4,$f6`
`mul.s`	R	17	16	6	4	2	2	`mul.s $f2,$f4,$f6`
`div.s`	R	17	16	6	4	2	3	`div.s $f2,$f4,$f6`
`add.d`	R	17	17	6	4	2	0	`add.d $f2,$f4,$f6`
`sub.d`	R	17	17	6	4	2	1	`sub.d $f2,$f4,$f6`
`mul.d`	R	17	17	6	4	2	2	`mul.d $f2,$f4,$f6`
`div.d`	R	17	17	6	4	2	3	`div.d $f2,$f4,$f6`
`lwc1`	I	49	20	2	100			`lwc1 $f2,100($s4)`
`swc1`	I	57	20	2	100			`swc1 $f2,100($s4)`
`bc1t`	I	17	8	1	25			`bc1t 25`
`bc1f`	I	17	8	0	25			`bc1f 25`
`c.lt.s`	R	17	16	4	2	0	60	`c.lt.s $f2,$f4`
`c.lt.d`	R	17	17	4	2	0	60	`c.lt.d $f2,$f4`
Tamaño del campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas las instrucciones del MIPS tienen 32 bits

FIGURA 3.18 Arquitectura en punto flotante de MIPS presentada hasta ahora. Véase el apéndice B, sección B.10, para más detalles. Esta información se encuentra también en la columna 2 de la Tarjeta de Datos de Referencia de MIPS, que se incluye con el libro.

op(31:26):								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FlPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

FIGURA 3.19 Codificación de las instrucciones en punto flotante de MIPS. Esta notación da el valor de cada campo por fila y columna. Por ejemplo, en la parte superior de la figura `lw` se encuentra en la fila número 4 (100_{dos} en los bits 31-29 de la instrucción) y en la columna número 3 (011_{dos} en los bits 28-26 de la instrucción), de manera que el valor correspondiente al campo *op* (bits 31-26) es 100011_{dos}. El subrayado indica que el campo se usa en otra parte. Por ejemplo, FlPt en la fila 2 columna 1 (010001_{dos}) se define en la parte inferior de la figura. Por tanto, `sub.f` en la fila 0 columna 1 de la parte inferior de la sección significa que el campo `funct` (bits 5-0) de la instrucción es 000001_{dos} y el campo *op* (bits 31-26) es 010001_{dos}. Obsérvese que el campo *rs*, de 5 bits, especificado en la parte central de la figura, determina si la operación es de precisión simple($f = s$, o sea $rs = 10\,000$) o precisión doble ($f = d$, o sea $rs = 10\,001$). Análogamente, el bit 16 de la instrucción determina si la instrucción `bc1.c` comprueba el valor cierto (bit 16 = 1 => `bc1.t`) o falso (bit 16 = 0 => `bc1.f`). Las instrucciones en color se describen en el capítulo 2, mientras que el apéndice B cubre todas las instrucciones. Esta información se encuentra también en la columna 2 de la Tarjeta de Datos de Referencia de MIPS, que se incluye con el libro.

Interfaz hardware software

Una cuestión que deben afrontar los diseñadores al incorporar aritmética en punto flotante es la de si utilizar los mismos registros usados por las instrucciones para enteros o añadir un conjunto separado de registros para punto flotante. Debido a que los programas realizan normalmente operaciones enteras y de punto flotante sobre datos distintos, separar los registros sólo incrementará ligeramente las instrucciones necesarias para ejecutar el programa. El impacto mayor está en crear un conjunto separado de instrucciones de transferencia de datos para mover datos entre los registros de punto flotante y la memoria.

El beneficio de tener registros de punto flotante separados es que se tiene el doble de registros sin tener que usar más bits en el formato de instrucción, se tiene más ancho de banda de registros al tener conjuntos separados de registros de enteros y de punto flotante y se tiene la capacidad de configurar registros para punto flotante; por ejemplo, algunos computadores convierten todos los operandos de registros en un único formato interno.

EJEMPLO

Compilación de un programa de punto flotante en C a código ensamblador MIPS

Convirtamos una temperatura de Fahrenheit a Celsius:

```
float f2c (float fahr)
        {
              return ((5.0/9.0) * (fahr - 32.0));
        }
```

Supongamos que el argumento en punto flotante `fahr` se pasa en `$12` y el resultado debe quedar en `$f0`. (A diferencia de los registros de enteros, el registro en punto flotante 0 puede contener un número). ¿Cuál es el código ensamblador MIPS?

RESPUESTA

Supongamos que el compilador coloca las tres constantes en punto flotante en memoria dentro del rango de fácil acceso con el puntero global `$gp`. Las dos primeras instrucciones cargan las constantes 5.0 y 9.0 en registros de punto flotante:

```
f2c:
      lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 en memoria)
      lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 en memoria)
```

A continuación, se dividen para obtener la mantisa 5.0/9.0:

```
      div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Muchos compiladores dividirían 5.0 entre 9.0 en tiempo de compilación y guardarían la constante 5.0/9.0 en memoria, evitando así tener que dividir en tiempo de ejecución). A continuación, cargamos la constante 32.0 y la restamos de `fahr` (`$f12`):

```
lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18  # $f18 = fahr - 32.0
```

Finalmente, multiplicamos los dos resultados intermedios, dejando el producto en `$f0` como valor a retornar, y, entonces, se retorna:

```
mul.s$f0,  $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr  $ra # return
```

Ahora, veamos operaciones en punto flotante entre matrices, código muy habitual en programas científicos.

Compilación en MIPS de un procedimiento en punto flotante en C con dos matrices de dos dimensiones

EJEMPLO

Muchas operaciones en punto flotante se realizan en precisión doble. Realicemos la multiplicación de matrices X = X + Y * Z. Supongamos que X, Y y Z son matrices cuadradas con 32 elementos en cada dimensión.

```
void mm (double x[][], double y[][], double z[][])
{
        int i, j, k;

        for (i = 0; i!= 32; i = i + 1)
        for (j = 0; j!= 32; j = j + 1)
        for (k = 0; k!= 32; k = k + 1)
          x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

Las direcciones de inicio de las matrices son parámetros, de manera que están en `$a0`, `$a1` y `$a2`. Supongamos que las variables enteras están en `$s0`, `$s1` y `$s2`, respectivamente. ¿Cuál es el código ensamblador MIPS para el cuerpo del procedimiento?

RESPUESTA

Observe que en el lazo más interno se usa `x[i][j]`. Puesto que el índice del lazo es k, este índice no afecta a `x[i][j]`, por lo que se puede evitar cargarlo y almacenarlo en cada iteración. En lugar de eso, el compilador carga `x[i][j]` en un registro antes del lazo, acumula la suma de los productos de `y[i][k]` y `z[k][j]` en ese mismo registro, y almacena la suma en `x[i][j]` al finalizar el lazo más interno.

Conseguimos un código más simple usando la pseudoinstrucción del lenguaje ensamblador `li` (que carga una constante en un registro), y `l.d` y `s.d` (que el ensamblador transforma en un par de instrucciones de transferencia de datos, `lwc1` o `swc1`, para un par de registros en punto flotante).

El cuerpo del procedimiento empieza guardando el valor 32 de finalización del lazo en un registro temporal e inicializando las tres variables de los lazos *for*:

```
mm:...
        li    $t1, 32    # $t1 = 32 (tamaño de la fila/
                         # fin del lazo)
        li    $s0, 0     # i = 0; inicia el primer lazo for
L1:     li    $s1, 0     # j = 0; reinicia el segundo
                         # lazo for
L2:     li    $s2, 0     # k = 0; reinicia el tercer
                         # lazo for
```

Para calcular la dirección de `x[i][j]`, necesitamos saber cómo se almacena en memoria una matriz de dos dimensiones 32 × 32. Como es de esperar, su organización es la misma que si hubiera 32 vectores de 32 elementos cada uno. Así que el primer paso es saltar los `i` vectores, o filas, para obtener el que queremos. Así, multiplicamos el índice en la primera dimensión por el tamaño de la fila, 32. Puesto que 32 es una potencia de 2, en lugar del producto podemos usar un desplazamiento:

```
sll  $t2, $s0, 5 # $t2 = i * 2^5 (tamaño de la fila de x)
```

Ahora añadimos el segundo índice para seleccionar el `j`-ésimo elemento de la fila deseada:

```
addu  $t2, $t2, $s1 # $t2 = i * tamaño(fila) + j
```

Para convertir esta suma en un índice de byte, la multiplicamos por el tamaño en bytes de un elemento de la matriz. Como cada elemento es de 8 bytes por ser de precisión doble, en lugar de multiplicar se puede desplazar a la izquierda tres posiciones:

```
sll  $t2, $t2, 3 # $t2 = desplazamiento byte de [i][j]
```

A continuación, añadimos esta suma a la dirección base de `x`, que da la dirección de `x [i][j]`, y después se carga el número en precisión doble `x[i][j]` en `$f4`:

```
addu  $t2, $a0, $t2 # $t2 = dirección byte de x[i][j]
l.d   $f4, 0($t2)   # $f4 = 8 bytes de x[i][j]
```

Las siguientes cinco instrucciones son casi idénticas a las cinco precedentes: calculan la dirección y después cargan el número en precisión doble `z[k][j]`.

```
L3: sll $t0, $s2, 5 # $t0 = k * 2^5 (tamaño de la fila de z)
    addu $t0, $t0, $s1 # $t0 = k * tamaño(fila) + j
    sll $t0, $t0, 3 # $t0 = desplazamiento byte de [k][j]
    addu $t0, $a2, $t0 # $t0 = dirección byte de z[k][j]
    l.d $f16, 0($t0)   # $f16 = 8 bytes de z[k][j]
```

Análogamente, las siguientes cinco instrucciones son como las últimas cinco: calculan la dirección y después cargan el número en precisión doble `y[i][k]`.

```
sll    $t2, $s0, 5    # $t0 = i * 2^5 (tamaño de la fila de y)
addu   $t0, $t0, $s2  # $t0 = i * tamaño (fila) + k
sll    $t0, $t0, 3    # $t0 = tamaño (fila) [i][k]
addu   $t0, $a1, $t0  # $t0 = desplazamiento byte de y[i][k]
l.d    $f18, 0($t0)   # $f18 = 8 bytes de y[i][k]
```

Ahora que ya tenemos todos los datos cargados, ¡estamos preparados para hacer alguna operación en punto flotante! Multiplicamos los elementos de `y` y `z` que se encuentran en los registros `$f18` y `$f16`, y entonces acumulamos la suma en `$f4`.

```
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # f4 = x[i][j]+ y[i][k] * z[k][j]
```

El bloque final incrementa el índice `k` y vuelve a iterar si el índice no es 32. Si es 32, y por tanto se ha acabado el lazo más interno, necesitamos almacenar en `x[i][j]` el resultado de la suma acumulada en `$f4`.

```
addiu  $s2, $s2, 1   # $k k + 1
bne    $s2, $t1, L3  # si (k != 32) ir a L3
s.d    $f4, 0($t2)   # x[i][j] = $f4
```

Análogamente, estas últimas cuatro instrucciones incrementan las variables índice de los lazos intermedio y externo, volviendo a iterar si el índice no es 32 y finalizando si el índice es 32.

```
addiu  $s1, $s1, 1  # $j = j + 1
bne    $s1, $t1, L2 # si (j != 32) ir a L2
addiu  $s0, $s0, 1  # $i = i + 1
bne    $s0, $t1, L1 # si (i != 32) ir a L1
...
```

Extensión: La organización de las matrices presentada en este ejemplo, llamada *por filas*, se usa en C y muchos otros lenguajes de programación. Fortran, en cambio, usa la organización *por columnas*, donde la matriz se guarda columna por columna.

Extensión: Sólo 16 de los 32 registros en punto flotante de MIPS se pueden usar para operaciones en precisión doble: `$f0`, `$f2`, `$f4`,..., `$f30`. La precisión doble se calcula usando pares de estos registros de precisión simple. Los registros impares se usan sólo para cargar y almacenar la mitad derecha de los números en punto flotante de 64 bits. MIPS-32 añadió `l.d` y `s.d` al repertorio de instrucciones. MIPS-32 también añadió versiones "para pares sueltos" de todas las instrucciones en punto flotante, donde una instrucción simple se convertía en dos operaciones en punto flotante en paralelo sobre dos operandos de 32 bits dentro de los registros de 64 bits. Por ejemplo, `add.ps $f0, $f2, $f4` es equivalente a `add.s $f0, $f2, $f4` seguida por `add.s $f1, $f3, $f5`.

Extensión: Otra razón para separar los registros enteros y de punto flotante es que los microprocesadores en los años 80 no tenían suficientes transistores para poner la unidad de punto flotante en el mismo circuito integrado que la unidad de enteros. Por ello, la unidad de punto flotante, incluidos los registros de punto flotante, estaba disponible de manera opcional en un segundo circuito integrado. Estos circuitos aceleradores se llaman *coprocesadores*, y explican el acrónimo de las instrucciones *load* de punto flotante de MIPS: `lwc1` significa cargar una palabra en el coprocesador 1, la unidad de punto flotante. (El coprocesador 0 se ocupa de la memoria virtual, descrita en el capítulo 5). Desde principios de los años 90, los microprocesadores integraron el punto flotante (y casi todo lo demás) en el mismo circuito integrado, y por ello el término "coprocesador", junto con "acumulador" y "memoria central", se añade a la lista de términos anticuados que delatan la edad de quien los usa.

Extensión: Como se ha mencionado en la sección 3.4, acelerar la división es más difícil que acelerar la multiplicación. Además de la SRT, otra técnica para aprovechar las ventajas de un multiplicador rápido es la *iteración de Newton*, en la que la división se convierte en la búsqueda del cero de una función para hallar el inverso $1/x$, el cual se multiplica por el otro operando. Las técnicas de iteración no se pueden redondear adecuadamente sin calcular muchos bits extra. Un chip de TI (*Texas Instruments*) resuelve este problema calculando un inverso extra-preciso.

Extensión: Java adopta el estándar IEEE 754 en su definición de tipos de datos y operaciones en punto flotante. Así, el código del primer ejemplo podría haber sido generado para un método de una clase que convirtiese Fahrenheit a Celsius.

El segundo ejemplo usa matrices multidimensionales, las cuales no están soportadas explícitamente en Java. Java permite vectores de vectores, pero cada vector puede tener su propia longitud, a diferencia de las matrices multidimensionales en C. Como los ejemplos del capítulo 2, una versión Java de este segundo ejemplo requeriría un buen trabajo de código de comprobación de los límites de las matrices, incluyendo un cálculo de una nueva longitud al final de cada fila. También necesitaría comprobar que las referencias a los objetos no son nulas.

Aritmética exacta

Guarda: el primero de los dos bits extra almacenados a la derecha durante los cálculos intermedios de números en punto flotante; se usa para mejorar la precisión del redondeo.

Redondeo: método para hacer que el resultado en punto flotante quepa en el formato en punto flotante; normalmente el objetivo es encontrar el número más cercano que se puede representar en el formato.

A diferencia de los enteros, que pueden representar con exactitud todos los números que hay entre el menor y el mayor, los números en punto flotante normalmente son aproximaciones de un número que no pueden representar. La razón es que entre dos reales (0 y 1, por ejemplo) hay siempre una cantidad infinita de números reales, pero en punto flotante de precisión doble no se pueden representar más de 2^{53} números. Lo mejor que se puede hacer es conseguir la representación en punto flotante más cercana al número real. Así, el IEEE 754 ofrece varios métodos de redondeo para permitir al programador elegir la aproximación deseada.

El redondeo parece bastante simple, pero para redondear de forma exacta es necesario que el hardware incluya bits extra en el cálculo. En los ejemplos precedentes no concretamos el número de bits que podían ocupar las representaciones intermedias, pero claramente si hubiera que truncar cada resultado intermedio al número exacto de cifras, entonces no habría oportunidad de redondear. El IEEE 754, por tanto, mantiene 2 bits adicionales a la derecha durante las sumas intermedias, llamados **guarda** y **redondeo**, respectivamente. Veamos un ejemplo en decimal para mostrar la importancia de estos bits adicionales.

Redondeo con dígitos de guarda

EJEMPLO

Sumar $2.56_{diez} \times 10^0$ a $2.34_{diez} \times 10^2$, suponiendo que tenemos 3 dígitos decimales significativos. Redondear al decimal más cercano con tres dígitos decimales, primero con dígitos de guarda y de redondeo, y luego sin ellos.

RESPUESTA

Primero debemos desplazar el número menor a la derecha para alinear los exponentes, de manera que $2.56_{diez} \times 10^0$ se convierte en $0.0256_{diez} \times 10^2$. Puesto que tenemos dígitos de guarda y redondeo, somos capaces de representar las dos cifras menos significativas cuando alineamos los exponentes. El dígito de guarda mantiene el 5, y el dígito de redondeo el 6. La suma es

$$\begin{array}{rl} & 2.3400_{diez} \\ + & 0.0256_{diez} \\ \hline & 2.3656_{diez} \end{array}$$

Por lo tanto, la suma es $2.3656_{diez} \times 10^2$. Puesto que tenemos dos dígitos para redondear, queremos redondear hacia abajo los valores entre 0 y 49, y hacia arriba los valores entre 51 y 99, siendo 50 el punto medio. Redondear hacia arriba la suma con tres dígitos significativos da $2.37_{diez} \times 10^2$.

Hacer esto sin dígitos de guarda ni redondeo deja dos cifras fuera del cálculo. La nueva suma es

$$\begin{array}{rl} & 2.34_{diez} \\ + & 0.02_{diez} \\ \hline & 2.36_{diez} \end{array}$$

La respuesta es $2.36_{diez} \times 10^2$, errónea en 1 unidad en el último dígito respecto a la suma anterior.

Puesto que el peor caso para el redondeo sería cuando el número verdadero estuviese exactamente a mitad de dos representaciones en punto flotante, la exactitud en punto flotante se mide, normalmente, en términos del número de bits de error en los bits menos significativos del significando; la medida se llama *número de* **unidades en el último lugar**, o **ulp** (*units in the last place*). Si un número estuviera equivocado en 2 unidades en los bits menos significativos, se diría que es erróneo en 2 ulps. Suponiendo que no hay desbordamiento, ni desbordamiento a cero, o excepciones de operación inválida, IEEE 754 garantiza que el computador usa el número que está dentro de media ulp.

Unidades en el último lugar (ulp): número de bits erróneos en los bits menos significativos del significando entre el número real y el número que puede ser representado.

Extensión: Aunque el ejemplo anterior en realidad necesitaba justo un dígito extra, la multiplicación puede necesitar dos. Un producto binario puede tener un 0 inicial, lo que significa que el paso de normalización debe desplazar el producto un bit a la izquierda. Esto desplaza el bit de guarda a la posición del bit menos significativo del producto, haciendo que sea el bit de redondeo el que ayude a redondear con precisión el producto.

IEEE 754 tiene 4 modos de redondeo: redondear siempre hacia arriba (hacia $+\infty$), siempre hacia abajo (hacia $-\infty$), truncar, y redondear al par más próximo. El último modo es el que determina qué hacer si el número está exactamente a la mitad de dos representaciones. En Estados Unidos, el organismo recaudador de impuestos siempre redondea 0.50 dólares al alza, posiblemente para beneficio del propio organismo. Una manera más equitativa sería redondear al alza la mitad de las veces y a la baja la otra mitad. El IEEE 754 dice que si el bit menos significativo guardado es impar, se añade uno y, si es par, se trunca. Este método siempre crea un 0 en el bit menos significativo en el caso de desempate, y de aquí el nombre de este modo de redondeo, que es el que se usa con mayor frecuencia y el único que soporta Java.

El objetivo de los bits adicionales para redondeo es conseguir que el computador obtenga el mismo resultado que se obtendría si los resultados intermedios fuesen calculados con precisión infinita y luego redondeados. Para dar soporte a este objetivo y redondear al próximo par, el estándar tiene un tercer bit, además de los de guarda y redondeo, que se actualiza a 1 cuando hay bits no nulos a la derecha del bit de redondeo. Este **bit pegajoso** (*sticky bit*) permite al computador ver la diferencia entre $0.50\ldots00_{diez}$ y $0.50\ldots01_{diez}$ cuando se redondea.

Bit sticky: bit usado para redondear, además de los bits de guarda y redondeo, que se activa a 1 siempre que hay bits no nulos a la derecha del bit de redondeo.

El bit sticky se activa a 1, por ejemplo, durante la suma, cuando el número menor se desplaza a la derecha. Supongamos que sumamos $5.01_{diez} \times 10^{-1}$ a $2.34_{diez} \times 10^{2}$ en el ejemplo anterior. Incluso con guarda y redondeo, estaríamos sumando 0.0050 a 2.34, con un resultado de 2.3450. El bit sticky se activaría porque hay bits no nulos a la derecha. Sin el bit sticky para recordar que hay algún 1 que se desplazó fuera, supondríamos que el número es igual a 2.345000...00, y redondeando al próximo par daría 2.34. Con el bit sticky, que recuerda que el número es mayor que 2.345000...000, redondearíamos a 2.35.

Extensión: Las arquitecturas PowerPC, SPARC64 y AMD SSE5 tienen un instrucción para hacer una multiplicación y una suma con tres registros: $a = a + (b \times c)$. Obviamente, esta instrucción permite, potencialmente, unas mayores prestaciones punto flotante para esta operación tan frecuente. Igualmente importante es que en lugar de hacer dos redondeos —después de la multiplicación y después de la suma— como ocurriría si se implementa con dos instrucciones diferentes, la instrucción de multiplicación-suma sólo hace un redondeo después de la suma, y esto aumenta la precisión de la multiplicación-suma. Esta operación con un solo redondeo se llama **multiplicación-suma fusionada** (*fused multiply add*) y se ha incorporado a la revisión del estándar IEEE754 (véase **sección 31.0** en el CD).

Multiplicación-suma fusionada (*fused multiply add*): instrucción punto flotante que realiza la multiplicación y la suma, con un solo redondeo después de la suma.

Resumen

La siguiente *Idea clave* refuerza el concepto de programa almacenado del capítulo 2; el significado de la información no se puede determinar simplemente mirando los bits, porque los mismos bits pueden representar diversos objetos. Esta sección muestra que la aritmética del computador es finita, y por ello puede no coincidir con la aritmética natural. Por ejemplo, la representación en punto flotante del estándar IEEE 754

$$(-1)^{S} \times (1 + \text{mantisa}) \times 2^{(\text{exponente} - \text{sesgo})}$$

es casi siempre una aproximación al número real. Los sistemas de computación deben tener cuidado para minimizar este desfase entre la aritmética del computador y la aritmética del mundo real, y los programadores a veces necesitan estar atentos a las implicaciones de esta aproximación.

IDEA clave

Las tiras de bits no tienen ningún significado inherente. Pueden representar números con signo, sin signo, en punto flotante, instrucciones, etc. Lo que se representa depende de la instrucción que opera con los bits de esa palabra.

La mayor diferencia entre los números del computador y los del mundo real es que los números del computador están limitados en su tamaño, lo cual limita su precisión; es posible calcular un número demasiado grande o demasiado pequeño para representarlo en una palabra. Los programadores deben recordar estos límites y escribir los programas consecuentemente.

Tipo C	Tipo Java	Transferencia de datos	Operaciones
`int`	`int`	`lw, sw, lui`	`addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti`
`unsigned int`	—	`lw, sw, lui`	`addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu`
`char`	—	`lb, sb, lui`	`addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu`
—	`char`	`lh, sh, lui`	`addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu`
`float`	`float`	`lwc1, swc1`	`add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s`
`double`	`double`	`l.d, s.d`	`add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d`

Interfaz hardware software

En el último capítulo presentamos las clases de almacenamiento del lenguaje de programación C (véase la sección "*Interfaz hardware software*" de la sección 2.7). La tabla anterior muestra algunos de los tipos de datos de C y Java junto con las instrucciones de transferencia de datos de MIPS y las instrucciones que operan sobre estos tipos que aparecen en el capítulo 2 y en este capítulo. Observe que Java omite los enteros sin signo.

Autoevaluación

Imaginemos un formato en punto flotante IEEE 754 de 16 bits con 5 bits de exponente. ¿Cuál sería el rango aproximado de números que podría representar?

1. $1.0000\ 0000\ 00 \times 2^{0}$ a $1.1111\ 1111\ 11 \times 2^{31}, 0$
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ a $\pm 1.1111\ 1111\ 1 \times 2^{15}, \pm 0, \pm\infty$, NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ a $\pm 1.1111\ 1111\ 11 \times 2^{15}, \pm 0, \pm\infty$, NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ a $\pm 1.1111\ 1111\ 11 \times 2^{14}, \pm 0, \pm\infty$, NaN

Extensión: Para ayudar a las comparaciones que pueden incluir NaNs, el estándar incluye opciones de comparación *con orden* y *sin orden*. Por ello, todo el repertorio de instrucciones tiene muchas formas de comparación para soportar NaNs. (Java no soporta las comparaciones no ordenadas).

En un intento de exprimir hasta el último bit de precisión en una operación en punto flotante, el estándar permite que algunos números se representen en forma no normalizada. En lugar de tener un hueco entre 0 y el número normalizado más pequeño, IEEE permite los números desnormalizados (también llamados, en inglés, *denorms* o *subnormals*). Tienen el mismo exponente que el 0 pero una mantisa no cero. Permiten reducir paulatinamente la magnitud de un número hasta llegar a ser 0, cualidad llamada desbordamiento a cero gradual. Por ejemplo, el menor número positivo normalizado en precisión simple es

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{dos} \times 2^{-126}$$

pero el menor número no normalizado en precision simple es

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{dos} \times 2^{-126}, \text{ o } 1.0_{dos} \times 2^{-149}$$

Para precisión doble, el hueco al no normalizar va de 1.0×2^{-1022} a 1.0×2^{-1074}.

La posibilidad de un operando no normalizado ocasional ha dado muchos quebraderos de cabeza a lo diseñadores de punto flotante que intentan hacer unidades en punto flotante rápidas. Por ello, muchos computadores provocan una excepción si un operando no está normalizado, y dejan que el software complete la operación. Aunque las implementaciones en software son perfectamente válidas, sus menores prestaciones han reducido la popularidad de los números *denorm* en software en punto flotante portable. Además, si los programadores no esperan *denorms*, sus programas pueden tener sorpresas.

3.6 Paralelismo y aritmética del computador: asociatividad

Los programas, típicamente, han sido escritos para ejecutarse secuencialmente antes de reescribirlos para una ejecución concurrente, de modo que la pregunta natural es "¿obtienen las dos versiones los mismos resultados?". Si la respuesta es no, se supone que hay algún error en la versión paralela que debe ser localizado.

Actuando de esta forma se asume que al pasar de la versión secuencial a la paralela, el resultado no se ve afectado por la aritmética del computador. Es decir, si se suman un millón de números, el resultado es el mismo usando 1 procesador o 1000 procesadores. Esta suposición se cumple para números enteros en complemento a 2, incluso si se produce desbordamiento. Dicho de otra forma, la suma de enteros es asociativa.

Desgraciadamente, esta afirmación no se mantiene para número en representación de punto flotante, porque son aproximaciones de los número reales y la aritmética del computador tiene precisión limitada. Es decir, la suma en punto flotante no es asociativa.

EJEMPLO

Representación en punto flotante

Veamos si $x + (y + z) = (x + y) + z$. Supongamos $x = -1.5_{diez} \times 10^{38}$, $y = 1.5_{diez} \times 10^{38}$, y $z = 1.0$ y que son números en precisión simple.

RESPUESTA

Debido a que el intervalo de números que pueden representarse en punto flotante es muy grande, pueden aparecer problemas cuando se suman dos número muy grandes de distinto signo más un número pequeño, como se verá a continuación:

$$
\begin{aligned}
x + (y + z) &= -1.5_{\text{diez}} \times 10^{38} + (1.5_{\text{diez}} \times 10^{38} + 1.0) \\
&= -1.5_{\text{diez}} \times 10^{38} + (1.5_{\text{diez}} \times 10^{38}) = 0.0 \\
(x + y) + z &= (-1.5_{\text{diez}} \times 10^{38} + 1.5_{\text{diez}} \times 10^{38}) + 1.0 \\
&= (0.0_{\text{diez}}) + 1.0 \\
&= 1.0
\end{aligned}
$$

Por lo tanto $x + (y + z) \neq (x + y) + z$, es decir, la suma punto flotante no es asociativa.

Puesto que los números punto flotante tienen precisión limitada y son aproximaciones de número reales, $1.5_{\text{diez}} \times 10^{38}$ es tan grande respecto a 1.0_{diez} que $1.5_{\text{diez}} \times 10^{38} + 1.0$ es todavía $1.5_{\text{diez}} \times 10^{38}$. Este es el motivo de que la suma de *x*, *y* y *z* sea 0.0 o 1.0 dependiendo del orden en que se hacen las sumas en punto flotante; en consecuencia la suma punto flotante *no* es asociativa.

Una versión aún más desconcertante de este error habitual se produce en computadores paralelos en los que el planificador del sistema operativo utiliza un número diferente de procesadores dependiendo de qué programas estén ejecutándose en el computador. Un programador de sistemas paralelos no cuidadoso puede quedar desconcertado al ver que su programa proporciona respuestas ligeramente diferentes cada vez que se ejecuta, aún siendo el mismo código y las mismas entradas, porque al variar el número de procesadores de una ejecución a otra causaría que las sumas en punto flotante se calculen en un orden diferente.

Dado este dilema, los programadores que escriben código paralelo con números punto flotante deben verificar si los resultados son creíbles incluso si no son exactamente igual a los del código secuencial. El campo que aborda estos aspectos se llama análisis numérico, y hay libros expresamente dedicados al mismo. Estos dilemas son una de las razones de la popularidad de las bibliotecas numéricas, como LAPACK y SCALAPACK, que han sido validadas tanto en su versión secuencial como en su versión paralela.

Extensión: Una versión sutil del problema de la asociatividad se presenta cuando dos procesadores hacen una computación redundante que se ejecutan en orden diferente, de forma que cada uno obtiene resultados ligeramente diferentes, aunque ambas respuestas sean consideras correctas. El error se produce si un salto condicional compara con un número punto flotante y los dos procesadores toman salidas diferentes, cuando el sentido común dice que deberían tomar el mismo camino.

3.7 Caso real: punto flotante en el x86

El x86 tiene instrucciones normales para multiplicar y dividir que operan enteramente sobre registros, a diferencia de la dependencia de *Hi* y *Lo* de MIPS. (De hecho, versiones posteriores del repertorio de instrucciones MIPS incorporan instrucciones similares).

Las principales diferencias se encuentran en las instrucciones de punto flotante. La arquitectura en punto flotante de x86 es diferente de las de todos los demás computadores del mundo.

La arquitectura en punto flotante del x86

El coprocesador en punto flotante Intel 8087 se anunció en 1980. Esta arquitectura extendía el 8086 con unas 60 instrucciones en punto flotante.

Intel proporcionó una arquitectura tipo pila para sus instrucciones en punto flotante: las cargas apilaban números en la pila, las operaciones buscaban los operandos en los dos elementos del tope de la pila, y los almacenamientos sacaban elementos de la pila. Intel complementó esta arquitectura con instrucciones y modos de direccionamiento que le permitían obtener algunas de las ventajas del modelo memoria-registro. Además de buscar los operandos en los dos elementos del tope de la pila, un operando puede estar en memoria o en uno de los siete registros debajo del tope de la pila internos al chip. De este modo, un repertorio de instrucciones completo tipo pila se complementa con un repertorio limitado de instrucciones memoria-registro.

Sin embargo, este híbrido aún es un modelo de memoria-registro restringido, puesto que las cargas siempre mueven datos al tope de la pila, incrementando el puntero del tope de la pila, y los almacenamientos sólo pueden mover el tope de la pila a memoria. Intel usa la notación `ST` para indicar el tope de la pila, y `ST(i)` para representar el *i*-ésimo registro por debajo del tope de la pila.

Otra característica novedosa de esta arquitectura es que los operandos son más largos en la pila de registros que cuando se almacenan en memoria, y todas las operaciones se realizan con esta precisión interna mayor. A diferencia del máximo de 64 bits en MIPS, los operandos en punto flotante de x86 en la pila tienen una longitud de 80 bits. Los números se convierten automáticamente al formato interno de 80 bits cuando se cargan y se reconvierten al tamaño adecuado en los almacenamientos. Esta *precisión doble extendida* no es soportada por los lenguajes de programación, aunque ha sido útil a los programadores de software matemático.

Los datos en memoria pueden ser de 32 bits (precisión simple) o 64 bits (precisión doble) para números en punto flotante. La versión memoria-registro de estas instrucciones convierte automáticamente el operando de memoria a este formato de 80 bits de Intel antes de realizar la operación. Las instrucciones de transferencia de datos también convierten automáticamente los enteros de 16 y 32 bits a punto flotante, y viceversa, en las cargas y almacenamientos de enteros.

Las operaciones en punto flotante x86 se pueden dividir en cuatro grandes grupos:

1. Instrucciones de movimiento de datos, que incluyen carga, carga constante y almacenamiento
2. Instrucciones aritméticas, que incluyen suma, resta, multiplicación, división, raíz cuadrada y valor absoluto
3. Comparación, que incluye instrucciones de envío del resultado al procesador entero para que pueda saltar
4. Instrucciones transcendentes, que incluyen las funciones seno, coseno, logaritmo y exponenciación

La figura 3.20 muestra algunas de las 60 operaciones en punto flotante. Obsérvese que se dan incluso más combinaciones cuando se incluyen los modos de operando para estas operaciones. La figura 3.21 muestra las diversas opciones para la suma en punto flotante.

Transferencia de datos	Aritméticas	Comparación	Transcendentales
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

FIGURA 3.20 Las instrucciones en punto flotante de x86. Se usan las llaves {} para mostrar las variaciones opcionales de las operaciones básicas: {I} significa que hay una versión entera de la instrucción, {P} significa que esta versión sacará un operando de la pila después de la operación, y {R} significa invertir el orden de los operandos en esta operación. La primera columna muestra las instrucciones de tranferencia de datos, las cuales mueven datos a memoria o a uno de los registros que están por debajo del tope de la pila. Las tres últimas operaciones de la primera columna cargan constantes en la pila: pi, 1.0 y 0.0. La segunda columna contiene las operaciones aritméticas descritas más arriba. Observe que las tres últimas operan sólo con el tope de la pila. La tercera columna son la instrucciones de comparación. Puesto que no hay instrucciones de salto específicas para punto flotante, el resultado de una comparación debe transferirse a la CPU de enteros vía la instrucción FSTSW, tanto al registro AX como a memoria, seguida por la instrucción SAHF para activar los códigos de condición. Entonces, se puede comprobar la comparación en punto flotante usando instrucciones de salto de enteros. La última columna muestra las operaciones en punto flotante de mayor nivel. No se soportan todas las operaciones sugeridas por la notación. Así, las operaciones F{I}SUB{R}{P} representan estas instrucciones del x86: FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP. Para las instrucciones de resta entera, no hay versión con *pop* (FISUBP) ni *pop* inverso (FISUBRP).

Las instrucciones en punto flotante se codifican usando el código de operación ESC del 8086 y el especificador de dirección postbyte (véase figura 2.47). Las operaciones de memoria reservan dos bits para decidir si el operando es un punto flotante de 32 o 64 bits o un entero de 16 o 32 bits. Estos dos mismos bits se utilizan

Instrucción	Operandos	Comentarios
FADD		Ambos operandos en la pila; el resultado reemplaza el tope de la pila.
FADD	ST(i)	Un operando fuente es el *i*-ésimo registro por debajo del tope de la pila; el resultado reemplaza el tope de la pila.
FADD	ST(i), ST	Un operando fuente es el tope de la pila; el resultado reemplaza el *i*-ésimo registro por debajo del tope de la pila.
FADD	mem32	Un operando fuente es una posición en memoria de 32 bits; el resultado reemplaza el tope de la pila.
FADD	mem64	Un operando fuente es una posición en memoria de 64 bits; el resultado reemplaza el tope de la pila.

FIGURA 3.21 Las variaciones de operandos para la suma en punto flotante del x86.

en versiones que no acceden a memoria para decidir si se debe sacar un dato de la pila después de la operación y si es el tope de la pila o un registro inferior el que debe recibir el resultado.

Las prestaciones del punto flotante de la familia x86 tradicionalmente ha quedado rezagado respecto a otros computadores. Por este motivo, Intel creó una arquitectura en punto flotante más tradicional como parte de SSE2.

La arquitectura en punto flotante Streaming SIMD Extensión 2 (SSE2) de Intel

En el capítulo 2 se indica que en el año 2001 Intel añadió 144 instrucciones a su arquitectura, que incluían operaciones y registros en punto flotante de precisión doble. Incluye ocho registros de 64 bits que se pueden usar para operandos en punto flotante, dando al compilador una alternativa diferente para las operaciones en punto flotante que la arquitectura de pila única. Los compiladores pueden escoger utilizar los ocho registros SSE2 como registros en punto flotante como los que se encuentran en otros computadores. AMD amplió el número a 16 como parte de AMD64, que fue renombrado como EM64T por Intel. La figura 3.22 resume las instrucciones SSE y SSE2.

Además de almacenar un número punto flotante de precisión simple o doble en un registro, Intel permite que varios operandos punto flotante se empaqueten en un único registro SSE2 de 128 bits: cuatro de precisión simple o dos de precisión doble. Así, los 16 registros punto flotante del SSE2 son realmente de 128 bits. Si los operandos se pueden disponer en memoria como datos alineados de 128 bits, entonces las transferencias de datos de 128 bits pueden cargar o almacenar varios operandos por instrucción. Las operaciones aritméticas admiten este formato de punto flotante empaquetado, ya que pueden operar simultáneamente sobre cuatro simples o dos dobles. Esta nueva arquitectura puede más que doblar las prestaciones de la arquitectura de pila.

Transferencia de datos	Aritmética	Comparación
MOV {A/U} {SS/PS/SD/PD} xmm, mem/xmm	ADD {SS/PS/SD/PD} xmm, mem/xmm	CMP {SS/PS/SD/PD}
MOV {H/L} {PS/PD} xmm, mem/xmm	SUB{SS/PS/SD/PD} xmm, mem/xmm	
	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURA 3.22 Las instrucciones punto flotante SSE/SSE2 del x86. xmm significa que un operando es un registro SSE2 de 128 bits, y mem/xmm que el otro operando o bien está en memoria o bien está en un registro SSE2. Usamos las llaves {} para mostrar variantes opcionales de las operaciones básicas: {SS} significa punto flotante de precisión simple escalar, o un operando de 32 bits en un registro de 128 bits; {PS} significa punto flotante de precisión simple empaquetado, o cuatro operandos de 32 bits en un registro de 128 bits; {SD} significa punto flotante de precisión doble escalar, o un operando de 64 bits en un registro de 128 bits; {PD} significa punto flotante de precisión doble empaquetado, o dos operandos de 32 bits en un registro de 128 bits; {A} significa que el operando de 128 bits está alineado en memoria; {U} significa que el operando de 128 bits no está alineado en memoria; {H} significa mover la mitad superior del operando de 128 bits; {L} significa mover la mitad inferior del operando de 128 bits.

3.8 Falacias y errores habituales

Entonces las matemáticas se pueden definir como aquella materia en la que nunca sabemos de lo que hablamos, ni si lo que decimos es cierto.

BERTRAND RUSSELL, *Recent Words on the Principles of mathematics*, 1901

Las falacias y errores en aritmética generalmente surgen de la diferencia entre la precisión limitada de la aritmética del computador y la precisión ilimitada de la aritmética natural.

Falacia: del mismo modo que una instrucción de desplazamiento a la izquierda puede sustituir una multiplicación entera por una potencia de 2, un desplazamiento a la derecha es lo mismo que una división entera por una potencia de 2.

Recordemos que un número binario x, donde xi significa el i–ésimo bit, representa el número

$$\ldots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0)$$

Desplazar los bits de x a la derecha n bits parecería ser lo mismo que dividir por 2^n. Y esto es cierto para enteros sin signo. El problema está con los enteros con signo. Por ejemplo, supongamos que queremos dividir -5_{diez} entre 4_{diez}; el cociente debería ser -1_{diez}. La representación en complemento a dos de -5_{diez} es

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{dos}$$

Según esta falacia, desplazar a la derecha dos posiciones debería dividir por 4_{diez} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos}$$

Con un 0 en el bit de signo, este resultado es claramente erróneo. El valor creado por el desplazamiento a la derecha es en realidad $1\,073\,741\,822_{diez}$ en lugar de -1_{diez}.

Una solución sería tener un desplazamiento a la derecha aritmético que extiende el signo en lugar de desplazar con 0s. Un desplazamiento aritmético de dos bits a la derecha de -5_{diez} produce

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{dos}$$

El resultado es -2_{diez} en lugar de -1_{diez}; cercano, pero no correcto.

Error: la instrucción de suma con inmediato sin signo de MIPS (add immediate unsigned, `addiu`*) extiende el signo de su campo inmediato de 16 bits.*

A pesar de su nombre, `addiu` se usa para sumar constantes a enteros con signo cuando no se tiene en cuenta el desbordamiento. MIPS no tiene una instrucción de resta con inmediato y los números negativos necesitan la extensión de signo, de manera que los arquitectos de MIPS decidieron extender el signo del campo inmediato.

Falacia: sólo los matemáticos teóricos se preocupan de la precisión en punto flotante.

Los titulares de los periódicos de noviembre de 1994 demuestran la falsedad de esta afirmación (véase figura 3.23). A continuación está la historia oculta que se encuentra tras estos titulares.

El Pentium usa un algoritmo estándar de división en punto flotante que genera múltiples bits del cociente por paso, usando los bits más significativos del divisor y del dividendo para estimar los 2 bits siguientes del cociente. La estimación se toma de una tabla de búsqueda que contiene –2, –1, 0, +1 ó +2. La predicción se multiplica por el divisor y se resta del resto para generar un nuevo resto. Igual que la división sin restauración, si una predicción previa da un resto demasiado grande, el resto parcial se ajusta en el siguiente paso.

Evidentemente, había cinco elementos de la tabla del 80486 a los que Intel pensó que nunca se accedería, y en el Pentium optimizaron la PLA para que devolviera 0 en lugar de 2 en estas situaciones. Intel se equivocaba: mientras los 11 primeros bits eran siempre correctos, se podían producir errores ocasionales en los bits 12 a 52, o de la 4ª a la 15ª cifra decimal.

A continuación reconstruimos cronológicamente los hechos del desaguisado del error del Pentium:

- *Julio de 1994:* Intel descubre el error en el Pentium. El coste real para arreglarlo asciende a varios cientos de miles de dólares. Siguiendo los procedimientos normales para corregir errores, se tardaría meses en hacer el cambio, volver a verificar y poner el chip correcto en producción. Intel planificó poner chips correctos en producción en enero de 1995, estimando que se fabricarían entre 3 y 5 millones de Pentiums con el error.

FIGURA 3.23 Recortes de artículos de revista y periódicos de noviembre de 1994, que incluyen el *New York Times*, *el San Jose Mercury News*, *el San Francisco Chronicle* y el *Infoworld*. El error en la división en punto flotante del Pentium llegó a estar en el número uno en la lista de los "Top 10" del *show* televisivo de *David Letterman*. Intel finalmente gastó unos 300 millones de dólares para reemplazar los chips defectuosos.

- *Septiembre de 1994:* Thomas Nicely, profesor de matemáticas en el Lynchburg College de Virginia, descubre el error. Después de llamar al soporte técnico de Intel y no obtener reacción oficial alguna, informa el descubrimiento a través de Internet. La noticia corre como la pólvora y algunos señalan que incluso los errores pequeños se convierten en grandes cuando se multiplican por números grandes: la proporción de gente con una enfermedad rara multiplicada por la población de Europa, por ejemplo, podría conducir a una estimación incorrecta del número de personas enfermas.
- *7 de noviembre de 1994: Electronic Engineering Times* pone la historia en su portada, y rápidamente otros periódicos la imitan.
- *22 de noviembre de 1994:* Intel publica una nota de prensa, considerándolo un "pequeño error" (*glitch*). "El Pentium puede tener errores en la décima cifra… La mayoría de los ingenieros y analistas financieros sólo necesitan precisión hasta la cuarta o quinta cifra decimal. Los usuarios de hojas de cálculo y procesadores de texto no tienen por qué preocuparse… Habrá unas cuantas docenas de personas a quienes pueda afectar. Hasta ahora sólo hemos tenido noticias de una… [Sólo] los matemáticos teóricos (con

máquinas Pentium compradas antes del verano) podrían verse afectados." Lo que más molestó a muchos usuarios fue que se les pidió que describieran sus aplicaciones a Intel y luego Intel decidiría si su aplicación merecía un Pentium nuevo sin el error de división.

- *5 de diciembre de 1994:* Intel asegura que el error se produce una vez cada 27 000 años para los usuarios habituales de hojas de cálculo. Intel supone que un usuario hace 1000 divisiones por día y multiplica la tasa de error suponiendo que los números en punto flotante son aleatorios, lo que es uno entre 9000 millones, con lo que obtiene 9 millones de días, o 27 000 años. La situación empieza a calmarse, a pesar de que Intel no se preocupó de explicar por qué un usuario típico accedería a números en punto flotante de forma aleatoria.

- *12 de diciembre de 1994:* la división de investigación de IBM cuestiona los cálculos de la tasa de errores (este artículo está disponible en *www.mkp.com/books_catalog/cod/links.htm*). IBM asegura que las hojas de cálculo habituales, haciedo cálculos durante 15 minutos al día, podrían producir errores relacionados con el Pentium con una frecuencia de uno cada 24 días. IBM supone 5000 divisiones por segundo, durante 15 minutos, lo que da 4.2 millones de divisiones por día, y no tiene en cuenta la distribución aleatoria, calculando en su lugar las posibilidades como una entre 100 millones. Como resultado, IBM detiene inmediatamente la producción de computadores personales basados en el Pentium. Las cosas se ponen de nuevo mal para Intel.

- *21 de diciembre de 1994:* Intel publica lo siguiente, firmado por el presidente de Intel y varios cargos directivos:

  "Intel desea disculparse sinceramente por su manera de llevar el tema del error del procesador Pentium recientemente aparecido. El símbolo *Intel Inside* significa que su computador tiene un microprocesador sin igual en calidad y prestaciones. Miles de empleados en Intel trabajan muy duro para asegurar la veracidad de esta afirmación. Pero ningún microprocesador es perfecto. Lo que Intel continúa considerando un problema tecnológico extremadamente minúsculo ha cobrado vida propia. Aunque Intel se mantiene firme en su confianza en la calidad de la versión actual del procesador Pentium, admitimos que muchos usuarios se han mostrados preocupados. Queremos resolver estas preocupaciones. Intel cambiará la versión actual del procesador Pentium por una actualizada, en la cual este error en la división en punto flotante esté corregido, para cualquier usuario que lo solicite, sin coste alguno durante la vida de su computador".

  Los analistas estimaron que esta operación costó a Intel unos 500 millones de dólares, y los empleados de Intel no obtuvieron aguinaldo de Navidad ese año.

Esta historia nos lleva a reflexionar sobre algunos puntos. ¿No habría sido más barato reparar el error en julio de 1994? ¿Cuál fue el coste de restituir la reputación de Intel? ¿Y cuál es la responsabilidad de la compañía cuando se descubren errores en un producto tan ampliamente usado y en el que se confía tanto como un microprocesador?

Instrucciones del núcleo del MIPS	Nombre	Formato	Núcleo aritmético del MIPS	Nombre	Formato
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
and	and	R	move from system control (EPC)	mfc0	R
and immediate	andi	I	floating-point add single	add.s	R
or	or	R	floating-point add double	add.d	R
or immediate	ori	I	floating-point subtract single	sub.s	R
nor	nor	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
branch on equal	beq	I	branch on floating-point false	bc1f	I
branch on not equal	bne	I	floating-point compare single	c.x.s	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J	floating-point compare double	c.x.d	R
jump register	jr	R	(x = eq, neq, lt, le, gt, ge)		
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURA 3.24 Instrucciones de MIPS vistas hasta el momento. Este libro se centra en las instrucciones de la columna izquierda. Esta información también se encuentra en las columnas 1 y 2 de la tarjeta de referencia MIPS que se encuentra al principio de este libro.

En abril de 1997 se encontró otro error en el punto flotante de los microprocesadores Pentium Pro y Pentium II. Cuando las instrucciones de almacenamiento punto flotante a entero (`fist`, `fistp`) encontraban un número en punto flotante que era demasiado grande para caber en una palabra de 16 o 32 bits tras convertirlo a entero, activaban un bit equivocado en la palabra de estado de FPO (excepción de precisión en lugar excepción de operación no válida). Para honra de Intel, esta vez reconocieron públicamente el error y ofrecieron un software para evitarlo, una reacción bastante diferente de la que tuvieron en 1994.

3.9 Conclusiones finales

Un efecto lateral de los computadores de programa almacenado es que las tiras de bits no tienen un significado propio. La misma tira de bits puede representar un entero con signo, un entero sin signo, un número en punto flotante, una instrucción, etc. Es la instrucción que opera con una palabra la que determina su significado.

La aritmética del computador se distingue de la aritmética de lápiz y papel por las restricciones de la precisión limitada. Este límite puede producir operaciones erróneas al calcular números mayores o menores que los límites predefinidos. Tales anomalías, llamadas "desbordamiento" y "desbordamiento a cero", pueden provocar excepciones o interrupciones, que son casos de emergencia similares a las llamadas a subrutinas no planificadas. El capítulo 4 trata las excepciones con más detalle.

La aritmética en punto flotante tiene la dificultad añadida de ser una aproximación de los números reales, y se necesita tener cuidado para poder asegurar que el número elegido por el computador sea la representación más cercana al número real. Los desafíos de la imprecisión y la representación limitada son algunas de las cuestiones que inspiran el campo del análisis numérico. El cambio reciente al paralelismo hará brillar otra vez el faro del análisis numérico, porque las soluciones que fueron consideradas seguras en secuencial durante largo tiempo deberán ser reconsideradas cuando se busque el algoritmo más rápido para computadores paralelos que obtenga resultados correctos.

A lo largo de los años, la aritmética del computador se ha ido estandarizando, aumentado así en gran medida la portabilidad de los programas. La aritmética de enteros en complemento a dos y la aritmética en punto flotante IEEE 754 se encuentran en la gran mayoría de los computadores que se venden hoy en día. Por ejemplo, todos los computadores de sobremesa que se han vendido desde la aparición de este libro siguen estas convenciones.

Junto con la explicación de la aritmética del computador, en este capítulo se describe la mayor parte del repertorio de instrucciones de MIPS. Un punto de confusión son las diferencias entre las instrucciones que se ven en este capítulo, las instrucciones que realmente ejecutan los chips de MIPS y las instrucciones que aceptan los ensambladores de MIPS. Las dos figuras siguientes intentan aclarar esta cuestión.

La figura 3.24 lista las instrucciones MIPS cubiertas en este capítulo y el capítulo 2. Al conjunto de instrucciones del lado izquierdo las llamamos *núcleo de MIPS*. Las instrucciones del lado derecho se llaman *núcleo aritmético de MIPS*. A la izquierda de la figura 3.25 están las instrucciones que el procesador MIPS ejecuta y que no están en la figura 3.24. A éstas las llamamos *el repertorio completo de instrucciones hardware MIPS-32*. A la derecha de la figura 3.25 están las instrucciones aceptadas por el ensamblador que no son parte de MIPS-32. A éstas las llamamos *repertorio de instrucciones pseudo MIPS*.

La figura 3.26 muestra la frecuencia de aparición de las instrucciones MIPS en los programas de prueba de enteros y punto flotante SPEC2006. Se listan todas las instrucciones que representaban al menos un 0,3% de las instrucciones totales ejecutadas. La siguiente tabla resume esta información.

Resto del MIPS-32	Nombre	Formato	Pseudo MIPS	Nombre	Formato
exclusive or (*rs* ⊕ *rt*)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate *(signed or unsigned)*	neg*s*	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left *(unaligned)*	lwl	I	load address	la	rd,addr
load word right *(unaligned)*	lwr	I	load double	ld	rd,addr
store word left *(unaligned)*	swl	I	store double	sd	rd,addr
store word right *(unaligned)*	swr	I	unaligned load word	ulw	rd,addr
load linked (*atomic update*)	ll	I	unaligned store word	usw	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or *uns.*)	madd*s*	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or *uns.*)	msub*s*	I	branch on compare *(signed or unsigned)*	b*x**s*	rs,rt,L
branch on ≥ zero and link	bgezal	I	(x = lt, le, gt, ge)		
branch on < zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare *(signed or unsigned)*	s*x**s*	rd,rs,rt
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			load to floating point (s or d)	l.*f*	rd,addr
branch compare reg likely	bxl	I	store from floating point (s or d)	s.*f*	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
(x = eq, neq, lt, le, gt, ge)					
return from exception	rfe	R			
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move (s or d)	mov.*f*	R			
FP move if zero (s or d)	movz.*f*	R			
FP move if not zero (s or d)	movn.*f*	R			
FP square root (s or d)	sqrt.*f*	R			
FP absolute value (s or d)	abs.*f*	R			
FP negate (s or d)	neg.*f*	R			
FP convert (w, s, or d)	cvt.*f*.*f*	R			
FP compare un (s or d)	c.xn.*f*	R			

FIGURA 3.25 Resto del repertorio de instrucciones de MIPS-32 y "pseudo MIPS". La presencia de una *f* significa la existencia de una versión de la instrucción en punto flotante de precisión simple (s) o precisión doble (d), mientras que una *s* indica la existencia de versiones con signo y sin signo (u). MIPS-32 también tiene instrucciones en punto flotante para multiplicar y sumar/restar (madd, *f*/msub.*f*), techo (ceil.*f*), truncar (trunc.*f*), redondeo (round.*f*) y recíproco (recip.*f*). El subrayado indica la letra que se debe incluir para representar el tipo de dato.

Núcleo del MIPS	Nombre	Entero	Pt. ft.	Núcleo aritmético + MIPS-32	Nombre	Entero	Pt. ft.
add	add	0.0%	0.0%	FP add double	add.d	0.0%	10.6%
add immediate	addi	0.0%	0.0%	FP subtract double	sub.d	0.0%	4.9%
add unsigned	addu	5.2%	3.5%	FP multiply double	mul.d	0.0%	15.0%
add immediate unsigned	addiu	9.0%	7.2%	FP divide double	div.d	0.0%	0.2%
subtract unsigned	subu	2.2%	0.6%	FP add single	add.s	0.0%	1.5%
AND	AND	0.2%	0.1%	FP subtract single	sub.s	0.0%	1.8%
AND immediate	ANDi	0.7%	0.2%	FP multiply single	mul.s	0.0%	2.4%
OR	OR	4.0%	1.2%	FP divide single	div.s	0.0%	0.2%
OR immediate	ORi	1.0%	0.2%	load word to FP double	l.d	0.0%	17.5%
NOR	NOR	0.4%	0.2%	store word to FP double	s.d	0.0%	4.9%
shift left logical	sll	4.4%	1.9%	load word to FP single	l.s	0.0%	4.2%
shift right logical	srl	1.1%	0.5%	store word to FP single	s.s	0.0%	1.1%
load upper immediate	lui	3.3%	0.5%	branch on floating-point true	bc1t	0.0%	0.2%
load word	lw	18.6%	5.8%	branch on floating-point false	bc1f	0.0%	0.2%
store word	sw	7.6%	2.0%	floating-point compare double	c.x.d	0.0%	0.6%
load byte	lbu	3.7%	0.1%	multiply	mul	0.0%	0.2%
store byte	sb	0.6%	0.0%	shift right arithmetic	sra	0.5%	0.3%
branch on equal (zero)	beq	8.6%	2.2%	load half	lhu	1.3%	0.0%
branch on not equal (zero)	bne	8.4%	1.4%	store half	sh	0.1%	0.0%
jump and link	jal	0.7%	0.2%				
jump register	jr	1.1%	0.2%				
set less than	slt	9.9%	2.3%				
set less than immediate	slti	3.1%	0.3%				
set less than unsigned	sltu	3.4%	0.8%				
set less than imm. uns.	sltiu	1.1%	0.1%				

FIGURA 3.26 Frecuencia de las instrucciones MIPS en los programas SPEC2006 de entero y punto flotante. En esta tabla se incluyen todas las instrucciones que obtuvieron al menos un 1%. Las pseudoinstrucciones se convierten a MIPS-32 antes de la ejecución, y por esto no aparecen aquí.

Observe que aunque los programadores y autores de compiladores podrían usar MIPS-32 para tener un menú de opciones más rico, las instrucciones núcleo de MIPS dominan la ejecución de enteros de SPEC2006, y el núcleo entero más el núcleo aritmético dominan el SPEC2006 de punto flotante, como se muestra en la siguiente tabla.

Subconjunto de instrucciones	Entero	Pt. ft.
Núcleo del MIPS	98%	31%
Núcleo aritmético del MIPS	2%	66%
Resto del MIPS-32	0%	3%

En el resto del libro, nos centraremos en las instrucciones del núcleo de MIPS –el repertorio de instrucciones enteras, excluyendo la multiplicación y la división– para hacer la explicación del diseño del computador más fácil. Como se puede ver, el núcleo de MIPS incluye las instrucciones más habituales, y asegurará que entender un computador que ejecuta el núcleo de MIPS dará suficiente bagaje para comprender computadores más ambiciosos.

3.10 Perspectiva histórica y lecturas recomendadas

La ley de Gresham ("El dinero malo expulsa al bueno") para los computadores diría "Lo rápido expulsa a lo lento, incluso aunque lo rápido sea erróneo".

W. Kahan, 1992

Esta sección examina la historia del punto flotante desde von Neumann, incluido el esfuerzo de estandarización sorprendentemente controvertido de IEEE, más el análisis razonado de la arquitectura de pila de 80 bits para punto flotante del x86. Véase sección 3.10.

3.11 Ejercicios

Contribución de Matthew Farrens, UC Davis

Nunca rendirse, nunca rendirse, nunca, nunca, nunca —en nada, pequeño o grande, importante o insignificante— nunca rendirse.

Winston Churchil, discurso en la Harrow School, 1941.

Ejercicio 3.1

El libro muestra cómo sumar y restar números binarios y números decimales. Sin embargo, hay otros sistemas de numeración que han sido también muy populares en el mundo de los computadores, por ejemplo el sistema de numeración octal (base 8). La siguiente tabla muestra parejas de números octales.

	A	B
a.	5323	2275
b.	0147	3457

3.1.1 [5]<3.2> ¿Cuál es el resultado de sumar A y B suponiendo que representan números octales de 12 bits sin signo? Dé el resultado en octal.

3.1.2 [5]<3.2> ¿Cuál es el resultado de sumar A y B suponiendo que representan números octales de 12 bits en formato signo-magnitud? Dé el resultado en octal.

3.1.3 [10]<3.2> Convierta A en un número decimal, suponiendo que es un número sin signo. Repita el problema suponiendo que es un número en formato signo-magnitud.

La siguiente tabla muestra parejas de números octales.

	A	B
a.	2162	2032
b.	2646	1066

3.1.4 [5]<3.2> ¿Cuál es el resultado de A – B suponiendo que representan números octales de 12 bits sin signo? Dé el resultado en octal.

3.1.5 [5]<3.2> ¿Cuál es el resultado de A – B suponiendo que representan números octales de 12 bits en formato signo-magnitud? Dé el resultado en octal.

3.1.6 [10]<3.2> Convierta A en un número decimal. ¿Por qué base 8 (octal) es un sistema de numeración interesante para representar valores en un computador?

Ejercicio 3.2

Otro sistema de numeración también ampliamente utilizado para representar números en un computador es el sistema hexadecimal (base 16). De hecho, es mucho más popular que base 8. La siguiente tabla muestra parejas de números hexadecimales.

	A	B
a.	OD34	DD17
b.	BA1D	3617

3.2.1 [5]<3.2> ¿Cuál es el resultado de sumar A y B suponiendo que representan números hexadecimales de 16 bits sin signo? Dé el resultado en hexadecimal.

3.2.2 [5]<3.2> ¿Cuál es el resultado de sumar A y B suponiendo que representan números hexadecimales de 16 bits en formato signo-magnitud? Dé el resultado en hexadecimal.

3.2.3 [10]<3.2> Convierta A en un número decimal, suponiendo que es un número sin signo. Repita el problema suponiendo que es un número en formato signo-magnitud.

La siguiente tabla muestra parejas de números hexadecimales.

	A	B
a.	BA7C	241A
b.	AADF	47BE

3.2.4 [5]<3.2> ¿Cuál es el resultado de A – B suponiendo que representan números hexadecimales de 16 bits sin signo? Dé el resultado en hexadecimal.

3.2.5 [5]<3.2> ¿Cuál es el resultado de A – B suponiendo que representan números hexadecimales de 16 bits en formato signo-magnitud? Dé el resultado en hexadecimal.

3.2.6 [10]<3.2> Convertir A en un número decimal. ¿Por qué base 16 (hexadecimal) es un sistema de numeración interesante para representar valores en un computador?

Ejercicio 3.3

Cuando un resultado es demasiado grande para ser representado correctamente con un tamaño dado de palabra finito se produce desbordamiento. Cuando número es demasiado pequeño para ser representado correctamente, por ejemplo un número negativo cuando se está trabajando con aritmética sin signo, se produce desbordamiento a cero. (Cuando la suma de dos números enteros negativos da un resultado positivo se considera en muchos textos como desbordamiento a cero, pero en este texto lo consideraremos un desbordamiento.) La siguiente tabla muestra parejas de números decimales.

	A	B
a.	69	90
b.	102	44

3.3.1 [5]<3.2> Calcule A – B suponiendo que A y B son números decimales enteros de 8 bits sin signo. ¿Se produce desbordamiento, desbordamiento a cero o ninguno de los dos?

3.3.2 [5]<3.2> Calcule A + B suponiendo que A y B son números decimales enteros de 8 bits en formato signo-magnitud. ¿Se produce desbordamiento, desbordamiento a cero o ninguno de los dos?

3.3.3 [5]<3.2> Calcule A – B suponiendo que A y B son números decimales enteros de 8 bits en formato signo-magnitud. ¿Se produce desbordamiento, desbordamiento a cero o ninguno de los dos?

La siguiente tabla muestra parejas de números decimales.

	A	B
a.	200	103
b.	247	237

3.3.4 [10]<3.2> Calcule A + B usando aritmética saturada suponiendo que A y B son números decimales enteros de 8 bits en complemento a 2. Dé el resultado en decimal.

3.3.5 [10]<3.2> Calcule A – B usando aritmética saturada suponiendo que A y B son números decimales enteros de 8 bits en complemento a 2. Dé el resultado en decimal.

3.3.6 [10]<3.2> Calcule A + B usando aritmética saturada suponiendo que A y B son números decimales enteros de 8 bits sin signo. Dé el resultado en decimal.

Ejercicio 3.4

Nos centramos ahora en la multiplicación. Usaremos los números de la siguiente tabla.

	A	B
a.	50	23
b.	66	04

3.4.1 [20]<3.3> Calcule el producto de A y B, octales enteros de 6 bits sin signo, utilizando una tabla similar a la de la figura 3.7 y el hardware descrito en la figura 3.4. Muestra el contenido de los registros en cada paso.

3.4.2 [20]<3.3> Calcule el producto de A y B, hexadecimales enteros de 8 bits sin signo, utilizando una tabla similar a la de la figura 3.7 y el hardware descrito en la figura 3.6. Muestre el contenido de los registros en cada paso.

3.4.3 [60]<3.3> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de dos números enteros sin signo A y B, utilizando el esquema de la figura 3.4.

La siguiente tabla muestra parejas de números octales.

	A	B
a.	54	67
b.	30	07

3.4.4 [30]<3.3> En la multiplicación de números con signo, una forma para obtener el resultado correcto es convertir el multiplicando y el multiplicador en números positivos y determinar el signo del resultado de acuerdo con los signos originales de los operandos. Calcule el producto de A y B utilizando una tabla similar a la de la figura 3.7 y el hardware descrito en la figura 3.4. Muestre el contenido de los registros en cada paso e incluya el paso necesario para obtener correctamente el signo del resultado. Suponga que A y B están en formato signo-magnitud de 6 bits con signo.

3.4.5 [30]<3.3> Al desplazar el contenido de un registro 1 bit a la derecha, hay varias formas para decidir cuál es el valor del bit que debe introducirse en el registro: siempre 0, siempre 1, o el bit que ha sido eliminado del registro por la derecha (convirtiendo el desplazamiento en una rotación), o el valor del bits más a la izquierda se mantiene (esto sería un desplazamiento aritmético a la derecha, porque se man-

tiene el signo del número que se está desplazando). Calcule el producto de A y B, números de 6 bits en complemento a 2, utilizando una tabla similar a la de la figura 3.7 y el hardware descrito en la figura 3.6. El desplazamiento a la derecha se realiza como un desplazamiento aritmético a la derecha. Observe que se tendrá que modificar ligeramente el algoritmo, en concreto, si el multiplicador es negativo. Se pueden encontrar más detalles buscando en la web. Muestre el contenido de los registros en cada paso.

3.4.6 [60]<3.3> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de dos números enteros con signo A y B. Indique si se están utilizando las alternativas planteadas en los ejercicios 3.4.4 o 3.4.5.

Ejercicio 3.5

Por muchas y diferentes razones, querríamos diseñar un multiplicador más rápido. Se han propuesto muchas alternativas para conseguirlo. En la siguiente tabla, A representa el número de bits de un entero y B representa las unidades de tiempo (ut) necesarias para llevar a cabo un paso de la operación.

	(nº de bits)	(unidades de tiempo)
a.	4	3 ut
b.	32	7 ut

3.5.1 [10]<3.3> Calcule el tiempo necesario para realizar una multiplicación usando las alternativas mostradas en las figuras 3.4 y 3.5 si un entero tiene A bits y cada paso de la operación requiere B unidades de tiempo. Suponga que en el paso 1a siempre se realiza una suma, tanto si se suma el multiplicando o si se suma un 0. Suponga también que los registros ya tienen un valor inicial (se necesita contar cuánto tardará el lazo de la multiplicación). En caso de implementarse en hardware, considere que los desplazamientos del multiplicando y el multiplicador se hacen simultáneamente. En caso de implementarse en software, tendrán que hacerse uno detrás del otro. Obtenga el tiempo de ambas implementaciones.

3.5.2 [10]<3.3> Calcule el tiempo necesario para realizar una multiplicación usando la alternativa descrita en el texto (31 sumadores apilados verticalmente) si un entero tiene A bits y cada sumador requiere B unidades de tiempo.

3.5.3 [20]<3.3> Calcule el tiempo necesario para realizar una multiplicación usando la alternativa descrita en la figura 3.8, si un entero tiene A bits y cada sumador requiere B unidades de tiempo.

Ejercicio 3.6

En este ejercicio se abordan otras formas de mejorar las prestaciones de la multiplicación, basadas en hacer más desplazamientos y menos operaciones aritméticas. La siguiente tabla muestra parejas de números hexadecimales.

	A	B
a.	24	c9
b.	41	18

3.6.1 [20]<3.3> Tal como se ha discutido en este capítulo, puede obtenerse una mejora de las prestaciones haciendo desplazamientos y sumas en lugar de una multiplicación real. Por ejemplo, 9 × 6 puede reescribirse como (2 × 2 × 2 + 1) × 6, y se puede calcular el producto desplazando 6 tres veces y sumando 6. Muestre la mejor forma de calcular A × B con desplazamientos y sumas/restas, suponiendo que A y B son números enteros de 8 bit sin signo.

3.6.2 [20]<3.3> Muestre la mejor forma de calcular A × B con desplazamientos y sumas, suponiendo que A y B son números enteros de 8 bit con signo en formato signo-magnitud.

3.6.3 [60]<3.3> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de dos números enteros con signo, utilizando desplazamientos y sumas, tal como se describe en 3.6.1.

La siguiente tabla muestra otras parejas de números hexadecimales.

	A	B
a.	42	36
b.	9F	8E

3.6.4 [30]<3.3> Otra alternativa para reducir el número de operaciones aritméticas de la multiplicación es el algoritmo de Booth. Este algoritmo se ha utilizado durante años y los detalles sobre su funcionamiento están disponibles en la web. Básicamente, supone que un desplazamiento es más rápido que una suma o una resta y, basándose en esta suposición, reduce el número de operaciones aritméticas necesarias para realizar una multiplicación. Identifique secuencias de unos y ceros y los sustituye por desplazamientos. Encuentre una descripción del algoritmo y explique su funcionamiento.

3.6.5 [30]<3.3> Muestre el cálculo paso a paso de la multiplicación de dos números A y B hexadecimales enteros de 8 bits en complemento a 2.

3.6.6 [60]<3.3> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de A y B con el algoritmo de Booth.

Ejercicio 3.7

En este ejercicio se profundiza en la división. Utilice los número octales de la siguiente tabla.

	A	B
a.	50	23
b.	25	44

3.7.1 [20]<3.4> Usando una tabla similar a la mostrada en la figura 3.11, calcule A/B usando el hardware de la figura 3.9. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros sin signo de 6 bits.

3.7.2 [30]<3.4> Usando una tabla similar a la mostrada en la figura 3.1, calcule A/B usando el hardware de la figura 3.12. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros sin signo de 6 bits. Este algoritmo es ligeramente diferente del mostrado en la figura 3.10. Se tendrá que pensar sobre esto, hacer uno o dos ejemplos o incluso consultar en la web para averiguar como conseguir que la operación se haga correctamente. (Pista: Una posible solución se puede obtener teniendo en cuenta que el registro del resto de la figura 3.12 puede desplazarse en ambas direcciones.)

3.7.3 [60]<3.4> Escriba un programa en lenguaje ensamblador de MIPS para calcular A/B con el algoritmo de la figura 3.9 y suponiendo que A y B son enteros sin signo de 6 bits.

La siguiente tabla muestra otras parejas de número octales.

	A	B
a.	55	24
b.	36	51

3.7.4 [30]<3.4> Usando una tabla similar a la mostrada en la figura 3.11, calcule A/B usando el hardware de la figura 3.9. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros con signo de 6 bits en formato signo-magnitud. Se debe incluir como se calculan los signos del cociente y el resto.

3.7.5 [30]<3.4> Usando una tabla similar a la mostrada en la figura 3.11, calcule A/B usando el hardware de la figura 3.12. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros con signo de 6 bits en formato signo-magnitud. Se debe incluir como se calculan los signos del cociente y el resto.

3.7.6 [60]<3.4> Escriba un programa en lenguaje ensamblador de MIPS para calcular A/B con el algoritmo de la figura 3.12 y suponiendo que A y B son enteros con signo.

Ejercicio 3.8

El algoritmo de división con restauración se describe en la figura 3.10, llamado así porque si el resultado de restar el resto menos el divisor es negativo, el divisor se vuelve a sumar al resto, restaurando su valor. Sin embargo, se han desarrollado otros algoritmos que eliminan esta suma extra. Estos algoritmos pueden consultarse en la web. Se analizarán estos algoritmos utilizando las parejas de números octales de la tabla.

	A	B
a.	75	12
b.	52	37

3.8.1 [30]<3.4> Usando una tabla similar a la mostrada en la figura 3.11, calcule A/B con un algoritmo de división sin restauración. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros sin signo de 6 bits.

3.8.2 [60]<3.4> Escriba un programa en lenguaje ensamblador de MIPS para calcular A/B con un algoritmo de división sin restauración y suponiendo que A y B son enteros de 6 bits con signo en formato de complemento a 2.

3.8.3 [60]<3.4> Compare las prestaciones de la división con y sin restauración; muestre el número de pasos necesarios para calcular A/B con cada método. Suponga que A y B son enteros con signo de 6 bits en formato signo-magnitud. Es suficiente con escribir un programa para división con restauración y sin restauración.

La siguiente tabla muestra otras parejas de número octales.

	A	B
a.	17	14
b.	70	23

3.8.4 [30]<3.4> Usando una tabla similar a la mostrada en la figura 3.11, calcule A/B usando división sin representación. Muestre los contenidos de los registros en cada paso. Suponga que A y B son enteros sin signo de 6 bits.

3.8.5 [60]<3.4> Escriba un programa en lenguaje ensamblador de MIPS para calcular A/B con división sin representación y suponiendo que A y B son enteros de 6 bits con signo en formato de complemento a 2.

3.8.6 [60]<3.4> Compare las prestaciones de la división sin restauración y sin representación; muestre el número de pasos necesarios para calcular A/B con cada método. Suponga que A y B son enteros con signo de 6 bits en formato signo-magnitud. Es suficiente con escribir un programa para división con restauración y sin restauración.

Ejercicio 3.9

La división es una operación tan lenta y compleja que el manual Fortran Optimizations de CRAY T3E dice: "La mejor estrategia para la división es evitarla cuando sea posible". En este ejercicio se verán diferentes estrategias para realizar la división.

a.	División con restauración
b.	División SRT

3.9.1 [30]<3.4> Describa el algoritmo de la tabla en detalle.

3.9.2 [60]<3.4> Explique cómo funciona el algoritmo de la tabla mediante un diagrama de flujo (o un fragmento de código de alto nivel).

3.9.3 [60]<3.4> Escriba un programa en lenguaje ensamblador de MIPS para calcular A/B con el algoritmo de la tabla.

Ejercicio 3.10

En una arquitectura Von Neumann, los grupos de bits no tienen ningún significado en sí mismos; lo que representa un grupo de bits depende completamente de como se usen. La siguiente tabla muestra grupos de bits en notación hexadecimal.

a.	0x24A60004
b.	0xAFBF0000

3.10.1 [5]<3.5> ¿Qué número decimal representa si es un entero en complemento a 2? ¿Y un entero sin signo?

3.10.2 [10]<3.5> Si este grupo de bits estuviese en el registro de instrucciones ¿qué instrucción MIPS se ejecutaría?

3.10.3 [10]<3.5> ¿Qué número decimal representa si es un número en punto flotante? Utilice el estándar IEEE 754.

La siguiente tabla muestra números decimales.

a.	-1609.5
b.	-938.8125

3.10.4 [10]<3.5> Escriba la representación binaria del número de la tabla en formato IEEE 754 de precisión simple.

3.10.5 [10]<3.5> Escriba la representación binaria del número de la tabla en formato IEEE 754 de precisión doble.

3.10.6 [10]<3.5> Escriba la representación binaria del número de la tabla en formato IBM (base 16, en vez de base 2, con 7 bits para el exponente).

Ejercicio 3.11

En el estándar de punto flotante IEEE 754 el exponente se almacena con un sesgo (también llamado "Exceso-N"). El motivo de esta desviación es que se quiere que una secuencia con todos los bits igual a cero esté lo más próxima a cero que sea posible. Dado que estamos usando un bit oculto (*hidden bit*) igual a 1, si el exponente se representase en complemento a 2 la secuencia con todos los bits iguales a cero ¡estaría representando el número 1! (Cualquier número elevado a 0 es 1, entonces $1.0^0 = 1$.) Hay otros muchos aspectos del estándar IEEE 754 que han sido establecidos para conseguir que el hardware de las unidades de punto flotante sean más rápidas. Sin embargo, en muchos computadores antiguos los cálculos en punto flotante se realizan por software y con otros formatos. La siguiente tabla muestra número decimales.

a.	5.00736125×10^{5}
b.	$-2.691650390625 \times 10^{-2}$

3.11.1 [20]<3.5> Escriba la representación binaria del número de la tabla en el formato empleado en el DEC PDP-8 (los 12 bits más a la izquierda representan el exponente en complemento a 2 y los 24 bits más a la derecha representan la mantisa en complemento a 2). No se usa el bit oculto. Compare el rango y la exactitud de esta representación de 36 bits con los estándares IEEE 754 de precisión simple y doble.

3.11.2 [20]<3.5> NVIDIA utiliza un formato "mitad", similar al IEEE 754 pero con solo 16 bits. El bit de la izquierda es el signo, se reservan 5 bits para el exponente en exceso 16 y 10 bits para la mantisa. Se asume un bit oculto igual a 1.

Represente el número de la tabla en este formato. Compare el rango y la exactitud de esta representación de 16 bits con el estándar IEEE 754 de precisión simple.

3.11.3 [20]<3.5> Los Hewlett-Packard 2114, 2115 y 2116 usaban un formato con la mantisa almacenada en los 16 bits más a la izquierda en representación de complemento a 2, seguido de otro campo de 16 bits que reservaba los 8 de la izquierda para una extensión de la mantisa (haciendo que la mantisa tuviese 24 bits) y los otros 8 bits representaban el exponente. El exponente se almacenaba en formato de signo-magnitud ¡con el bit de signo a la derecha! No se utilizaba bit oculto igual a 1. Represente el número de la tabla en este formato. Compare el rango y la exactitud de esta representación de 32 bits con el estándar IEEE 754 de precisión simple.

La siguiente tabla muestra números decimales.

	A	B
a.	-1278×10^{3}	-3.90625×10^{-1}
b.	2.3109375×10^{1}	$6.391601562 \times 10^{-1}$

3.11.4 [20]<3.5> Calcule la suma de A y B suponiendo que A y B están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos.

3.11.5 [60]<3.5> Escriba un programa en lenguaje ensamblador de MIPS para calcular A + B suponiendo que A y B están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky.

3.11.6 [60]<3.5> Escriba un programa en lenguaje ensamblador de MIPS para calcular A + B suponiendo que A y B están representados en el formato descrito en el ejercicio 3.11.1.. Modifique el programa para calcular la suma de dos números en el formato del ejercicio 3.11.3. ¿Qué formato es más sencillo para programar? Comparar con el formato IEEE 754 (en este ejercicio no hay que preocuparse del bit sticky).

Ejercicio 3.12

La multiplicación en punto flotante es más complicada y exigente que la suma, y ambas son insignificantes en comparación con la división punto flotante.

	A	B
a.	5.66015625×10^{0}	8.59375×10^{0}
b.	6.18×10^{2}	5.796875×10^{1}

3.12.1 [30]<3.5> Calcule el producto de A y B a mano, suponiendo que A y B están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos; sin embargo, tal como se ha hecho en el ejemplo del capítulo, se puede hacer la multiplicación en un formato legible para los humanos, en lugar de con las técnicas de los ejercicios 3.4 a 3.6. Indique si se produce desbordamiento o desbordamiento a cero. Muestre el resultado en el formato de NVIDIA y en decimal. ¿Cuál es la exactitud del resultado? Compare con el resultado que se obtiene al hacer la multiplicación en una calculadora.

3.12.2 [60]<3.5> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de A y B con A y B representados en el formato IEEE 754. Indicar si se produce desbordamiento o desbordamiento a cero. Recuérdese que IEEE 754 asume un bit de guarda, un bit de redondeo, un bit de sticky y redondeo al más próximo par.

3.12.3 [60]<3.5> Escriba un programa en lenguaje ensamblador de MIPS para calcular el producto de A y B con A y B representados en el formato del ejercicio 3.11.1. Modifique el programa para calcular la suma de dos números en el formato del ejercicio 3.11.3. ¿Qué formato es más sencillo para programar? Compare con el formato IEEE 754 (en este ejercicio no hay que preocuparse del bit sticky).

La siguiente tabla muestra otros números decimales.

	A	B
a.	3.264×10^3	6.52×10^2
b.	-2.27734375×10^0	1.154375×10^2

3.12.4 [30]<3.5> Calcule A/B, mostrando todos los pasos necesarios para obtener el resultado. Suponga un bit de guarda, un bit de redondeo y un bit de sticky, y utilícelos si es necesario. Escriba el resultado en formato punto flotante de 16 y en decimal y compare con el resultado que se obtiene al hacer la operación en una calculadora.

Los Livermore Loops son un conjunto de núcleos de cálculo intensivo en punto flotante extraídos de programas científicos del Laboratorio Lawrence Livermore. La siguiente tabla identifica algunos núcleos del conjunto.

Se pueden obtener en http://www.netlib.org/benchmark/livermore.

a.	Livermore Loop 1
b.	Livermore Loop 2

3.12.5 [60]<3.5> Escriba el núcleo en lenguaje ensamblador del MIPS.

3.12.6 [60]<3.5> Describa con detalle una técnica para la realización de la división en un computador. Incluya referencias de las fuentes que haya consultado.

Ejercicio 3.13

Las operaciones con enteros en punto fijo se comportan como se espera: se cumplen las propiedades asociativa, conmutativa y distributiva. Esto no siempre es cierto para operaciones con número punto flotante. Nos centramos la propiedad asociativa. La siguiente tabla muestra un conjunto de números decimales.

	A	B	C
a.	-1.6360×10^4	1.6360×10^4	1.0×10^0
b.	2.865625×10^1	4.140625×10^{-1}	1.2140625×10^1

3.13.1 [20]<3.2, 3.5, 3.6> Calcule (A + B) + C a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Realice redondeo al más próximo par, para lo que se necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.13.2 [20]<3.2, 3.5, 3.6> Calcule A + (B + C) a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.13.3 [10]<3.2, 3.5, 3.6> Basándose en las respuestas de los ejercicios 3.13.1 y 3.13.2, ¿(A + B) + C = A + (B + C)?

La siguiente tabla muestra otro conjunto de números decimales.

	A	B	C
a.	4.8828125×10^{-4}	1.768×10^3	2.50125×10^2
b.	4.721875×10^1	2.809375×10^1	3.575×10^1

3.13.4 [30]<3.2, 3.5, 3.6> Calcule (A × B) × C a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.13.5 [30]<3.2, 3.5, 3.6> Calcule A × (B × C) a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.13.6 [10]<3.2, 3.5, 3.6> Basándose en las respuestas de los ejercicios 3.13.5 y 3.13.4, ¿(A × B) × C = A × (B × C)?

Ejercicio 3.14

La propiedad asociativa no es la única que no siempre se mantiene cuando se trabaja con números en punto flotante. Hay otras singularidades que también ocurren. La siguiente tabla muestra un conjunto de números decimales.

	A	B	C
a.	1.5234375×10^{-1}	2.0703125×10^{-1}	9.96875×10^{1}
b.	-2.7890625×10^{1}	-8.088×10^{3}	1.0216×10^{4}

3.14.1 [30]<3.2, 3.3, 3.5, 3.6> Calcule A × (B + C) a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.14.2 [30]<3.2, 3.3, 3.5, 3.6> Calcule (A × B) + (A × C) a mano, suponiendo que A, B y C están representados en el formato de 16 bits de NVIDIA descrito en el ejercicio 3.11.2. Redondee al más próximo par, para lo que necesitará un bit de guarda, un bit de redondeo y un bit de sticky. Muestre todos los pasos y escriba el resultado en el formato punto flotante de 16 bits y en decimal.

3.14.3 [10]<3.2, 3.5, 3.6> Basándose en las respuestas de los ejercicios 3.14.1 y 3.14.2, ¿A × (B + C)=(A × B) + (A × C)?

La siguiente tabla muestra parejas de operandos, formadas por una fracción y un entero.

	A	B
a.	1/3	3
b.	-1/7	7

3.14.4 [10]<3.5> Escriba la representación binaria de A en formato punto flotante IEE 754. ¿Se puede representar A de forma exacta?

3.14.5 [10]<3.2, 3.3, 3.5, 3.6> ¿Qué se obtiene si se suma B veces el operando A? ¿Cuál es el resultado de A × B? ¿Son iguales? ¿Cuáles deberían haber sido estos resultados?

3.14.6 [60]<3.2, 3.3, 3.4, 3.5, 3.6> ¿Qué se obtiene si se calcula la raíz cuadrada de B y este valor se multiplica por sí mismo? ¿Qué se debería haber obtenido? Haga este ejercicio para número punto flotante con precisión simple y doble. (Escriba un programa para hacer estas operaciones.)

Ejercicio 3.15

En la mantisa se usa una representación binaria, pero podría usarse otra representación. Por ejemplo, IBM utiliza números en base 16 en algunos de sus formatos de punto flotante. Hay otras alternativas que también son posibles, cada una con sus ventajas e inconvenientes. En la tabla siguiente se muestran fracciones que serán representadas en varios formato de punto flotante.

a.	1/2
b.	1/9

3.15.1 [10]<3.5, 3.6> Escriba la representación binaria de la mantisa para una representación punto flotante que utiliza números binarios para la mantisa (esencialmente, esto es lo que se ha estado haciendo en este capítulo). Suponga que la mantisa es de 24 bits y no se necesita normalizar. ¿Esta representación es exacta?

3.15.2 [10]<3.5, 3.6> Escriba la representación binaria de la mantisa para una representación punto flotante que utiliza números BCD (Binary Coded Decimal, base 10) para la mantisa en lugar de base 2. Suponga que la mantisa es de 24 bits y no se necesita normalizar. ¿Esta representación es exacta?

3.15.3 [10]<3.5, 3.6> Escriba la representación binaria de la mantisa para una representación punto flotante que utiliza números en base 15 para la mantisa en lugar de base 2. (Los números en base 16 utilizan los símbolos 0-9 y A-F; los números en base 15 usarían 0-9, A-E.) Suponga que la mantisa es de 24 bits y no se necesita normalizar. ¿Esta representación es exacta?

3.15.4 [10]<3.5, 3.6> Escriba la representación binaria de la mantisa para una representación punto flotante que utiliza números en base 30 para la mantisa en lugar de base 2. (Los números en base 16 utilizan los símbolos 0-9 y A-F; los números en base 15 usarían 0-9, A-T). Supóngase que la mantisa es de 24 bits y no se necesita normalizar. ¿Esta representación es exacta? ¿Esta representación tiene alguna ventaja?

Respuestas a las autoevaluaciones

$3.2, página 229: 3.

$3.4, página 269: 3.

4

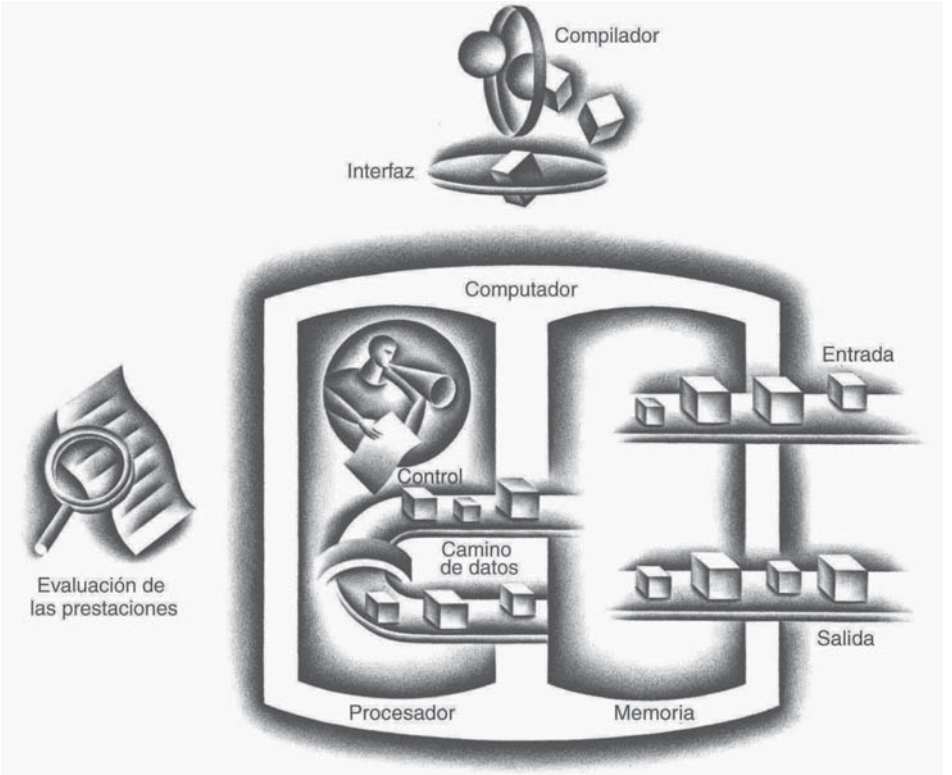
El procesador

En una materia fundamental, ningún detalle es insignificante.

Proverbio francés

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco componentes clásicos del computador



4.1 Introducción

En el capítulo 1 vimos que las prestaciones de una máquina están determinadas por tres factores clave: el número de instrucciones, el tiempo del ciclo de reloj y los ciclos de reloj por instrucción (*cycles per instruction*, CPI). El compilador y la arquitectura del repertorio de instrucciones, que hemos examinado en el capítulo 2, determinan el número de instrucciones requerido por un cierto programa. Sin embargo, tanto el tiempo de ciclo del reloj como el número de ciclos por instrucción vienen dados por la implementación del procesador. En este capítulo se construye el camino de datos y la unidad de control para dos realizaciones diferentes del repertorio de instrucciones MIPS.

Este capítulo incluye una explicación de los principios y técnicas utilizadas en la implementación de un procesador, comenzando con un resumen altamente abstracto y simplificado en esta sección, seguido por una sección que construye un camino de datos y una versión simple de un procesador suficiente para realizar repertorios de instrucciones como MIPS. El núcleo del capítulo describe una implementación segmentada más realista del MIPS, seguido de una sección que desarrolla los conceptos necesarios para la implementación de un repertorio de instrucciones más complejo, como el x86.

Para el lector interesado en comprender la interpretación a alto nivel de las instrucciones y su impacto en las prestaciones del programa, esta sección inicial y la sección 4.5 proporcionan los conceptos básicos de la segmentación. Las tendencias actuales se indican en la sección 4.10, y la sección 4.11 describe el microprocesador AMD Opteron X4 (Barcelona). Estas secciones proporcionan suficiente bagaje para comprender los conceptos de segmentación a alto nivel.

Las secciones 4.3, 4.4 y 4.6 son útiles para los lectores interesados en entender el procesador y sus prestaciones con mayor profundidad y las secciones 4.2, 4.7, 4.8 y 4.9 para los interesados en como construir un procesador. Para los lectores interesados en diseño hardware moderno, la **sección 4.12** en el CD describe como se utilizan los lenguajes de descripción hardware y las herramientas de CAD para la implementación de hardware y como utilizar un lenguaje de descripción hardware para describir una implementación segmentada. También se incluyen más figuras sobre la ejecución en un hardware segmentado.

Una implementación básica MIPS

Examinaremos una implementación que incluirá un subconjunto básico del repertorio de instrucciones del MIPS formado por:

- Las instrucciones de referencia a memoria cargar palabra (*load word* `lw`) y almacenar palabra (*store word* `sw`).
- Las instrucciones aritmético-lógicas `add`, `sub`, `and`, `or` y `slt`.
- Las instrucciones de saltar si igual (*branch on equal* `beq`) y salto incondicional (*jump* `j`), que se añadirán en último lugar.

Este subconjunto no incluye todas las instrucciones con enteros (por ejemplo, se han omitido las de desplazamiento, multiplicación y división), ni ninguna de las instrucciones de punto flotante. Sin embargo, servirá para ilustrar los principios básicos que se utilizan en la construcción del camino de datos y el diseño de la unidad de control. La implementación de las instrucciones restantes es similar.

Al examinar la implementación, tendremos la oportunidad de ver cómo la arquitectura del repertorio de instrucciones determina muchos aspectos de la implementación y cómo la elección de varias estrategias de implementación afecta a la frecuencia del reloj y al CPI de la máquina. Muchos de los principios básicos de diseño introducidos en el capítulo 1, como las directrices «hacer rápido el caso común» y «la simplicidad favorece la regularidad», pueden observarse en la implementación. Además, la mayoría de los conceptos utilizados para realizar el subconjunto MIPS en este capítulo y en el siguiente son las ideas básicas que se utilizan para construir un amplio espectro de computadores, desde servidores de altas prestaciones hasta microprocesadores de propósito general o procesadores empotrados.

Una visión general de la implementación

En el capítulo 2 analizamos el núcleo básico de instrucciones MIPS, incluidas las instrucciones aritmético-lógicas sobre enteros, las instrucciones de referencia a memoria y las instrucciones de salto. La mayor parte de lo necesario para implementar estas instrucciones es común para todas ellas, independientemente de su tipo concreto. Para cada instrucción, los dos primeros pasos son idénticos:

1. Enviar el contador de programa (PC) a la memoria que contiene el código y cargar la instrucción desde esa memoria.
2. Leer uno o dos registros, utilizando para ello los campos específicos de la instrucción para seleccionar los registros a leer. Para la instrucción de cargar palabra es necesario un solo registro, pero la mayoría del resto de las instrucciones requiere la lectura de dos registros.

Después de estos dos pasos, las acciones necesarias para completar la instrucción dependen del tipo de la misma. Afortunadamente, para los tres tipos de instrucciones (referencia a memoria, aritmético-lógicas y saltos) las acciones son generalmente las mismas, independientemente del código de operación exacto. La sencillez y regularidad del repertorio de instrucciones MIPS simplifica la implementación porque la ejecución de muchas clases de instrucciones es similar.

Por ejemplo, todos los tipos de instrucciones, excepto las de salto incondicional, utilizan la unidad aritmético-lógica (ALU) después de la lectura de los registros. Las instrucciones de referencia a memoria utilizan la ALU para el cálculo de la dirección, las instrucciones aritmético-lógicas para ejecutar la operación, y las de salto condicional para comparar. Después de utilizar la ALU, las acciones requeridas para completar la ejecución de los diferentes tipos de instrucciones son distintas. Una instrucción de referencia a memoria necesitará acceder a memoria, ya sea para escribir el dato en una operación de almacenamiento o para leer, en caso de una carga. Una instrucción aritmético-lógica o una de carga debe escribir el resultado calculado por la ALU o el leido de la memoria en un registro. Finalmente, en una instrucción de salto condicional es necesario modificar la dirección de la siguiente

instrucción según el resultado de la comparación; en caso contrario el PC debe incrementarse en 4 para obtener la dirección de la siguiente instrucción.

La figura 4.1 muestra un esquema de alto nivel de una implementación MIPS, en el que se muestran las diferentes unidades funcionales y su interconexión. A pesar de que esta figura muestra la mayor parte del flujo de datos en el procesador, omite dos aspectos importantes de la ejecución de las instrucciones.

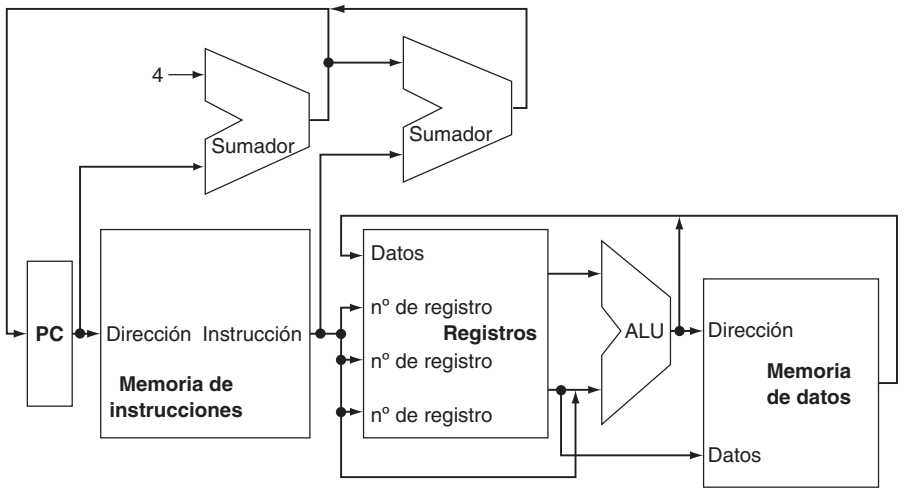


FIGURA 4.1 Una visión abstracta de la implementación del subconjunto MIPS en la que se muestra la mayor parte de las unidades funcionales y las conexiones entre ellas. Todas las instrucciones comienzan utilizando el contador de programa (PC) para proporcionar la dirección de la instrucción en la memoria de instrucciones. Tras la captación de la instrucción, ciertos campos de ésta especifican los registros que se utilizan como operandos fuente. Una vez que éstos han sido leídos, puede operarse con ellos, ya sea para calcular una dirección de memoria (en una carga o un almacenamiento), para calcular un resultado aritmético (para una instrucción aritmético-lógica entera), o bien para realizar una comparación (en un salto condicional). Si la instrucción es una instrucción aritmético-lógica, el resultado de la ALU debe almacenarse en un registro. Si la operación es una carga o un almacenamiento, el resultado de la ALU se utiliza como la dirección donde almacenar un valor en memoria o cargar un valor en los registros. El resultado de la ALU o memoria se escribe en el banco de registros. Los saltos condicionales requieren el uso de la salida de la ALU para determinar la dirección de la siguiente instrucción, que proviene de la ALU (donde se suma el PC y el desplazamiento) o desde un sumador que incrementa el valor actual del PC en 4. Las líneas gruesas que interconectan las unidades funcionales representan los buses, que consisten en múltiples señales. Las flechas se utilizan para indicar al lector cómo fluye la información. Ya que las líneas de señal pueden cruzarse, se muestra explícitamente la conexión con la presencia de un punto donde se cruzan las líneas.

En primer lugar, en varios puntos, la figura muestra datos dirigidos a una unidad particular provenientes de dos orígenes diferentes. Por ejemplo, el valor escrito en el PC puede provenir de cualquiera de los dos sumadores, el dato escrito en el banco de registros puede provenir de la ALU o de la memoria de datos, y la segunda entrada de la ALU proviene de un registro o del campo inmediato de la instrucción. En la práctica, estas líneas de datos no pueden conectarse directamente; debe añadirse un elemento que seleccione entre los múltiples orígenes y dirija una de estas fuentes al destino. Esta selección se realiza comúnmente mediante un dispositivo denominado *multiplexor*, aunque sería más adecuado

denominarlo *selector de datos*. El multiplexor, que se describe en detalle en el apéndice C, selecciona una de entre varias entradas según la configuración de sus líneas de control. Las líneas de control se configuran principalmente a partir de información tomada de la instrucción en ejecución.

En segundo lugar, varias de las unidades deben controlarse dependiendo del tipo de instrucción. Por ejemplo, la memoria de datos debe leer en una carga y escribir en un almacenamiento. Debe escribirse en el banco de registros en una instrucción de carga y en una instrucción aritmético-lógica. Y, por supuesto, la ALU debe realizar una entre varias operaciones, tal como se mostró en el capítulo 2. (El apéndice C describe el diseño lógico detallado de la ALU.) Al igual que los multiplexores, estas operaciones son dirigidas por las líneas de control que se establecen según los diversos campos de la instrucción.

La figura 4.2 muestra el camino de datos de la figura 4.1 con los tres multiplexores necesarios añadidos, así como las líneas de control para las principales unidades funcionales. Para determinar la activación de las líneas de control de las unidades funcionales y de los multiplexores se utiliza una unidad de control que toma la instrucción como entrada. El tercer multiplexor, que determina si PC + 4 o la dirección destino del salto se escribe en el PC, se activa según el valor de la salida Cero de la ALU, que se utiliza para realizar la comparación de la instrucción `beq`. La regularidad y simplicidad del repertorio de instrucciones MIPS implica que un simple proceso de descodificación puede ser utilizado para determinar cómo activar las líneas de control.

En el resto de este capítulo, se refina este esquema para añadir los detalles, lo que va a requerir que se incluyan unidades funcionales adicionales, incrementar el número de conexiones entre las unidades y, por supuesto, añadir una unidad de control que determine las acciones que deben realizarse para cada uno de los distintos tipos de instrucciones. Las secciones 4.3 y 4.4 describen una realización simple que utiliza un único ciclo de reloj largo para cada instrucción y sigue la forma general de las figuras 4.1 y 4.2. En este primer diseño, cada instrucción inicia su ejecución en un flanco de reloj y completa la ejecución en el siguiente flanco de reloj.

Aunque es más fácil de comprender, este enfoque no es práctico, puesto que hay que alargar el ciclo de la señal de reloj para permitir la ejecución de la instrucción más lenta. Una vez diseñado el control para este computador sencillo, se analizará la implementación segmentada en toda su complejidad, incluyendo el tratamiento de las excepciones.

Autoevaluación

¿Cuántos de los cinco componentes clásicos de un computador, mostrados en la página 299, se incluyen en la figuras 4.1 y 4.2?

4.2 Convenios de diseño lógico

Para tratar el diseño de la máquina, se debe decidir cómo operará su implementación lógica y cómo será su sincronización. Esta sección repasa unas cuantas ideas clave de diseño lógico que se utilizarán ampliamente en este capítulo. Si se tiene

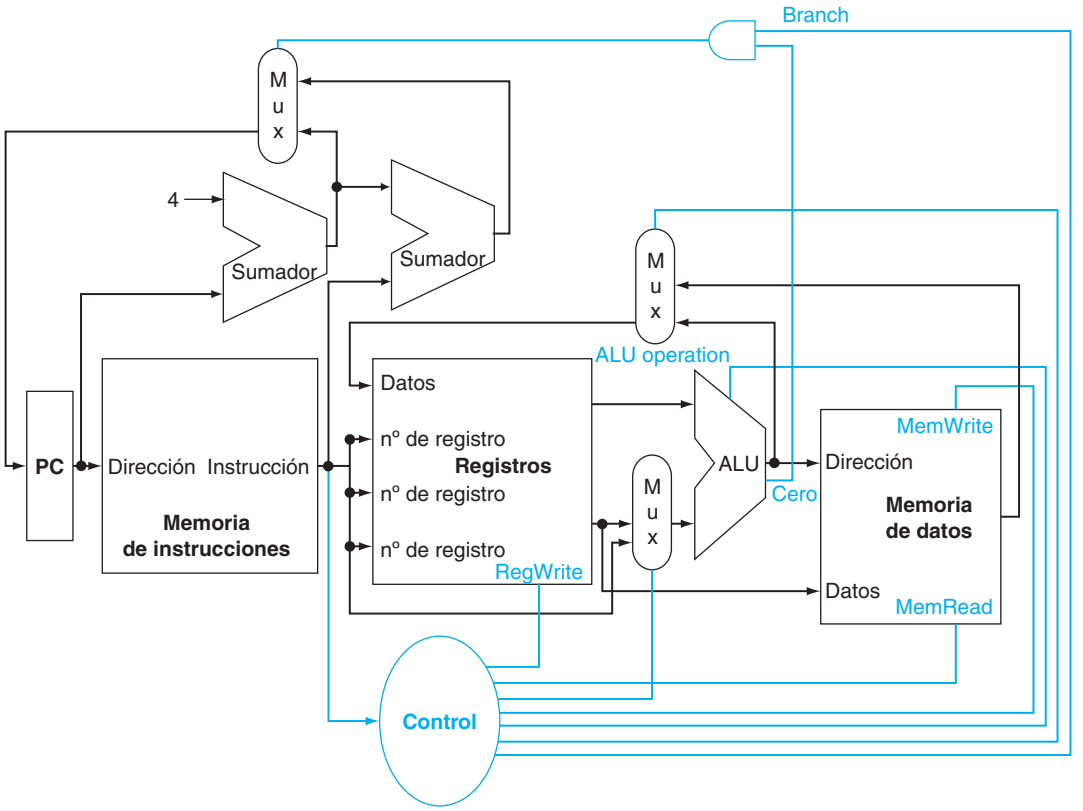


FIGURA 4.2 Implementación básica del subconjunto del MIPS incluyendo los multiplexores y líneas de control necesarias. El multiplexor de la parte superior controla el valor que se va a cargar en el PC (PC + 4 o la dirección destino del salto condicional); el multiplexor es controlado por una puerta que realiza una función «and» entre la salida Cero de la ALU y una señal de control que indica que la instrucción es de salto condicional. El multiplexor cuya salida está conectada al banco de registros se utiliza para seleccionar la salida de la ALU (en el caso de una instrucción aritmético-lógica) o la salida de la memoria de datos (en el caso de una carga desde memoria) para escribir en el banco de registros. Por último, el multiplexor de la parte inferior determina si la segunda entrada de la ALU procede de los registros (en una instrucción aritmético-lógica no inmediata) o del campo de desplazamiento de la instrucción (en una operación inmediata, una carga o almacenamiento, o un salto condicional). Las líneas de control añadidas son directas y determinan la operación realizada por la ALU, si se debe leer o escribir en la memoria de datos, y si los registros deben realizar una operación de escritura. Las líneas de control se muestran en color para facilitar su identificación.

escaso o ningún conocimiento sobre diseño lógico, el apéndice C resultará muy útil antes de abordar esta sección.

Las unidades funcionales de la implementación MIPS constan de dos tipos de elementos lógicos diferentes: elementos que operan con datos y elementos que contienen el estado. Los elementos que operan con datos son todos **combinacionales**, lo que significa que sus salidas dependen únicamente de los valores actuales de las entradas. Para una misma entrada, un elemento combinacional siempre produce la misma salida. La ALU mostrada en la figura 4.1 y analizada en detalle en el apéndice C es un elemento combinacional. Para un conjunto de entradas, siempre produce la misma salida porque no tiene almacenamiento interno.

Elemento combinacional: un elemento operacional tal como una puerta AND o una ALU.

Otros elementos en el diseño no son combinacionales, sino que contienen *estado*. Un elemento contiene estado si tiene almacenamiento interno. Estos elementos se denominan **elementos de estado** porque, si se apaga la máquina, se puede reiniciar cargando dichos elementos con los valores que contenían antes de apagarla. Además, si se guardan y se restauran, es como si la máquina no se hubiera apagado nunca. Así, los elementos de estado caracterizan completamente la máquina. En la figura 4.1, las memorias de instrucciones y datos, así como los registros, son ejemplos de elementos de estado.

Elemento de estado: elemento de memoria.

Un elemento de estado tiene al menos dos entradas y una salida. Las entradas necesarias son el valor del dato a almacenar en el elemento de estado y el reloj, que determina cuándo se almacena el dato. La salida del elemento de estado proporciona el valor que se almacenó en un ciclo de reloj anterior. Por ejemplo, uno de los elementos de estado más simple es un biestable de tipo D (véase el **apéndice C**), el cual tiene exactamente estas dos entradas (un dato y un reloj) y una salida. Además de los biestables, la implementación MIPS también utiliza otros dos tipos de elementos de estado: memorias y registros, que aparecen en la figura 4.1. El reloj se utiliza para determinar cuándo debería escribirse el elemento de estado. Un elemento de estado se puede leer en cualquier momento.

Los componentes lógicos que contienen estado también se denominan *secuenciales* porque sus salidas dependen tanto de sus entradas como del contenido de su estado interno. Por ejemplo, la salida de la unidad funcional que representa a los registros depende del identificador del registro y de lo que se haya almacenado en los registros anteriormente. La operación tanto de los elementos combinacionales como secuenciales, así como su construcción, se analiza con más detalle en el **apéndice C**.

Se utiliza la palabra **activado** para indicar que una señal se encuentra lógicamente a alta y *activar* para especificar que una señal debe ser puesta a alta, mientras que *desactivar* o **desactivado** representan un valor lógico a baja.

Activado: la señal está al nivel lógico alto o verdadero.
Desactivado: la señal está al nivel lógico bajo o falso.

Metodología de sincronización

Una **metodología de sincronización** define cuándo pueden leerse y escribirse las diferentes señales. Es importante especificar la temporización de las lecturas y las escrituras porque, si una señal se escribe en el mismo instante en que es leída, el valor leído puede corresponder al valor antiguo, al valor nuevo o ¡incluso a una combinación de ambos! No es necesario decir que los diseños de computadores no pueden tolerar tal imprevisibilidad. La metodología de sincronización se diseña para prevenir esta circunstancia.

Metodología de sincronización: aproximación que determina cuándo los datos son válidos y estables utilizando como referencia el reloj.

Por simplicidad, supondremos una metodología de **sincronización por flanco**. Esta metodología de sincronización implica que cualquier valor almacenado en un elemento lógico secuencial se actualiza únicamente en un flanco de reloj. Debido a que sólo los elementos de estado pueden almacenar datos, las entradas de cualquier lógica combinacional deben proceder de elementos de este tipo y las salidas también deben dirigirse hacia elementos de este tipo. Las entradas serán valores escritos en un ciclo anterior de reloj, mientras que las salidas son valores que pueden utilizarse en el ciclo siguiente.

Sincronización por flanco: esquema de sincronización en el cual todos los cambios de estado se producen en los flancos de reloj.

La figura 4.3 muestra dos elementos de estado que rodean a un bloque de lógica combinacional, que opera con un único ciclo de reloj. Todas las señales deben propagarse desde el elemento de estado 1, a través de la lógica combinacional hasta el elemento de estado 2 en un único ciclo de reloj. El tiempo necesario para que las señales alcancen el elemento de estado 2 define la duración del ciclo de reloj.

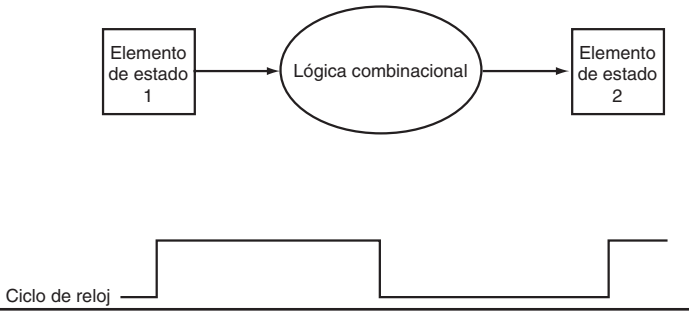


FIGURA 4.3 La lógica combinacional, los elementos de estado y el reloj están estrechamente relacionados. En un sistema digital síncrono, el reloj determina cuándo los elementos con estado van a escribir valores en el almacenamiento interno. Cualquiera de las entradas de un elemento de estado debe alcanzar un valor estable (es decir, alcanzar un valor que no va a cambiar hasta el siguiente flanco del reloj) antes de que el flanco de reloj activo cause la actualización del estado. Se supone que todos estos elementos, incluida la memoria, están sincronizados por flanco.

Señal de control: señal utilizada como selección en un multiplexor o para dirigir la operación de una unidad funcional; contrasta con una **señal de datos**, que continene información sobre la que opera una unidad funcional.

Por simplicidad, no se muestra una **señal de control** de escritura cuando se escribe en un elemento de estado en cada flanco activo de reloj. En cambio, si el elemento de estado no se actualiza en cada ciclo de reloj, es necesario incluir una señal de control de escritura (*write*). La señal de reloj y la señal de control de escritura son entradas, y el elemento de estado se actualiza únicamente cuando la señal de control de escritura se activa y ocurre un flanco de reloj.

La metodología por flanco permite leer el contenido de un registro, enviar el valor a través de alguna lógica combinacional, y escribir en ese mismo registro en un mismo ciclo de reloj, tal como se muestra en la figura 4.4. Es indiferente donde se asuma la implementación de las escrituras, ya sea en el flanco ascendente o en el descendente, ya que las entradas del bloque combinacional sólo pueden modifi-

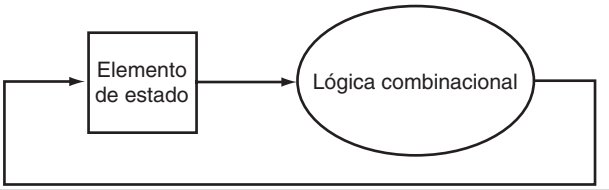


FIGURA 4.4 La metodología de sincronización por flanco permite leer y escribir en un elemento de estado en un mismo ciclo de reloj sin crear una condición de carrera que pueda conducir a valores indeterminados de los datos. Por supuesto, el ciclo de reloj debe ser suficientemente largo para que los valores de entrada sean estables cuando ocurra el flanco de reloj activo. El baipás no puede darse dentro de un mismo ciclo debido a la actualización por flanco del elemento de estado. Si fuera posible el baipás, puede que este diseño no funcione correctamente. Los diseños de este capítulo como los del próximo se basan en este tipo de metodología de sincronización y en estructuras como la mostrada en la figura.

carse en el flanco elegido. Con este tipo de metodología de sincronización, no existe baipás dentro de un mismo ciclo, y la lógica de la figura 4.4 funciona correctamente. En el apéndice C se analizan brevemente otras limitaciones de la sincronización —como los tiempos de preestabilización (*setup times*) y de mantenimiento (*hold times*)— así como otras metodologías.

En la arquitectura MIPS de 32 bits, casi todos estos elementos lógicos y de estado tendrán entradas y salidas de 32 bits, ya que ésta es la anchura de muchos de los datos tratados por el procesador. Se marcará de alguna forma cuando una unidad tenga una entrada o una salida con una anchura diferente. Las figuras mostrarán los buses (señales con anchura superior a un bit) mediante líneas más gruesas. Cuando se quiera combinar varios buses para formar uno de anchura superior, por ejemplo, si se quiere tener un bus de 32 bits combinando dos de 16, las etiquetas de dichos buses especificarán que hay varios buses agrupados. También se añaden como ayuda flechas para destacar la dirección del flujo de datos entre los elementos. Finalmente, el **color** indicará si se trata de una señal de control o de datos. Esta distinción se especificará mejor más adelante en este capítulo.

Autoevaluación

Verdadero o falso: ya que el banco de registros es leído y escrito en el mismo ciclo de reloj, cualquier camino de datos MIPS que utilice escrituras por flanco debe tener más de una copia del banco de registros.

Extensión: Existe una versión de 64 bits de la arquitectura MIPS y, naturalmente, la mayor parte de los componentes tienen un ancho de 64 bits. Por otra parte, se usan los términos activado y desactivado porque a veces el 1 representa el estado lógico alto y otras veces el estado lógico bajo.

4.3 Construcción de un camino de datos

Elemento del camino de datos: unidad funcional utilizada para operar o mantener un dato dentro de un procesador. En la implementación MIPS, los elementos del camino de datos incluyen las memorias de instrucciones y datos, el banco de registros, la unidad aritmético-lógica (ALU), y sumadores.

Contador de programa (PC): registro que contiene la dirección de la instrucción del programa que está siendo ejecutada.

Una forma razonable de empezar el diseño de un camino de datos es examinar los componentes principales necesarios para ejecutar cada tipo de instrucción del MIPS. Primero se consideran los **elementos del camino de datos** que necesita cada instrucción. Cuando se muestran los elementos del camino de datos, también se muestran sus señales de control.

La figura 4.5a muestra el primer elemento que se necesita: una unidad de memoria donde almacenar y suministrar las instrucciones a partir de una dirección. La figura 4.5b también muestra un registro, denominado **contador de programa (PC)**, que hemos visto en el capítulo 2 y se utiliza para almacenar la dirección de la instrucción actual. Finalmente se necesita un sumador encargado de incrementar el PC para que apunte a la dirección de la siguiente instrucción. Este sumador, que es combinacional, se puede construir a partir de la ALU descrita en detalle en el apéndice C, haciendo simplemente que las líneas de control siempre especifiquen una operación de suma. Este sumador se representa como una ALU con la eti-

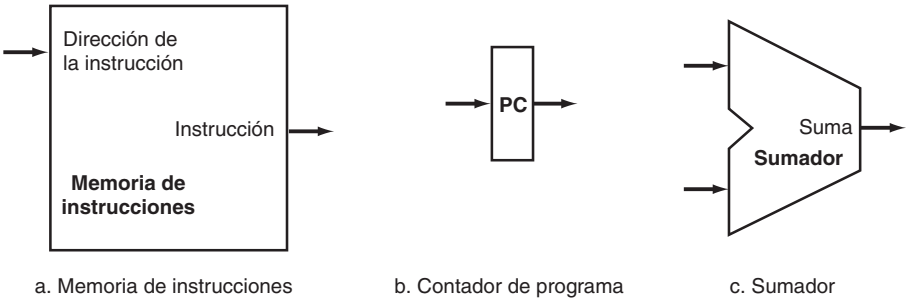


FIGURA 4.5 Se necesitan dos elementos de estado para almacenar y acceder a las instrucciones, y un sumador para calcular la dirección de la instrucción siguiente. Los elementos de estado son la memoria de instrucciones y el contador de programa. La memoria de instrucciones es solo de lectura, ya que el camino de datos nunca escribe instrucciones, y se trata como un elemento de lógica combinacional. La salida en cualquier instante refleja el contenido de la localización especificada por la dirección de entrada, y no se necesita ninguna señal de control de lectura (sólo se necesitará escribir en la memoria de las instrucciones cuando se cargue el programa; esto no es difícil de añadir, por lo que se ignora para simplificar). El contador de programa es un registro de 32 bits que se modifica al final de cada ciclo de reloj y, de esta manera, no necesita ninguna señal de control de escritura. El sumador es una ALU cableada para que sume siempre dos entradas de 32 bits y dé el resultado en su salida.

queta sumador como en la figura 4.5 para indicar que es un sumador permanente y que no puede realizar ninguna otra función propia de una ALU.

Para ejecutar cualquier instrucción se debe empezar por cargar la instrucción desde memoria. Para poder ejecutar la siguiente instrucción se debe incrementar el contador de programa para que apunte hacia ella 4 bytes más allá. La figura 4.6 muestra cómo se combinan los tres elementos de la figura 4.5 para formar un camino de datos que busca instrucciones e incrementa el PC para obtener la dirección de la siguiente instrucción secuencial.

Ahora considérense las instrucciones tipo R (véase la figura 2.20 de la página 136). Todas ellas leen dos registros, operan con la ALU los contenidos de dichos registros como operandos, y escriben el resultado. Estas instrucciones se denominan de tipo R o aritmético-lógicas (ya que realizan operaciones aritméticas o lógicas). En este tipo de instrucciones se incluyen las instrucciones `add`, `sub`, `and`, `or` y `slt`, que fueron introducidas en el capítulo 2. Recuérdese también que el ejemplo típico de este tipo de instrucciones es `add $t1,$t2,$t3`, que lee `$t2` y `$t3` y escribe en `$t1`.

Banco de registros: elemento de estado que consiste en un conjunto de registros que pueden ser leídos y escritos proporcionando el identificador del registro al que se desea acceder.

Los registros de 32 bits del procesador se agrupan en una estructura denominada **banco de registros**. Un banco de registros es una colección de registros donde cualquier registro puede leerse o escribirse especificando su número. El banco de registros contiene el estado de los registros de la máquina. Además, se necesita una ALU para poder operar con los valores leídos de los registros.

Debido a que las instrucciones tipo R tienen tres operandos registro, por cada instrucción se necesita leer dos datos del banco y escribir uno en él. Por cada registro que se lee, se necesita una entrada en el banco donde se especifique el número de registro que se quiere leer, así como una salida del banco donde se

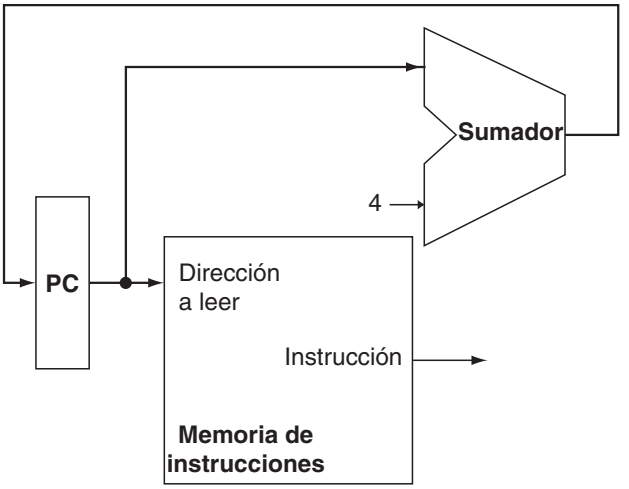


FIGURA 4.6 Parte del camino de datos utilizado para la búsqueda de las instrucciones y el incremento del contador de programa. La instrucción cargada se utiliza en otras partes del camino de datos.

presentará el valor correspondiente. Para escribir un valor se necesitan dos entradas: una para especificar el número de registro donde escribir y otra para suministrar el dato. El banco de registros siempre devuelve en sus salidas los contenidos de los registros cuyos identificadores están en las entradas de los registros a leer. Las escrituras se controlan mediante una señal de control de escritura, que debe estar activa para que una escritura se lleve a cabo en el flanco de reloj. Así, se necesitan un total de cuatro entradas (tres para los números de los registros y uno para los datos) y dos salidas (ambas de datos), como se puede ver en la figura 4.7a. Las entradas de los identificadores de registro son de cinco bits para especificar uno de los 32 ($32 = 2^5$) registros, mientras que los buses de la entrada y las dos salidas de datos son de 32 bits.

La figura 4.7b muestra la ALU, que tiene dos entradas de 32 bits y produce un resultado de 32 bits, así como una señal de 1 bit que se activa si el resultado es 0. La señal de control de cuatro bits de la ALU se describe en detalle en el **apéndice C**; repasaremos el control de la ALU brevemente cuando necesitemos saber cómo establecerlo.

Consideremos ahora las instrucciones del MIPS de carga y almacenamiento de palabras, las cuales tienen el formato general `lw $t1, despl($t2)` o `sw $t1, offset ($t2)`. Estas instrucciones calculan la dirección de memoria añadiendo al registro base `($t2)`, el campo de desplazamiento con signo de 16 bits contenido en la instrucción. Si la instrucción es un almacenamiento, el valor a almacenar debe leerse del registro especificado por `$t1`. En el caso de una carga, el valor leído de memoria debe escribirse en el registro especificado por `$t1` en el banco de registros. Así, se necesitarán tanto el banco de registros como la ALU que se muestran en la figura 4.7.

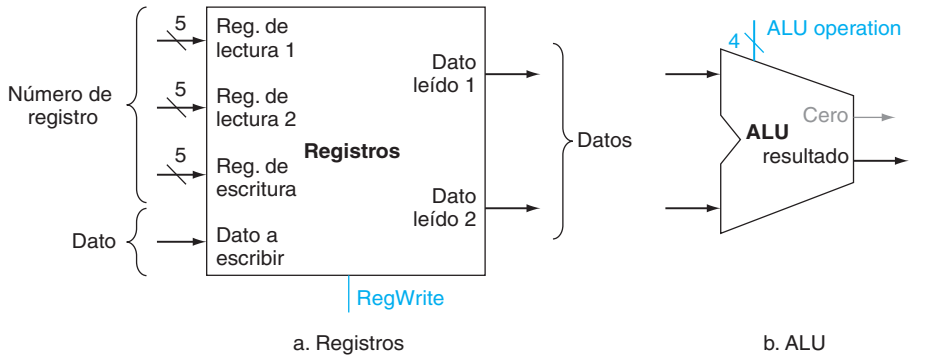


FIGURA 4.7 El banco de registros y la ALU son los dos elementos necesarios para la implementación de instrucciones de tipo R. El banco de registros contiene todos los registros y tiene dos puertos de lectura y uno de escritura. El diseño de los bancos de registros multipuerto se analiza en la sección C8 del apéndice C. El banco de registros siempre devuelve a la salida el contenido de los registros correspondientes a los identificadores que se encuentran en las entradas de los registros a leer; sin ser necesaria ninguna otra entrada de control. En cambio, la escritura de un registro debe indicarse explícitamente mediante la activación de la señal de control de escritura. Recuerde que las escrituras se sincronizan por flanco, por lo que todas las señales implicadas (el valor a escribir, el número de registro y la señal de control de escritura) deben ser válidas en el flanco de reloj. Por esta razón, este diseño puede leer y escribir sin ningún problema el mismo registro en un mismo ciclo: la lectura obtiene el valor escrito en un ciclo anterior, mientras que el valor que se escribe ahora estará disponible en ciclos siguientes. Las entradas que indican el número de registro al banco son todas de 5 bits, mientras que las líneas de datos son de 32 bits. La operación que realiza la ALU se controla mediante su señal de operación, que es de 4 bits, utilizando la ALU diseñada en el apéndice C. La salida de detección de Cero se utiliza para la implementación de los saltos condicionales. La salida de desbordamiento (*overflow*) no se necesitará hasta la sección 4.9, cuando se analicen las excepciones; por lo tanto se omitirá hasta entonces.

Extensión de signo: incrementar el tamaño de un dato mediante la replicación del bit de signo del dato original en los bits de orden alto del dato destino más largo.

Además se necesita una unidad para **extender el signo** del campo de desplazamiento de 16 bits de la instrucción a un valor con signo de 32 bits, y una unidad de memoria de datos para leer y escribir. La memoria de datos se escribe con instrucciones almacenamiento; por lo que tiene señales de control de escritura y de lectura, una entrada de dirección y una entrada de datos a escribir en memoria. La figura 4.8 muestra estos dos elementos.

Dirección destino de salto: dirección especificada en un salto que se convierte en la nueva dirección del contador de programas (PC) si se realiza el salto. En la arquitectura MIPS, el destino del salto viene dado por la suma del campo de desplazamiento de la instrucción y la dirección de la instrucción siguiente al salto.

La instrucción `beq` tiene tres operandos, dos registros que se comparan para comprobar su igualdad y un desplazamiento de 16 bits utilizado para calcular la **dirección destino del salto** relativa a la dirección de la instrucción. Su formato es `beq $t1,$t2,offset`. Para realizar esta instrucción se debe calcular la dirección destino del salto sumando el campo de desplazamiento con el signo extendido al PC. Hay dos detalles en la definición de las instrucciones de salto condicional (véase el capítulo 2) a los cuales se debe prestar atención:

- La arquitectura del repertorio de instrucciones especifica que es la dirección de la siguiente instrucción en orden secuencial la que se utiliza como base para el cálculo de la dirección destino. Puesto que se calcula el PC + 4 (la dirección de la siguiente instrucción) en el camino de datos de la carga de instrucciones, es fácil utilizar este valor como base para el cálculo de la dirección destino del salto.

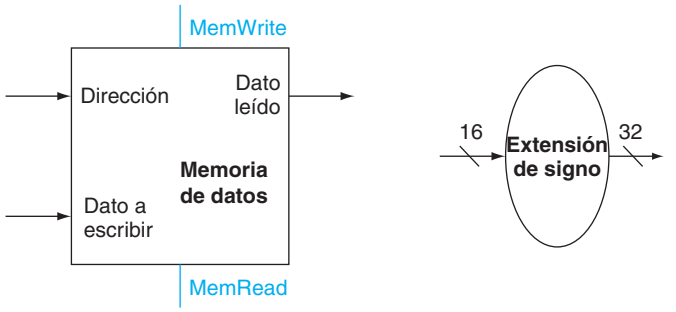


FIGURA 4.8 Las dos unidades necesarias para la implementación de cargas y almacenamientos, además del banco de registros y la ALU de la figura 4.7, son la unidad de memoria de datos y la unidad de extensión de signo. La unidad de memoria es un elemento de estado que tiene como entradas la dirección y el dato a escribir, y como única salida el valor leído. Hay controles separados de escritura y de lectura, aunque sólo uno de los dos puede estar activo en un momento determinado. La unidad de memoria requiere también una señal de lectura, ya que a diferencia del caso del banco de registros, la lectura de una dirección no válida puede causar problemas, como se verá en el capítulo 5. La unidad de extensión de signo tiene una entrada de 16 bits que se extenderá a los 32 bits que aparecen en la salida (véase capítulo 2). Se supone que la memoria de datos sólo escribe en el flanco. De hecho, los chips estándar de memoria tienen una señal de habilitación de escritura y, aunque esta señal no es con flanco, este diseño podría fácilmente adaptarse para trabajar con chips reales de memoria. Véase la sección C.8 del apéndice C para analizar más ampliamente cómo trabajan estos chips.

- La arquitectura también impone que el campo de desplazamiento se desplace hacia la izquierda 2 bits para que este desplazamiento corresponda a una palabra; de forma que se incrementa el rango efectivo de dicho campo en un factor de 4.

Para tratar la última complicación se necesita desplazar dos posiciones el campo de desplazamiento.

Además de calcular la dirección destino del salto, también se debe determinar si la siguiente instrucción a ejecutar es la que sigue secuencialmente o la situada en la dirección destino del salto. Cuando la condición se cumple (es decir, los operandos son iguales), la dirección calculada pasa a ser el nuevo PC, y se dice que el **salto condicional se ha tomado**. Si los operandos son diferentes, el PC incrementado debería reemplazar al PC actual (igual que para cualquier otra instrucción); en este caso se dice que el **salto no se ha tomado**.

Salto tomado: salto en el que se cumple la condición de salto y el contador de programa (PC) es cargado con la dirección destino. Todos los saltos incondicionales son saltos tomados.

Salto no tomado: salto donde la condición de salto es falsa y el contador de programa (PC) se carga con la dirección de la instrucción siguiente al salto.

Resumiendo, el camino de datos para saltos condicionales debe efectuar dos operaciones: calcular la dirección destino del salto y comparar el contenido de los registros (los saltos condicionales también requieren que se modifique la parte de carga de instrucciones del camino de datos, como se verá más adelante). La figura 4.9 muestra el camino de datos de los saltos condicionales. Para calcular la dirección destino del salto, el camino de datos incluye una unidad de extensión de signo, como en la figura 4.8, y un sumador. Para la comparación se necesita utilizar el banco de registros mostrado en la figura 4.7a a fin de obtener los dos regis-

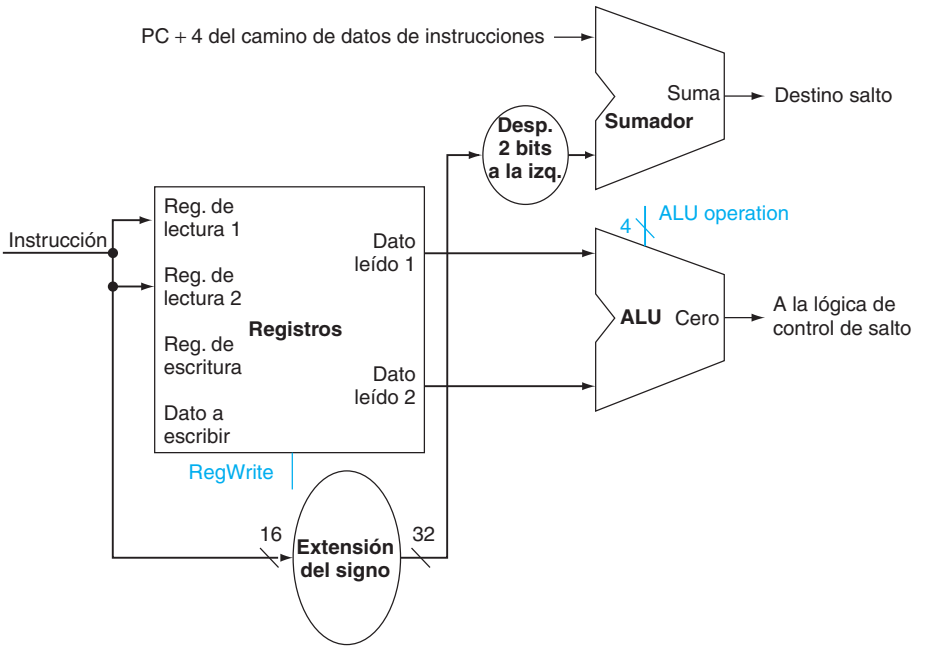


FIGURA 4.9 El camino de datos para un salto condicional utiliza la ALU para evaluar la condición de salto y un sumador aparte para calcular la dirección destino del salto como la suma del PC incrementado y los 16 bits de menor peso de la instrucción con el signo extendido (el desplazamiento de salto) y desplazado dos bits a la izquierda. La unidad etiquetada como «desp. 2 bits a la izq.» es simplemente un encaminamiento de las señales entre la entrada y la salida que añade 00_{dos} en la zona baja del campo de desplazamiento. No se necesita circuitería para el desplazamiento, ya que la cantidad a desplazar es constante. Como se sabe que el desplazamiento se extiende desde 16 bits, esta operación sólo desperdicia bits de signo. La lógica de control se utiliza para decidir si el PC incrementado o el destino del salto deberían reemplazar al PC, basándose en la salida Cero de la ALU.

tros de operando (aunque no será necesario escribir en él) y la ALU (diseñada en el apéndice C) para realizar dicha operación. Ya que esta ALU proporciona una señal de salida que indica si el resultado era 0, se le pueden enviar los dos operandos con la señal de control activada de forma que efectúe una resta. Si la señal Cero a la salida de la ALU está activa, entonces los dos valores son iguales. Aunque la salida Cero siempre indica si el resultado es 0, sólo se utiliza para realizar el test de igualdad en saltos condicionales. Más tarde se estudiará de forma más exacta cómo se conectan las señales de control de la ALU para utilizarla en el camino de datos.

La instrucción de salto incondicional reemplaza los 28 bits de menor peso del PC con los 26 bits de menor peso de la instrucción desplazados 2 bits hacia la izquierda. Este desplazamiento se realiza simplemente concatenando 00 a estos 26 bits (como se describe en el capítulo 2).

Extensión: En el repertorio de instrucciones del MIPS, los **saltos condicionales se retardan**, lo cual significa que la siguiente instrucción inmediatamente después del salto se ejecuta siempre, independientemente de si la condición de salto se cumple o no. Cuando la condición es falsa, la ejecución se comporta como un salto normal. Cuando es cierta, un salto retardado primero ejecuta la instrucción inmediatamente posterior al salto y posteriormente salta a la dirección destino. El motivo de los saltos retardados surge por el modo en que les afecta la segmentación (véase sección 4.8). Para simplificar, se ignorarán los saltos retardados en este capítulo y se realizará una instrucción `beq` no retardada.

Salto retardado: tipo de salto en el que la instrucción immediatamente siguiente al salto se ejecuta siempre, independientemente de si la condición del salto es verdadera o falsa.

Implementación de un camino de datos sencillo

Una vez que se han descrito los componentes necesarios para los distintos tipos de instrucciones, éstos pueden combinarse en un camino de datos sencillo y añadir el control para completar la implementación. El más sencillo de los diseños intentará ejecutar todas las instrucciones en un solo ciclo. Esto significa que ningún elemento del camino de datos puede utilizarse más de una vez por instrucción, de forma que cualquier recurso que se necesite más de una vez deberá estar replicado. Por tanto, la memoria de instrucciones ha de estar separada de la memoria de datos. Aunque se necesite duplicar algunas de las unidades funcionales, muchos de estos elementos pueden compartirse en los diferentes flujos de instrucciones.

Para compartir un elemento del camino de datos entre dos tipos de instrucciones diferentes, se requiere que dicho elemento disponga de múltiples conexiones a la entrada de un elemento utilizando un multiplexor, así como de una señal de control para seleccionar entre las múltiples entradas.

Construcción de un camino de datos

EJEMPLO

El camino de datos de las instrucciones aritmético-lógicas (o tipo R), así como el de las instrucciones de referencia a memoria son muy parecidos, siendo las principales diferencias las siguientes:

- Las instrucciones aritmético-lógicas utilizan como entradas de la ALU los valores procedentes de dos registros. Las instrucciones de referencia a memoria pueden utilizar la ALU para realizar el cálculo de la dirección, aunque la segunda entrada es el campo de desplazamiento de 16 bits procedente de la instrucción con signo extendido.
- El valor guardado en el registro destino, o bien proviene de la ALU (para instrucciones tipo R) o de memoria (en caso de una carga).

Determine cómo construir un camino de datos para la parte operacional de las instrucciones de referencia a memoria y aritmético-lógicas que utilice un único banco de registros y una sola ALU para soportar ambos tipos de instrucciones, añadiendo los multiplexores necesarios.

RESPUESTA

Para combinar ambos caminos de datos y usar un único banco de registros y una sola ALU, la segunda entrada de ésta ha de soportar dos tipos de datos diferentes, además de dos posibles caminos para el dato a almacenar en el banco de registros. De esta manera, se coloca un multiplexor en la entrada de la ALU y un segundo en la entrada de datos del banco de registros. La figura 4.10 muestra este nuevo camino de datos.

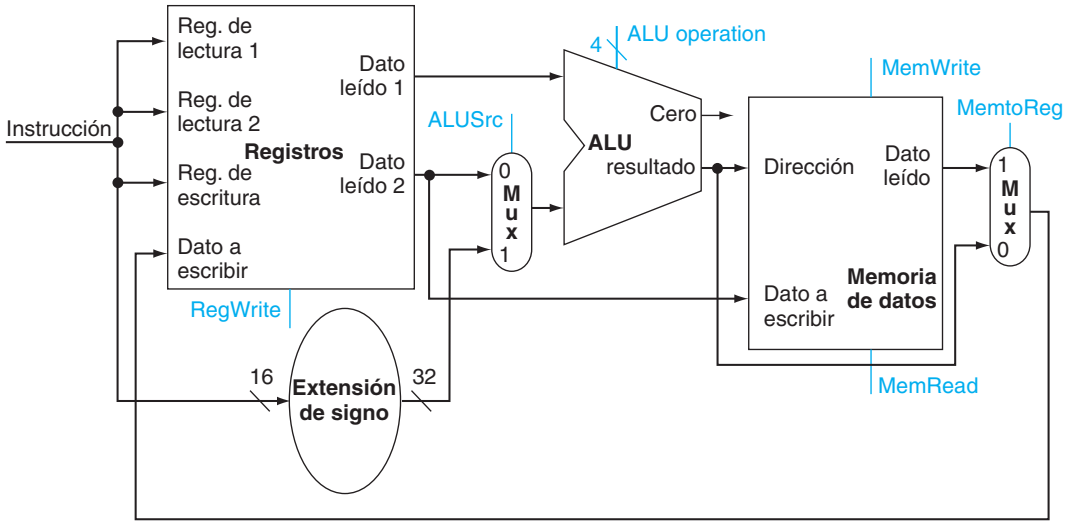


FIGURA 4.10 El camino de datos para las instrucciones de memoria y tipo R. Este ejemplo muestra cómo puede obtenerse un camino uniendo diferentes piezas (figuras 4.7 y 4.8) y añadiendo multiplexores. Se requieren dos multiplexores tal como se describe en el ejemplo.

Ahora se pueden combinar el resto de las piezas para obtener un camino de datos único para la arquitectura MIPS, añadiendo la parte encargada de la búsqueda de las instrucciones (figura 4.6), al camino de datos de las instrucciones tipo R y de referencia a memoria (figura 4.10), y el camino de datos para los saltos (figura 4.9). La figura 4.11 muestra este camino de datos obtenido al ensamblar las diferentes piezas. Las instrucciones de salto utilizan la ALU para la comparación de los registros fuente, de forma que debe ponerse el sumador de la figura 4.9 para calcular la dirección de destino del salto. También es necesario un nuevo multiplexor para escoger entre seguir en secuencia (PC + 4) y la dirección destino del salto para escribir esta nueva dirección en el PC.

Una vez completado este sencillo camino de datos, se puede añadir la unidad de control. Ésta debe ser capaz de generar las señales de escritura para cada elemento de estado y las de control para la ALU y para cada multiplexor a partir de las entradas. Debido a que el control de la ALU es diferente del resto en mayor o menor medida, resultaría útil diseñarlo como paso previo al diseño de la unidad de control.

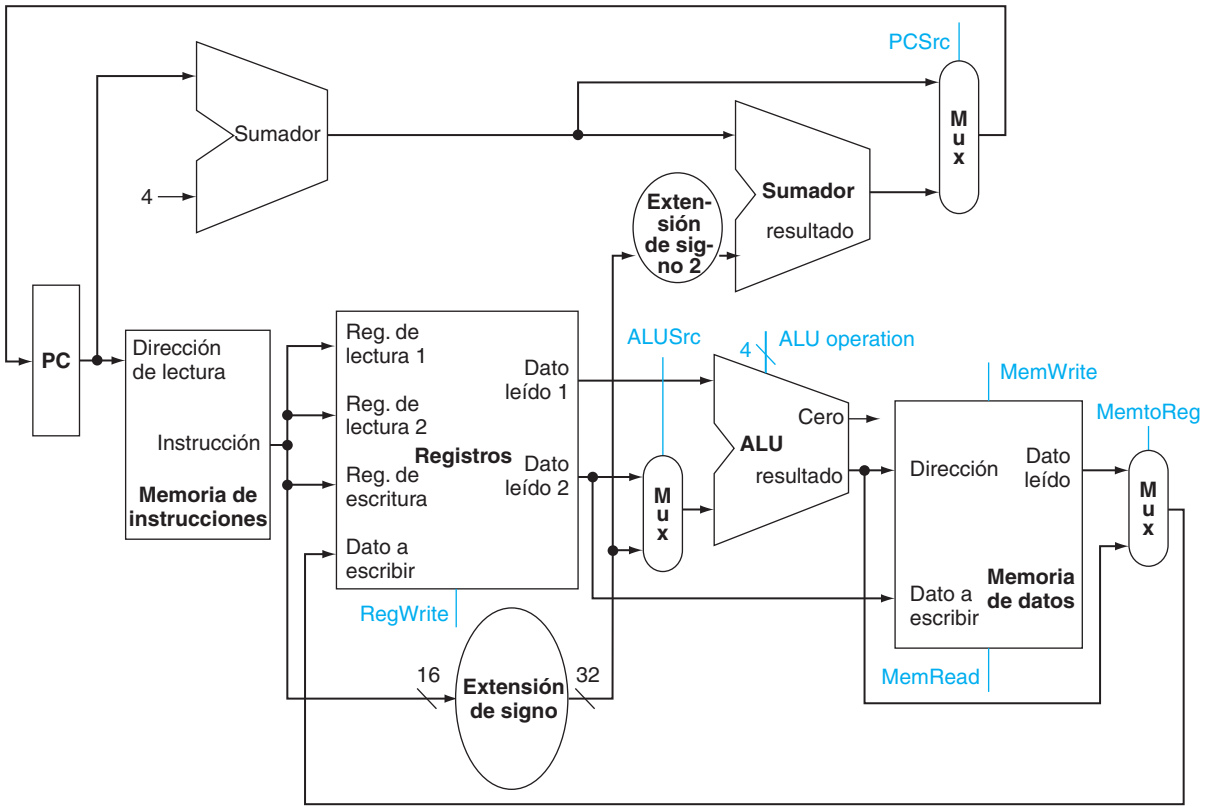


FIGURA 4.11 Un camino de datos sencillo para la arquitectura MIPS que combina los elementos necesarios para los diferentes tipos de instrucciones. Este camino de datos puede ejecutar las instrucciones básicas (carga/almacenamiento de palabras, operaciones de la ALU y saltos) en un solo ciclo de reloj. Es necesario un nuevo multiplexor para integrar saltos condicionales. La lógica necesaria para ejecutar instrucciones de salto incondicional (*jump*) se añadirá más tarde.

Autoevaluación

I. ¿Cuáles de las siguientes afirmaciones son correctas para una instrucción de carga? Utilizar la figura 4.10.

a. MemtoReg debe ser activada para provocar que el dato procedente de memoria sea enviado al banco de registros.

b. MemtoReg debe se activada para provocar que el registro destino correcto sea enviado al banco de registros.

c. La activación de MemtoReg no es relevante.

II. El camino de datos de ciclo único descrito conceptualmente en esta sección debe tener memorias de datos e instrucciones separadas, porque

a. En MIPS los formatos de datos e instrucciones son diferentes, y por lo tanto, se necesitan memorias diferentes.

b. Tener memorias separadas es más barato.

c. El procesador opera en un solo ciclo y no puede usar una memoria con un único puerto para dos acceso diferentes en el mismo ciclo.

4.4 Esquema de una implementación simple

En esta sección veremos la que podría considerarse como la implementación más sencilla posible de nuestro subconjunto de instrucciones MIPS. Se construirá esta sencilla implementación utilizando el camino de datos de la sección anterior y añadiendo una función de control simple. Esta implementación simple será capaz de ejecutar instrucciones de almacenamiento y carga de palabras de memoria (`lw` y `sw`), saltar si igual (`beq`), instrucciones aritméticas (`add`, `sub`, `and`, `or` y `slt`) y activar si es menor que (`set on less than`). Posteriormente se mejorará dicho diseño incluyendo la instrucción de salto incondicional (`j`).

El control de la ALU

La ALU MIPS del apéndice C define las siguientes 6 combinaciones de 4 entradas de control:

Líneas de conrol de la ALU	Función
0000	Y-lógico (AND)
0001	O-lógico (OR)
0010	sumar
0110	restar
0111	iniciar si menor que
1100	NOR

Dependiendo del tipo de instrucción a ejecutar, la ALU debe realizar una de las cinco primeras operaciones (NOR es necesaria para otras partes del repertorio de instrucciones MIPS). Las instrucciones de referencia a memoria utilizan la ALU para calcular la dirección de memoria por medio de una suma. Para las instrucciones tipo R, la ALU debe ejecutar una de las cinco operaciones (`and`, `or`, `add`, `sub` y `slt`) en función del valor de los 6 bits de menor peso de la instrucción, los cuales componen el código de función (véase el capítulo 2), mientras que para las instrucciones de saltar si igual, la ALU debe realizar una resta.

Mediante una pequeña unidad de control que tiene como entradas el código de función de la instrucción y 2 bits de control adicionales que reciben el nombre de ALUOp, se pueden generar los 4 bits que conforman las señales de control de la ALU. Estos bits (ALUOp) indican si la operación a realizar debería ser o bien una suma (00) para accesos a memoria, o bien una resta (01) para saltos condicionales, o bien si la operación a realizar está codificada en el código de función (10). La

salida de la unidad de control de la ALU es una señal de 4 bits que controla la ALU codificando una de las combinaciones de 4 bits mostradas anteriormente.

En la figura 4.12 se observa el conjunto de combinaciones de las señales de entrada formadas por los 2 bits que conforman la señal de ALUOp y los 6 bits del código de función. Posteriormente, en este capítulo, se verá cómo genera la unidad de control principal los bits de ALUOp.

Código de operación	ALUOp	Operación	Campo de la función	Acción deseada de la ALU	Entrada del control de la ALU
LW	00	cargar palabra	XXXXXX	sumar	0010
SW	00	almacenar palabra	XXXXXX	sumar	0010
Branch equal	01	saltar si igual	XXXXXX	restar	0110
R-type	10	sumar	100000	sumar	0010
R-type	10	restar	100010	restar	0110
R-type	10	AND	100100	Y lógica	0000
R-type	10	OR	100101	O lógica	0001
R-type	10	activar si es menor que	101010	activar si es menor que	0111

FIGURA 4.12 Cálculo de los bits de control de la ALU en función de los bits de ALUOp y los diferentes códigos de función de las instrucciones tipo R. El código de operación, que aparece en la primera columna, determina los valores de los bits del campo ALUOp. Todas las codificaciones se muestran en binario. Obsérvese que cuando ALUOp tiene como valor 00 ó 01, la salida no depende del código de función, y se dice que su valor es no-determinado y codificado como XXXXXX. En el caso de que ALUOp valga 10, el código de la función se utiliza para calcular la señal de control de la ALU. Véase el apéndice C.

Esta técnica basada en el uso de múltiples niveles de descodificación (o decodificación), es decir, la unidad de control principal genera los bits de ALUOp que se utilizarán como entrada de la unidad de control encargada de generar las señales de la ALU, es muy común. El hecho de usar múltiples niveles puede reducir el tamaño de la unidad principal. De igual manera, el uso de varias unidades de control más pequeñas puede incluso incrementar la velocidad de dicha unidad de control. Todas estas optimizaciones son importantes ya que, a menudo, la unidad de control se encuentra en el camino crítico.

Existen diferentes formas de establecer la correspondencia entre los 2 bits del campo de ALUOp y los 6 del código de función con los 3 bits que conforman la operación a realizar por la ALU. Debido a que sólo una pequeña parte de los 64 valores posibles del código de función son importantes y que dichos bits únicamente se utilizan cuando ALUOp vale 10, podría utilizarse una pequeña parte de lógica que reconociera dicho subconjunto de valores posibles y diera como resultado los valores correctos de los bits de control de la ALU.

Como paso previo al diseño de la lógica combinacional, es útil construir una tabla de verdad para aquellas combinaciones de interés de los códigos de función y de los bits de ALUOp, tal como se ha realizado en la figura 4.13. Esta tabla muestra cómo dependen de ambos campos las señales de control de la ALU. Debido a que la **tabla de verdad** es muy grande ($2^8 = 256$ entradas), y teniendo en cuenta que para muchas de dichas combinaciones los valores de la salida no tienen importancia, únicamente

Tabla de verdad: desde el punto de vista lógico, es una representación de una operación lógica que muestra todos los valores de las entradas y el valor de las salidas para cada caso.

ALUOp		Campo de la función						Operación
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURA 4.13 Tabla de verdad de los 3 bits de control de la ALU (también llamados operación). Las entradas son la ALUOp y el código de función. Únicamente se muestran aquellas entradas para las cuales la señal de control de la ALU tiene sentido. También se han añadido algunas entradas cuyo valor es indeterminado. Por ejemplo, el campo ALUOp no utiliza la codificación 11, de forma que la tabla de verdad puede contener las entradas 1X y X1 en vez de 10 y 01. También, cuando se utiliza el código de función, los 2 primeros bits (F5 y F4) de dichas instrucciones son siempre 10, de forma que también se consideran no-determinados y se reemplazan por XX en la tabla de verdad.

se dan los valores de las salidas para aquellas entradas de la tabla donde el control de la ALU debe tener un valor específico. Las diferentes tablas de verdad que se irán viendo a lo largo de este capítulo contendrán únicamente aquellos subconjuntos de entradas que deban estar activadas, eliminando aquellos cuyos valores de salida sean indeterminados. (Este método tiene un inconveniente que se analizará en la sección D.2 del apéndice D.

Términos indeterminados: elementos de una función lógica cuya salida no depende de los valores de todas las entradas. Los términos no-determinados pueden especificarse de distintas maneras.

Debido a que en muchos casos los valores de algunas entradas no tienen importancia, para mantener la tabla compacta incluimos **términos indeterminados**. Un término de este tipo (representado en la tabla mediante una X en la columna de entrada correspondiente) indica que la salida es independiente del valor de dicha entrada. Por ejemplo, cuando el campo ALUOp vale 00, caso de la primera fila de la tabla de la figura 4.13, la señal de control de la ALU siempre será 0010, independientemente del código de función. Es decir, en este caso, el código de la función se considera indeterminado en esta fila de la tabla de verdad. Más adelante se verán ejemplos de otro tipo de términos indeterminados. Si no se está familiarizado con este tipo de términos, véase el apéndice C para mayor información.

Una vez que se ha construido la tabla de verdad, ésta puede optimizarse y entonces pasar a su implementación mediante puertas lógicas. Este proceso es completamente mecánico. Así, en lugar de mostrar aquí los pasos finales, este proceso, así como sus resultados, se describe en la sección D.2 del apéndice D.

Diseño de la unidad de control principal

Una vez que ya se ha descrito como diseñar una ALU que utiliza el código de función y una señal de dos bits como entradas de control, podemos centrarnos en el resto del control. Para iniciar este proceso se deben identificar los campos de las instrucciones y las líneas de control necesarias para la construcción del camino de datos mostrado en la figura 4.11. Para entender mejor cómo se conectan los diferentes campos de las instrucciones en el camino de datos, sería útil revisar los formatos de los tres tipos de instrucciones: el tipo R (aritmético-lógica), los saltos y las operaciones de carga y almacenamiento. Estos formatos se muestran en la figura 4.14.

Campo	0	rs	rt	rd	shamt	funct
Posición de los bits	31:26	25:21	20:16	15:11	10:6	5:0

a. Instrucción tipo R

Campo	35 o 43	rs	rt	dirección
Posición de los bits	31:26	25:21	20:16	15:0

b. Instrucción de carga o almacenamiento

Campo	4	rs	rt	dirección
Posición de los bits	31:26	25:21	20:16	15:0

c. Instrucción de salto condicional

FIGURA 4.14 Los tres tipos de instrucciones (tipo R, referencia a memoria y salto condicional) utilizan dos formatos de instrucción diferentes. La instrucción de salto incondicional hace uso de otro formato, que se explicará más adelante. (a) Formato para las instrucciones tipo R, que tiene el campo de tipo de operación a 0. Estas instrucciones disponen de tres operandos registros: rs, rt y rd, donde rs y rt son registros fuente y rd es el registro destino. La función a realizar en la ALU se encuentra en el código de función (campo *funct*) y es descodificada por la unidad de control de la ALU, tal como se ha visto en la sección anterior. Pertenecen a este tipo las instrucciones `add`, `sub`, `and`, `or` y `slt`. El campo de desplazamiento (*shamt*) sólo se utiliza para desplazar y se ignora en este capítulo. (b) Formato de las instrucciones carga (código de operación 35_{diez}) y almacenamiento (código de operación 43_{diez}). El registro rs se utiliza como registro base al cual se le añade el campo de dirección de 16 bits para obtener la dirección de memoria. En el caso de las instrucciones de carga, rt es el registro destino del valor cargado de memoria, mientras que en los almacenamientos, rt es el registro fuente del valor que se debe almacenar en memoria. (c) Formato de las instrucciones de saltar si igual (código de operación 4_{diez}). Los registros rs y rt son los registros fuente que se compararán. Al campo de 16 bits de la dirección se le extiende el signo, se desplaza y se suma al PC para calcular la dirección de destino de salto.

Existen varias observaciones importantes a realizar sobre este formato de las instrucciones y que siempre se supondrán ciertas.

Opcode: campo que denota la operación y formato de una instrucción.

- El campo de código de operación (**opcode**) estará siempre contenido en los bits 31-26. Se referenciará dicho campo como Op[5-0].
- Los dos registros de lectura están siempre especificados en los campos rs y rt en las posiciones 25-21 y 20-16. Este hecho se cumple para las instrucciones tipo R, beq y almacenamiento.
- El registro base para las instrucciones de acceso a memoria se encuentra siempre en rs (bits 25-21).
- El desplazamiento relativo de 16 bits para saltar si igual (`beq`), cargas y almacenamientos está siempre en las posiciones 15-0.
- El registro destino puede estar en dos lugares. Para instrucciones de carga, su posición es 20-16 (rt), mientras que en las instrucciones aritmético-lógicas está en los bits 15-11 (rd). Esto implica tener que añadir un multiplexor para seleccionar cuál de estos dos campos de la instrucción va a utilizarse en el momento de indicar el registro en el que se va a escribir.

El primer principio de diseño del capítulo 2, *la sencillez favorece la regularidad*, vale para la especificación del control.

Mediante esta información pueden añadirse las etiquetas de las diferentes partes de las instrucciones y el multiplexor adicional (para el registro destino) a nuestro sencillo camino de datos. La figura 4.15 muestra estos añadidos, que incluyen la unidad de control de la ALU, las señales de escritura de los elementos de estado, la señal de lectura de la memoria de datos y las señales de control de los multiplexores. Debido a que estos últimos únicamente tienen dos entradas, sólo requieren una única línea de control cada uno.

La figura 4.15 muestra las siete señales de control de 1 bit a las que hay que añadir la señal de ALUOp (de 2 bits). El funcionamiento de dicha señal ya se ha explicado y ahora sería útil definirlo informalmente para el resto de las señales antes de determinar cómo van a trabajar durante la ejecución de las instrucciones. La figura 4.16 describe la funcionalidad de dichas líneas de control.

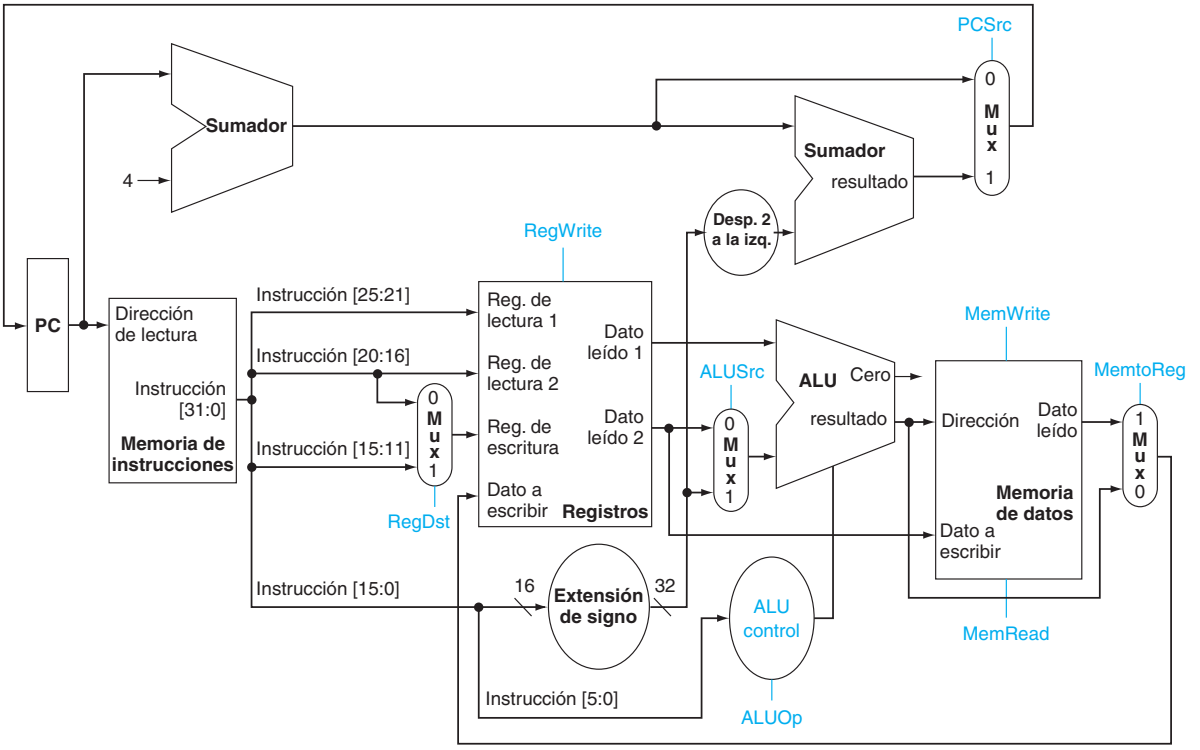


FIGURA 4.15 El camino de datos de la figura 4.12 con todos los multiplexores necesarios y todas las líneas de control identificadas. Las líneas de control se muestran en color. También se ha añadido el bloque encargado de controlar la ALU. El registro PC no necesita un control de escritura ya que sólo se escribe una vez al final de cada ciclo de reloj. La lógica de control del salto es la que determina si el nuevo valor será el valor anterior incrementado o la dirección destino del salto.

Señal de control	Efecto cuando no está activa	Efecto cuando está activa
RegDst	El identificador del registro destino viene determinado por el campo rt (bits 20-16).	El identificador del registro destino viene determinado por el campo rd (bits 15-11).
RegWrite	Ninguno.	El registro destino se actualiza con el valor a escribir.
ALUSrc	El segundo operando de la ALU proviene del segundo registro leído del banco de registros.	El segundo operando de la ALU son los 16 bits de menor peso de la instrucción con el signo extendido.
PCSrc	El PC es reemplazado por su valor anterior más 4 (PC + 4).	El PC es reemplazado por la salida del sumador que calcula la dirección destino del salto.
MemRead	Ninguno.	El valor de la posición de memoria designada por la dirección se coloca en la salida de lectura.
MemWrite	Ninguno.	El valor de la posicón de memoria designada por la dirección se reemplaza por el valor de la entrada de datos.
MemtoReg	El valor de entrada del banco de registros proviene de la ALU.	El valor de entrada del banco de registros proviene de la memoria.

FIGURA 4.16 El efecto de cada una de las siete señales de control. Cuando el bit de control encargado de controlar un multiplexor de dos vías está a 1, dicho multiplexor seleccionará la entrada etiquetada como l. De lo contrario, si el control está desactivado, se tomará la entrada etiquetada como 0. Recuerde que todos los elementos de estado tienen un reloj como señal de entrada implícita y cuya función es controlar las escrituras. Nunca se hace atravesar puertas al reloj, ya que puede crear problemas. (Véase en apéndice C una discusión más detallada de este problema.)

Una vez definidas las funciones de cada una de las señales de control, se puede pasar a ver cómo activarlas. La unidad de control puede activar todas las señales en función de los códigos de operación de las instrucciones exceptuando una de ellas (la señal PCSrc). Esta línea de control debe activarse si la instrucción es de tipo salto condicional (decisión que puede tomar la unidad de control) y la salida Cero de la ALU, que se utiliza en las comparaciones, está a 1. Para generar la señal de PCSrc se necesita aplicar una AND a una señal llamada *Branch*, que viene de la unidad de control, con la señal Cero procedente de la ALU.

Estas nueve señales de control (las siete de la figura 4.16, más las dos de la señal ALUOp), pueden activarse en función de las seis señales de entrada de la unidad de control, las cuales pertenecen al código de operación, bits 31 a 26. El camino de datos con la unidad y las señales de control se muestra en la figura 4.17.

Antes de intentar escribir el conjunto de ecuaciones de la tabla de verdad de la unidad de control, sería útil definirla informalmente. Debido a que la activación de las líneas de control únicamente depende del código de operación, se define si cada una de las señales de control debería valer 0, 1 o bien indeterminada (X), para cada uno de los códigos de operación. La figura 4.18 define cómo deben activarse las señales de control para cada código de operación; información que proviene directamente de las figuras 4.12, 4.16 y 4.17.

Funcionamiento del camino de datos

Con la información contenida en las figuras 4.16 y 4.18, se puede diseñar la lógica de la unidad de control, pero antes de esto, se verá cómo cada instrucción hace uso del

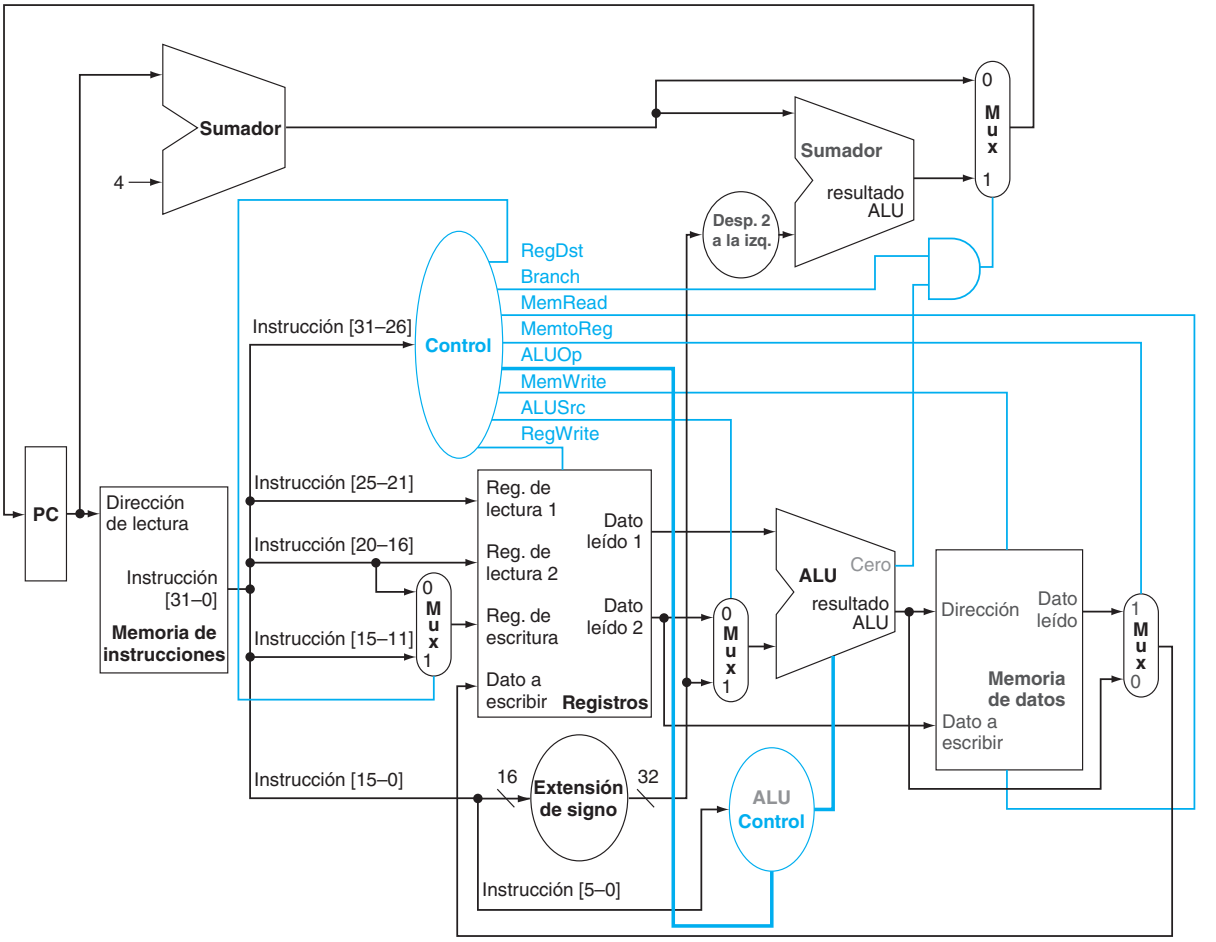


FIGURA 4.17 Un sencillo camino de datos con la unidad de control. La entrada de la unidad de control está compuesta por los seis bits pertenecientes al campo del código de operación de la instrucción. Las salidas de dicha unidad consisten en tres señales de un bit que se utilizan para controlar los multiplexores (RegDst, ALUSrc, y MemtoReg), tres señales para controlar lecturas y escrituras en el banco de registros y en la memoria (RegWrite, MemRead, y MemWrite), una señal de 1 bit para determinar si es posible saltar (*branch*), y una señal de control de 2 bits para la ALU (ALUOp). Se utiliza una puerta AND para combinar la señal de control *branch* de la unidad de control y la salida Cero de la ALU. Dicha puerta será la que controle la selección del próximo PC. Obsérvese que ahora PCSrc es una señal derivada aunque una de sus partes proviene directamente de la unidad de control. En adelante se suprimirá este nombre.

camino de datos. En las siguientes figuras se muestra el flujo a través del camino de datos de las tres clases diferentes de instrucciones. Las señales de control activadas, así como los elementos activos del camino de datos, están resaltados en cada caso. Obsérvese que un multiplexor cuyo control está a 0 es una acción definida, a pesar de que su línea de control no esté destacada. Las señales de control formadas por múltiples bits están destacadas si cualquiera de los que la constituyen está activa.

Instrucción	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
Formato R	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

FIGURA 4.18 La activación de las líneas de control está completamente determinada por el código de operación de las instrucciones. La primera fila de la tabla corresponde a las instrucciones tipo R (`add`, `sub`, `and`, `or` y `slt`). En todas estas instrucciones, los registros fuente se encuentran en rs y rt, y el registro destino en rd, de forma que las señales ALUSrc y RegDst deben estar activas. Además, este tipo de instrucciones siempre escribe en un registro (RegWrite = 1) y en ningún caso accede a la memoria de datos. Cuando la señal *Branch* está a 0, el PC se reemplaza de forma incondicional por PC + 4; en otro caso, el registro de PC se reemplaza por la dirección destino del salto si la salida Cero de la ALU también está en nivel alto. El campo ALUOp para las instrucciones tipo R siempre tiene el valor de 10 para indicar que las señales de control de la ALU deben generarse con ayuda del campo de función. La segunda y tercera fila de la tabla contienen el valor de las señales de control para las instrucciones `lw` y `sw`. En ambos casos, ALUSrc y ALUOp están activas para poder llevar a cabo el cálculo de direcciones. Las señales MemRead y MemWrite estarán a 1 según el tipo de acceso. Finalmente, cabe destacar que RegDst y RegWrite deben tener los valores 0 y 1, respectivamente, en caso de una carga, para que el valor se almacene en el registro rt. El control de las instrucciones de salto es similar a las tipo R ya que también envía los registros rs y rt a la ALU. La señal ALUOp, en caso de instrucción `beq`, toma el valor 01, para que la ALU realice una operación de resta y compruebe así la igualdad. Observe que el valor de la señal MemtoReg es irrelevante cuando RegWrite vale 0 (como el registro no va a ser escrito, el valor del dato existente en el puerto de escritura del banco de registros no va a ser utilizado). De esta manera, en las dos últimas filas de la tabla, los valores de la señal MemtoReg se han reemplazado por X, que indica términos indeterminados. De igual forma, este tipo de términos pueden añadirse a RegDst cuando RegWrite vale 0. Este tipo de términos los debe añadir el diseñador, pues dependen del conocimiento que se tenga sobre el funcionamiento del camino de datos.

La figura 4.19 muestra el funcionamiento de una instrucción tipo R, tal como `add $t1, $t2, $t3`. Aunque todo sucede en un único ciclo de reloj, la ejecución de una instrucción se puede dividir en cuatro pasos; estos pasos se pueden ordenar según el flujo de información:

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.
2. Se leen los registros `$t2` y `$t3` del banco de registros. Adicionalmente, la unidad de control principal se encarga de calcular la activación de las señales de control durante este paso.
3. La ALU se encarga de realizar la operación adecuada con los datos procedentes del banco de registros, utilizando para ello el campo de función (bits 5-0) para obtener la función de la ALU.
4. El resultado calculado en la ALU se escribe en el banco de registros utilizando los bits 15-11 de la instrucción para seleccionar el registro destino (`$t1`).

De forma similar, se puede ilustrar la ejecución de una instrucción de carga, tal como

```
lw $t1, offset($t2)
```

de un modo parecido a la figura 4.19. La figura 4.20 muestra las unidades funcionales y las líneas de control activadas para esta instrucción. Puede verse una carga como una instrucción que se ejecuta en 5 pasos (similares a los de las instrucciones tipo R, que se ejecutaban en 4).

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.
2. Se 1ee el valor del registro `$t2` del banco de registros.

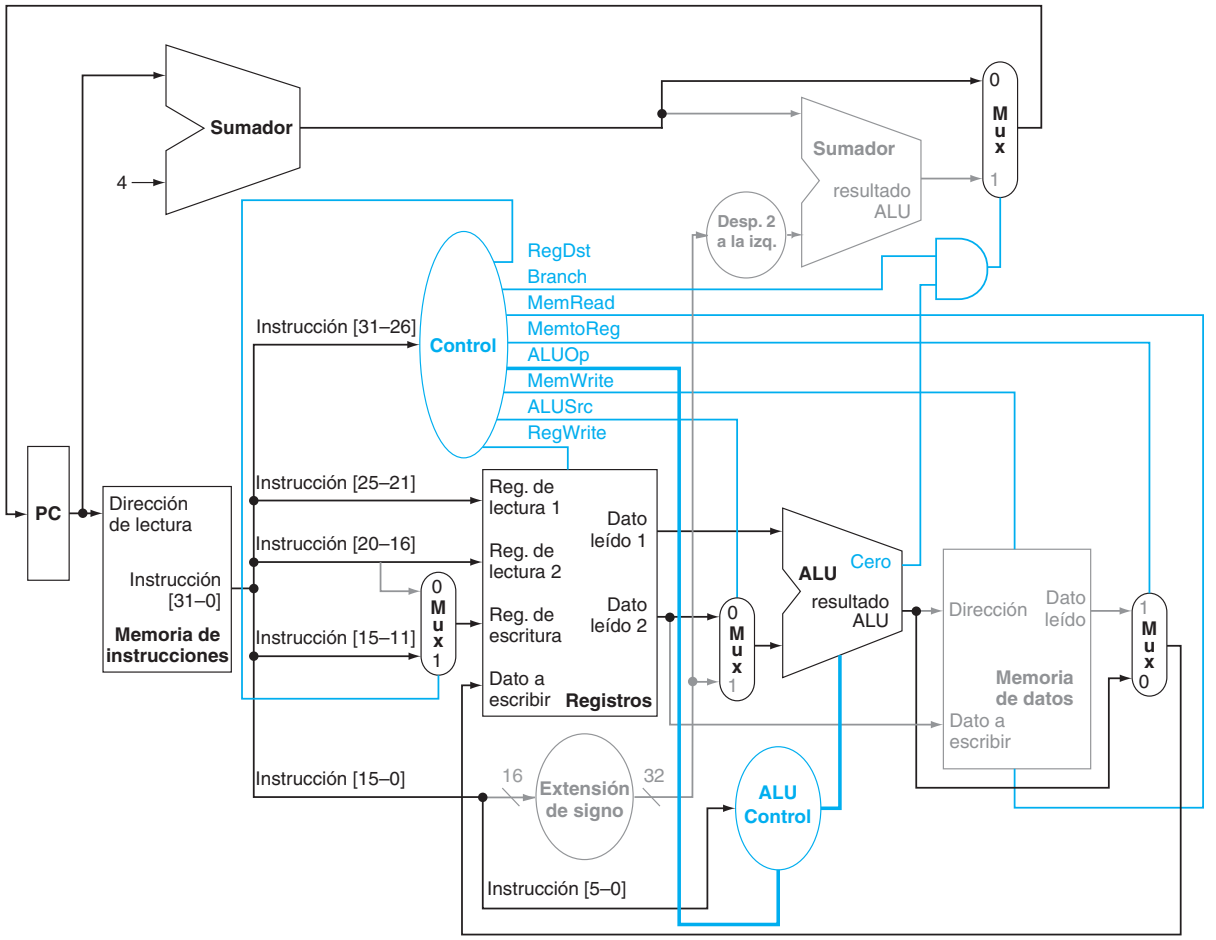


FIGURA 4.19 Funcionamiento del camino de datos para una instrucción R-type tal como `add $t1, $t2, $t3`. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas.

3. La ALU calcula la suma del valor leído del banco de registros y los 16 bits de menor peso de la instrucción, con el signo extendido (`offset`).

4. Dicha suma se utiliza como dirección para acceder a la memoria de datos.

5. El dato procedente de la memoria se escribe en el banco de registros. El registro destino viene determinado por los bits 20-16 de la instrucción (`$t1`).

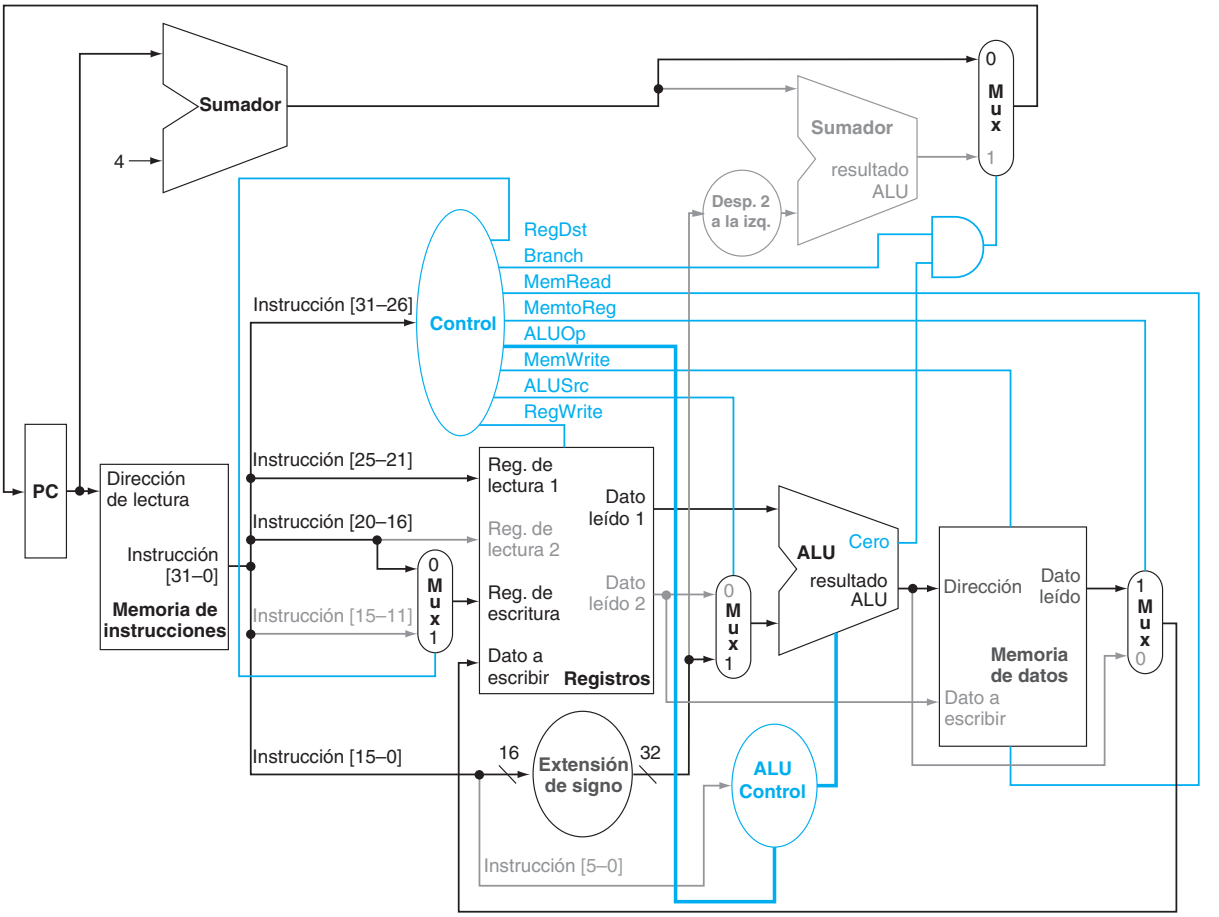


FIGURA 4.20 Funcionamiento del camino de datos para una instrucción de carga. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Una instrucción de almacenamiento opera de forma similar. La diferencia principal es que el control de la memoria indicará una escritura en lugar de una lectura, el valor del segundo registro contendría el dato a almacenar, y la operación final de escribir el valor de memoria en el banco de registros no se realizaría.

Finalmente, se puede ver de la misma manera la operación de saltar si igual, por ejemplo, `beq $t1, $t2, offset`. Su ejecución es similar a las instrucciones tipo R, pero la salida de la ALU se utiliza para determinar si el PC se actualizará con PC + 4 o con la dirección destino del salto. La figura 4.21 muestra los cuatro pasos de la ejecución.

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.

2. Se leen los registros `$t1` y `$t2` del banco de registros.

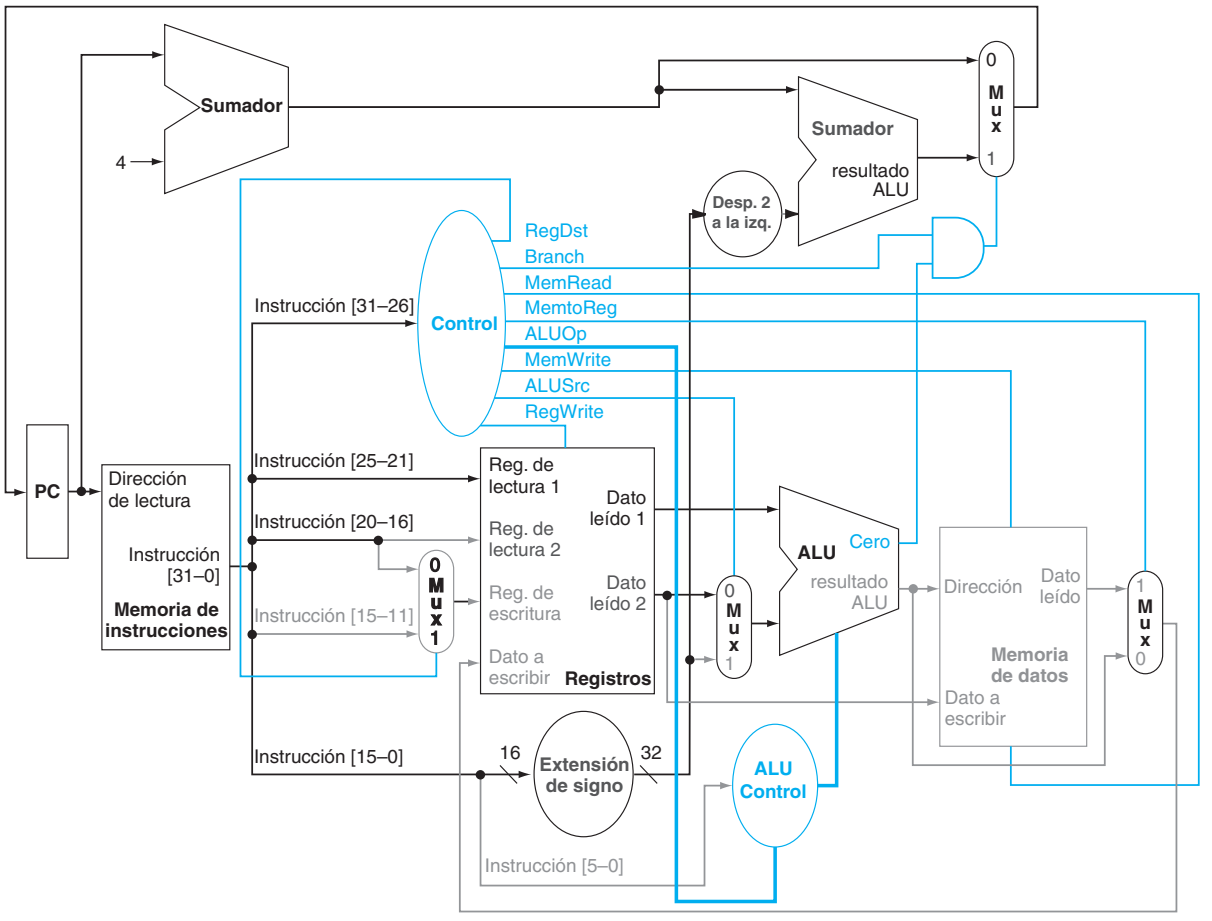


FIGURA 4.21 Funcionamiento del camino de datos para una instrucción de salto si igual. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Después de usar el banco de registros y la ALU para realizar la comparación, la salida Cero se utiliza para seleccionar el siguiente valor del contador de programas entre los dos valores candidatos.

3. La ALU realiza una resta de los operandos leídos del banco de registros. Se suma el valor de PC + 4 a los 16 bits de menor peso (con el signo extendido) de la instrucción (`offset`) desplazados 2 bits. El resultado es la dirección destino del salto.

4. Se utiliza la señal Cero de la ALU para decidir qué valor se almacena en el PC.

Finalización del control

Una vez que se ha visto los pasos en la ejecución de las instrucciones continuaremos con la implementación del control. La función de control puede definirse de forma precisa utilizando los contenidos de la figura 4.18. Las salidas son las líneas de control y las entradas los 6 bits que conforman el campo del código de operación (Op[5-0]). De esta manera se puede crear una tabla de verdad para cada una de las salidas basada en la codificación binaria de los códigos de operación.

La figura 4.22 muestra la lógica de la unidad de control como una gran tabla de verdad que combina todas las salidas y utiliza los bits del código de operación como entradas. Ésta especifica de forma completa la función de control y puede implementarse mediante puertas lógicas de forma automática. Este paso final se muestra en la sección D.2 en el **apéndice D**.

Ahora que ya tenemos una **implementación de ciclo de reloj único** para la mayor parte del núcleo del repertorio de instrucciones MIPS, se va a añadir la instrucción de salto incondicional (*jump*) y se verá cómo puede extenderse el camino de datos y su control para poder ejecutar otro tipo de instrucciones del repertorio.

Implementación de ciclo único (implementación de ciclo de reloj único): implementación en la que una instrucción se ejecuta en un único ciclo de reloj.

Entrada o salida	Nombre de la señal	Formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Salidas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURA 4.22 La función de control para una implementación de ciclo único está completamente especificada en esta tabla de verdad. La mitad superior de la tabla muestra las posibles combinaciones de las señales a la entrada, que corresponden a los cuatro posibles códigos de operación y que determinan la activación de las señales de control. Recuerde que Op[5-01] se corresponde con los bits 31-26 de la instrucción (el código de operación). La parte inferior muestra las salidas. Así, la señal de salida RegWrite está activada para dos combinaciones diferentes de entradas. Si únicamente se consideran estos cuatro posibles códigos de operación, esta tabla puede simplificarse utilizando términos indeterminados en las señales de entrada. Por ejemplo, puede detectarse una instrucción de tipo aritmético-lógico con la expresión $\overline{Op5} \cdot \overline{Op2}$, ya que es suficiente para distinguirlas del resto. De todas formas, no se utilizará esta simplificación, pues el resto de los códigos de operación del MIPS se usarán en la implementación completa.

Implementación de saltos incondicionales (*jump*)

EJEMPLO

La figura 4.17 muestra la implementación de muchas de las instrucciones vistas en el capítulo 2. Un tipo de instrucción ausente es el salto incondicional. Extienda el camino de datos de la figura 4.17, así como su control, para incluir dicho tipo de instrucciones. Describa cómo se ha de activar cualquier línea de control nueva.

RESPUESTA

La instrucción *jump*, mostrada en la figura 4.23, tiene un cierto parecido a la instrucción de salto condicional, pero calcula la dirección de destino de forma diferente y, además, es incondicional. Como en los saltos condicionales, los 2 bits de menor peso de la dirección de salto son siempre 00_{dos}. Los siguientes 26 bits de menor peso de la dirección están en el campo inmediato de la instrucción. Los 4 bits de mayor peso de la dirección que debería reemplazar al PC vienen del PC de la instrucción al cual se le ha sumado 4. Así, podría realizarse un salto incondicional almacenando en el registro de PC la concatenación de:

- Los 4 bits de mayor peso del actual PC + 4 (son los bits 31-28 de la dirección de la siguiente instrucción en orden secuencial).
- Los 26 bits correspondientes al campo inmediato de la instrucción *jump*.
- Los bits 00_{dos}.

La figura 4.24 muestra las partes añadidas al control para este tipo de instrucciones respecto a la figura 4.17. Se necesita un nuevo multiplexor para seleccionar el origen del nuevo PC, la dirección de 1a siguiente instrucción en orden secuencial (PC + 4), la dirección de salto condicional o la de una instrucción de salto incondicional. También se necesita una nueva señal de control para este multiplexor. Esta señal, llamada *Jump*, únicamente se activa cuando la instrucción es un salto incondicional, es decir, cuando el código de operación es 2.

Campo	000010	dirección
Posición de los bits	31-26	25-0

FIGURA 4.23 Formato de la instrucción *jump* (código de operación = 2). La dirección destino de este tipo de instrucciones se forma mediante la concatenación de los 4 bits de mayor peso de PC + 4 y los 26 bits de campo de dirección de la instrucción añadiendo 00 como los 2 bits de menor peso.

Por qué no se utiliza una implementación de ciclo único hoy en día

Aunque este tipo de implementaciones funciona correctamente, no se utiliza en los diseños actuales porque es ineficiente. Para ver por qué ocurre esto, debe saberse que el ciclo de reloj debe tener la misma longitud para todos los tipos de instrucciones. Por supuesto, el ciclo de reloj viene determinado por el mayor

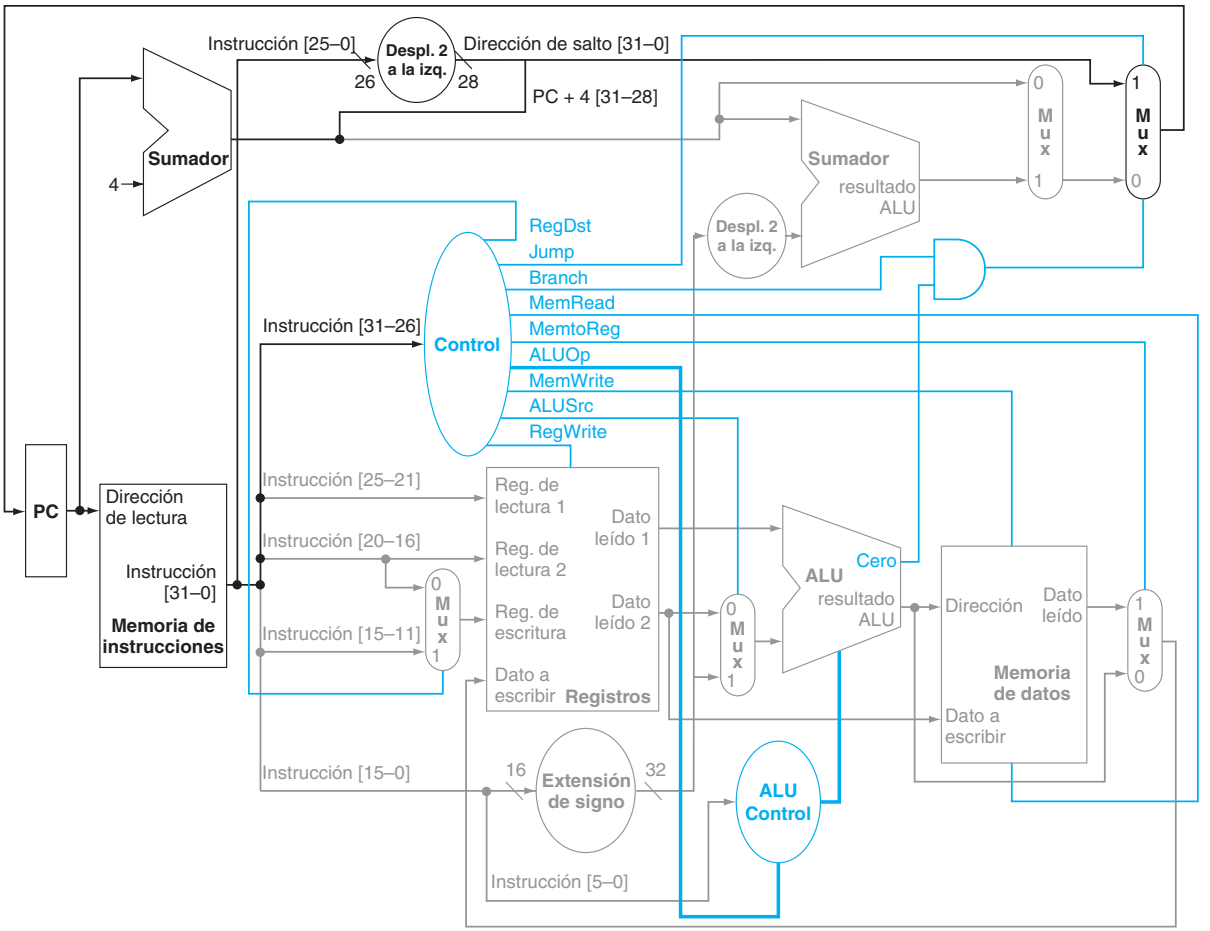


FIGURA 4.24 Un camino de datos sencillo y su control extendido para ejecutar instrucciones de salto incondicional. Se utiliza un multiplexor adicional (en la esquina superior derecha) para elegir entre el destino de la instrucción de salto incondicional y la dirección salto condicional o la siguiente en orden secuencial. Este multiplexor se controla mediante la señal de salto incondicional. La dirección de destino de la instrucción de salto incondicional se obtiene desplazando los 26 bits de menor peso 2 bits a la izquierda (se añaden 00 como bits de menor peso), concatenando los 4 bits de mayor peso de PC + 4 como bits de mayor peso, y obteniendo de esta manera una dirección de 32 bits.

tiempo de ejecución posible en el procesador. Esta instrucción en la mayoría de los casos es la instrucción *load*, que usa 5 unidades funcionales en serie: la memoria de instrucciones, el banco de registros, la ALU, la memoria de datos y el banco de registros nuevamente. Aunque el CPI es 1 (véase el capítulo 1), las prestaciones generales de la implementación de ciclo único probablemente no es muy bueno, porque el ciclo de reloj es demasiado largo.

La penalización por utilizar un diseño de ciclo único con un ciclo de reloj fijo es importante, pero podría considerarse aceptable para un repertorio con pocas instrucciones. Históricamente, los primeros computadores con un repertorio de instrucciones muy simple usaban esta técnica de implementación. Sin embargo, si

se intenta implementar una unidad de punto flotante o un repertorio de instrucciones con instrucciones más complejas, el diseño de ciclo único podría no funcionar correctamente.

Es inútil intentar implementar técnicas que reduzcan el retraso del caso más habitual pero que no mejoren el tiempo de ciclo del peor caso, porque el ciclo de la señal de reloj debe ser igual al retraso del peor caso de todas las instrucciones. Así, la implementación de ciclo único viola el principio de diseño clave del capítulo 2, de hacer rápido el caso más habitual.

En la siguiente sección analizaremos otra técnica de implementación, llamada segmentación, que utiliza un camino de datos muy similar a la de ciclo único pero mucho más eficiente, con una productividad más elevada. La segmentación mejora la eficiencia mediante la ejecución de varias instrucciones simultáneamente.

Autoevaluación

Fíjese en el control de la figura 4.22. ¿Se pueden combinar algunas señales de control? ¿Se puede reemplazar alguna señal de control de salida por la inversa de otra? (ténganse en cuenta las indeterminaciones). Si es así, ¿se puede reemplazar una señal por otra sin añadir inversores?

No malgastes el tiempo.
Proverbio Americano

4.5 Descripción general de la segmentación

Segmentación (*pipelining*): técnica de implementación que solapa la ejecución de múltiples instrucciones, de forma muy similar a una línea de ensamblaje.

Segmentación (*pipelining*) es una técnica de implementación que consiste en solapar la ejecución de múltiples instrucciones. Hoy en día, la segmentación es universal.

En esta sección utilizaremos una analogía para describir los términos básicos y las características principales de la segmentación. Si el lector sólo está interesado en la idea general, debe centrarse en esta sección y después saltar a las secciones 4.10 y 4.11 para leer una introducción a las técnicas avanzadas de segmentación que usan procesadores recientes como el AMD Opteron X4 (Barcelona) o Intel Core. Pero si está interesado en explorar la anatomía de un computador segmentado, entonces esta sección es una buena introducción a las secciones de la 4.6 a la 4.9.

Cualquiera que haya tenido que lavar grandes cantidades de ropa ha usado de forma intuitiva la estrategia de la segmentación. El enfoque no segmentado de hacer la colada sería

1. Introducir ropa sucia en la lavadora.
2. Cuando finaliza el lavado, introducir la ropa mojada en la secadora.
3. Cuando finaliza el secado, poner la ropa seca en una mesa para ordernarla y doblarla.
4. Una vez que toda la ropa está doblada, pedir al compañero de piso que guarde la ropa.

Cuando el compañero ha finalizado, entonces se vuelve a comenzar con la siguiente colada.

El enfoque segmentado de lavado require mucho menos tiempo, tal y como muestra la figura 4.25. Tan pronto como la lavadora termina con la primera colada y ésta es colocada en la secadora, se vuelve a cargar la lavadora con una

segunda colada de ropa sucia. Cuando la primera colada esté seca, se pone encima de la mesa para empezar a doblarla, se pasa la colada mojada de la lavadora a la secadora, y se mete en la lavadora la tercera colada de ropa sucia. Después, mientras el compañero de piso se lleva la ropa doblada, se empieza a doblar la segunda colada al tiempo que la tercera colada se pasa de la lavadora a la secadora y se introduce la cuarta colada en la lavadora. Llegados a este punto todos los pasos –denominados *etapas* de segmentación (o segmentos)– se llevan a cabo de forma concurrente. Se podrán segmentar las tareas siempre y cuando se disponga de recursos separados para cada etapa.

La paradoja de la segmentación es que el tiempo desde que se pone un calcetín sucio en la lavadora hasta que se seca, se dobla y se guarda no es más corto al utilizar la segmentación; la razón por la cual la segmentación es más rápida para varias coladas es que todas las etapas se llevan a cabo en paralelo, y por tanto se completan más coladas por hora. La segmentación mejora la productividad de la lavandería sin mejorar el tiempo necesario para completar una única colada. Consecuentemente, la segmentación no reducirá el tiempo para completar una colada, pero si tenemos que hacer muchas coladas, la mejora de la productividad reducirá el tiempo total para completar todo el trabajo.

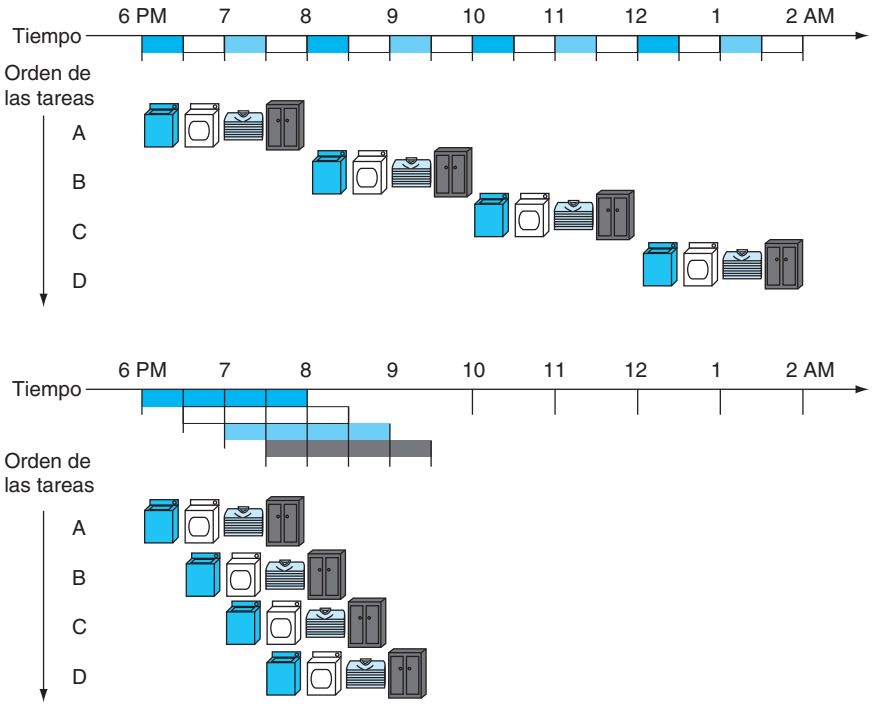


FIGURA 4.25 La analogía de la lavandería. Ann, Brian, Cathy y Don tienen ropa sucia que lavar, secar, doblar y guardar. Cada una de las cuatro tareas (lavar, secar, doblar y guardar) dura 30 minutos. Hacer la colada secuencialmente lleva 8 horas para realizar 4 coladas, mientras que la colada organizada de forma segmentada necesita solamente 3.5 horas. Aunque en realidad se dispone de un único recurso de cada tipo, para mostrar la etapa de segmentación usada por las diferentes coladas a lo largo del tiempo se usan copias de los 4 recursos.

Si todos los segmentos o etapas requieren la misma cantidad de tiempo y si hay el suficiente trabajo por hacer, entonces la ganancia que proporciona la segmentación es igual al número de segmentos en que se divide la tarea, que en este caso son cuatro: lavar, secar, doblar y colocar. Por tanto, el lavado segmentado es potencialmente cuatro veces más rápido que el no segmentado: 20 coladas necesitarán alrededor de 5 veces más tiempo que una única colada, mientras que la estrategia secuencial de lavado requiere 20 veces el tiempo de una colada. En la figura 4.25 la segmentación sólo es 2.3 veces más rápida porque se consideran únicamente 4 coladas. Observe que tanto al principio como al final de la tarea segmentada, el cauce de segmentación (que denominaremos *pipeline*) no está completamente lleno. Estos efectos transitorios iniciales y finales afectan a las prestaciones cuando el número de tareas es pequeño en comparación con el número de etapas. Cuando el número de coladas es mucho mayor que 4, las etapas estarán ocupadas la mayor parte del tiempo y el incremento en la productividad será muy cercano a 4.

Los mismos principios son aplicables a los procesadores cuando se segmenta la ejecución de instrucciones. Clásicamente se consideran cinco pasos para ejecutar las instrucciones MIPS:

1. Buscar una instrucción en memoria.
2. Leer los registros mientras se descodifica la instrucción. El formato de las instrucciones MIPS permite que la lectura y descodificación ocurran de forma simultánea.
3. Ejecutar una operación o calcular una dirección de memoria.
4. Acceder a un operando en la memoria de datos.
5. Escribir el resultado en un registro.

Por tanto, el pipeline MIPS que se explorará en este capítulo tiene cinco etapas. El siguiente ejemplo muestra que la segmentación acelera la ejecución de instrucciones del mismo modo que lo hace para la colada.

EJEMPLO

Las prestaciones en un diseño de ciclo único (monociclo) frente a un diseño segmentado

Para concretar la discusión se creará un pipeline. En este ejemplo y en el resto del capítulo limitaremos nuestra atención a sólo 8 instrucciones: cargar palabra (`lw`), almacenar palabra (`sw`), sumar (`add`), restar (`sub`), y-lógica (`and`), o-lógica (`or`), activar si menor (*set-less-than*, `slt`), y saltar si es igual (`beq`).

Compare el tiempo promedio entre la finalización de instrucciones consecutivas en una implementación monociclo, en la que todas las instrucciones se ejecutan en un único ciclo de reloj, con una implementación segmentada. En este ejemplo, la duración de las operaciones para las unidades funcionales principales es de 200 ps para los accesos a memoria, de 200 ps para las operaciones en la ALU, y de 100 ps para las lecturas o las escrituras en el banco de registros. En el modelo monociclo cada instrucción dura exactamente un ciclo de reloj, y por tanto el ciclo de reloj debe estirarse lo suficiente para poder acomodar a la instrucción más lenta.

RESPUESTA

La figura 4.26 indica el tiempo requerido para cada una de las ocho instrucciones. El diseño monociclo debe permitir acomodar a la instrucción más lenta —que en la figura 4.26 es `lw`— y por tanto la duración de todas las instrucciones será de 800 ps. Del mismo modo que la figura 4.25, la figura 4.27 compara las ejecuciones segmentadas y no segmentadas de tres instrucciones `lw`. En el caso del diseño no segmentado, el tiempo transcurrido entre el inicio de la primera instrucción y el inicio de la cuarta instrucción es de 3 × 800 ps ó 2400 ps.

Todas las etapas de segmentación duran un único ciclo de reloj, de modo que el periodo del reloj debe ser suficientemente largo como para dar cabida a la operación más lenta. Así, del mismo modo que el diseño monociclo debe tener un ciclo de reloj de 800 ps, necesario para el caso más desfavorable, aun cuando algunas instrucciones podrían ser ejecutadas en 500 ps, la ejecución segmentada debe tener un ciclo de reloj de 200 ps para dar cabida a la etapa más desfavorable, aunque algunas etapas sólo necesiten 100 ps. Aún así, la segmentación cuadriplica las prestaciones: el tiempo entre el inicio de la primera instrucción y el inicio de la cuarta instrucción es de 3 × 200 ps o 600 ps.

Clase de instrucción	Búsqueda de la instrucción	Lectura de registros	Operación ALU	Acceso al dato	Escritura en registro	Tiempo total
Almacenar palabra (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Cargar palabra (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Salto (`beq`)	200 ps	100 ps	200 ps			500 ps

FIGURA 4.26 Tiempo total para cada instrucción calculado a partir del tiempo de cada componente. El cálculo supone que no existe ningún retardo debido a los multiplexores, a la unidad de control, a los accesos al registro PC o a la unidad de extensión de signo.

Es posible convertir en una fórmula la discusión anterior sobre la ganancia de la segmentación. Si las etapas están perfectamente equilibradas, entonces el tiempo entre instrucciones en el procesador segmentado –suponiendo condiciones ideales– es igual a

$$\text{Tiempo entre instrucciones}_{\text{segmentado}} = \frac{\text{Tiempo entre instrucciones}_{\text{no segmentado}}}{\text{Número de etapas de segmentación}}$$

En condiciones ideales y con un gran número de instrucciones, la ganancia debida a la segmentación es aproximadamente igual al número de etapas; un pipeline de cinco etapas es casi cinco veces más rápido.

La fórmula sugiere que con cinco etapas se debe mejorar en cinco veces el tiempo de 800 ps que proporciona el esquema no segmentado, es decir, lograr un ciclo de reloj de 160 ps. Sin embargo, el ejemplo muestra que las etapas pueden estar equilibradas de manera imperfecta. Además, la segmentación implica algún gasto o sobrecarga adicional, cuya fuente será presentada con más claridad en breve. Así, el tiempo por instrucción en el procesador segmentado excederá el mínimo valor posible, y la ganancia será menor que el número de etapas de segmentación.

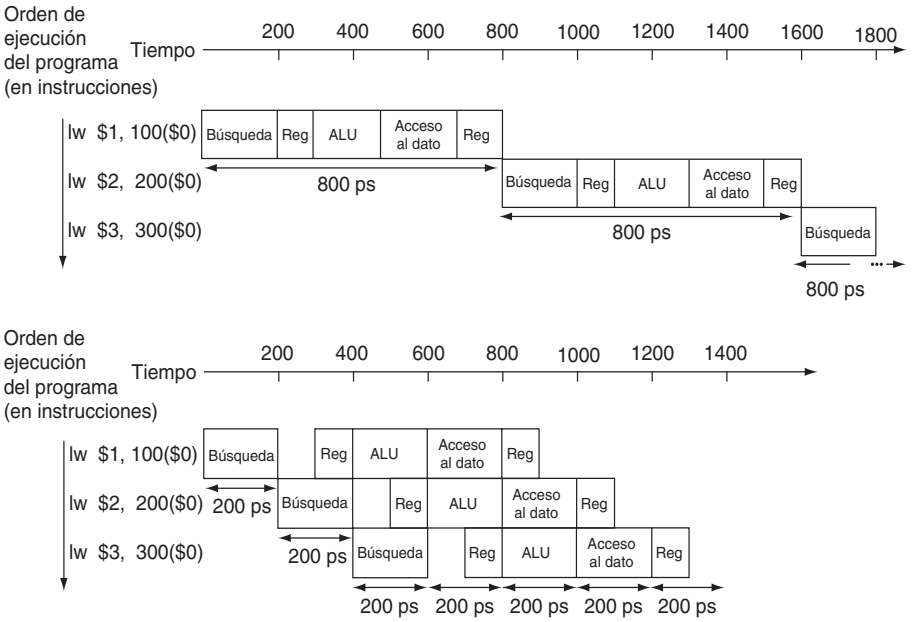


FIGURA 4.27 Arriba: ejecución monociclo sin segmentar; abajo: ejecución segmentada. Ambos esquemas usan los mismos componentes hardware, cuyo tiempo se lista en la figura 4.26. En este caso se aprecia una ganancia de 4 en el tiempo promedio entre instrucciones, desde 800 ps hasta 200 ps. Comparar esta figura con la figura 4.25. En el caso de la lavandería, se suponía que todas las etapas eran iguales. Si la secadora hubiera sido el elemento más lento, entonces hubiera determinado el tiempo de cada etapa. Los segmentos del computador están limitados a operar a la velocidad del más lento, bien sea la operación de la ALU o el acceso a memoria. Se supondrá que la escritura en el banco de registros ocurre durante la primera mitad del ciclo de reloj mientras que las lecturas del banco de registros ocurren durante la segunda mitad. Se usará esta suposición a lo largo de todo el capítulo.

Incluso nuestra afirmación de que en el ejemplo se mejoran cuatro veces los resultados no queda reflejada en el tiempo total de ejecución para las tres instrucciones: 1400 ps frente a 2400 ps. Por supuesto, esto es debido a que el número de instrucciones analizado es muy pequeño. ¿Qué ocurriría si se incrementara el número de instrucciones? Podemos extender los números del ejemplo previo a 1.000.003 instrucciones. Añadiendo 1.000.000 instrucciones en el ejemplo segmentado se suman 200 ps por instrucción al tiempo total de ejecución. El tiempo total de ejecución sería de 1.000.000 × 200 ps + 1400 ps, ó 200.001.400 ps. En el ejemplo no segmentado, añadiríamos 1.000.000 instrucciones, cada una con una duración de 800 ps, de modo que el tiempo total de ejecución sería 1.000.000 × 800 ps + 2400 ps, ó 800.002.400 ps. En estas condiciones ideales, la razón entre los tiempos totales de ejecución para programas reales en procesadores no segmentados comparados con los tiempos en procesadores segmentados es cercana a la razón de los tiempos entre instrucciones:

$$\frac{800.002.400 \text{ ps}}{200.001.400 \text{ ps}} \approx \frac{800 \text{ ps}}{200 \text{ ps}} \approx 4{,}00$$

La segmentación mejora las prestaciones *incrementando la productividad (throughput) de las instrucciones, en lugar de disminuir el tiempo de ejecución de cada instrucción individual,* pero la productividad de las instrucciones es la métrica importante, ya que los programas reales ejecutan miles de millones de instrucciones.

Diseño de repertorios de instrucciones para la segmentación

Incluso con esta explicación sencilla de la segmentación es posible comprender el diseño del repertorio de instrucciones del MIPS, diseñado expresamente para la ejecución segmentada.

En primer lugar, todas las instrucciones MIPS tienen la misma longitud. Esta restricción hace que sea mucho más fácil la búsqueda de instrucciones en la primera etapa de la segmentación y la descodificación en la segunda etapa. En un repertorio de instrucciones como el x86, donde las instrucciones varían de 1 byte a 17 bytes, la segmentación es un reto considerable. Las implementaciones recientes de la arquitectura x86 en realidad convierten las instrucciones x86 en microoperaciones simples muy parecidas a instrucciones MIPS. Y se segmenta la ejecución de las microoperaciones en lugar de las instrucciones x86 nativas (véase sección 4.10).

Segundo, MIPS tiene sólo unos pocos formatos de instrucciones, y además en todos ellos los campos de los registros fuentes están situados siempre en la misma posición de la instrucción. Esta simetría implica que la segunda etapa pueda empezar a leer del banco de registros al mismo tiempo que el hardware está determinando el tipo de la instrucción que se ha leído. Si los formatos de instrucciones del MIPS no fueran simétricos, sería necesario partir la segunda etapa, dando como resultado un pipeline de seis etapas. En breve se indicarán la desventaja de los pipelines más largos.

En tercer lugar, en el MIPS los operandos a memoria sólo aparecen en instrucciones de carga y almacenamiento. Esta restricción hace que se pueda usar la etapa de ejecución para calcular la dirección de memoria y en la siguiente etapa se pueda acceder a memoria. Si se permitiera usar operandos en memoria en todas las operaciones, como hace la arquitectura x86, las etapas 3 y 4 se extenderían a una etapa de cálculo de dirección, una etapa de acceso a memoria, y luego una etapa de ejecución.

Y cuarto, como se discutió en el capítulo 2, los operandos deben estar alineados en memoria. Por lo tanto, no hay que preocuparse de que una instrucción de transferencia de datos necesite dos accesos a memoria para acceder a un solo dato; el dato pedido siempre podrá ser transferido entre procesador y memoria en una única etapa.

Riesgos del pipeline

Hay situaciones de segmentación en las que la instrucción siguiente no se puede ejecutar en el ciclo siguiente. Estos sucesos se llaman riesgos (*hazards*) y los hay de tres tipos diferentes

Riesgo estructural: caso en el que una instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque el hardware no admite la combinación de instrucciones que se pretenden ejecutar en ese ciclo de reloj.

Riesgos estructurales

El primer riesgo se denomina **riesgo estructural (*structural hazard*)**. Significa que el hardware no admite una cierta combinación de instrucciones durante el mismo ciclo. Un riesgo estructural en la lavandería ocurriría si se usara una combinación lavadora-secadora en lugar de tener la lavadora y la secadora separadas,

o si nuestro compañero de piso estuviera ocupado y no pudiera guardar la ropa. De esta forma, se frustraría la tan cuidadosamente planeada segmentación.

Como se ha mencionado con anterioridad, el repertorio de instrucciones MIPS fue diseñado para ser segmentado, por lo que facilita a los diseñadores evitar riesgos estructurales. Supongamos, sin embargo, que se tuviera un solo banco de memoria en lugar de dos. Si el pipeline de la figura 4.27 tuviera una cuarta instrucción, se vería que en un mismo ciclo la primera instrucción está accediendo a datos de memoria mientras que la cuarta está buscando una instrucción de la misma memoria. Sin disponer de dos bancos de memoria, nuestra segmentación podría tener riesgos estructurales.

Riesgos de datos

Riesgo de datos: (riesgo de datos en el pipeline): caso en el que la instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque el dato necesario para ejecutar la instrucción no está todavía disponible

Los **riesgos de datos (*data hazards*)**, ocurren cuando el pipeline se debe bloquear debido a que un paso de ejecución debe esperar a que otro paso sea completado. Volviendo a la lavandería, supongamos que se está doblando una colada y no se encuentra la pareja de un cierto calcetín. Una estrategia posible es ir al piso a buscar en todos los armarios hasta encontrar la pareja. Obviamente, mientras se busca deberán esperar todas las coladas secas y preparadas para ser dobladas, y todas las coladas limpias y preparadas para ser secadas.

En un pipeline del computador, los riesgos de datos surgen de las dependencias entre una instrucción y otra anterior que se encuentra todavía en el pipeline (una relación que en realidad no se da en la lavandería). Por ejemplo, suponer que se tiene una instrucción `add` seguida inmediatamente por una instrucción `sub` que usa el resultado de la suma (`$s0`):

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Si no se interviene, un riesgo de datos puede bloquear severamente al procesador. La instrucción `add` no escribe su resultado hasta la quinta etapa, por lo que se tendrían que añadir tres burbujas en el procesador.

Anticipación de resultados (realimentación): método de resolver un riesgo de datos que consiste en obtener el dato buscado de los búfer internos en lugar de esperar a que llegue a los registros visibles a nivel de programación o a la memoria.

Aunque se podría confiar que los compiladores eliminaran todos esos riesgos, los resultados no serían satisfactorios. Estas dependencias ocurren demasiado a menudo y el retardo es demasiado largo para esperar que el compilador nos salve de este problema.

La principal solución se basa en la observación de que no es necesario esperar a que la instrucción se complete antes de intentar resolver el riesgo de datos. Para la secuencia de código anterior, tan pronto como la ALU calcula la suma para la instrucción `add`, se puede suministrar el resultado como entrada de la resta. Se denomina **anticipación de resultados (*forwarding*)** o **realimentación (*bypassing*)** al uso de hardware extra para anticipar antes el dato buscado usando los recursos internos del procesador.

EJEMPLO

Anticipar datos entre dos instrucciones

Para las dos instrucciones anteriores, mostrar qué etapas del pipeline estarían conectadas por el mecanismo de anticipación de resultados. Usar el dibujo de la figura 4.28 para representar el camino de datos durante las cinco etapas del pipeline. Alinear una copia del camino de datos para cada instrucción, de forma similar a como se hizo en la figura 4.25 para el pipeline de la lavandería.

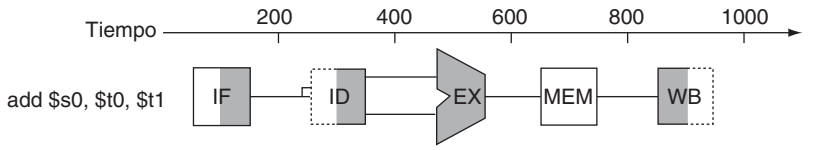


FIGURA 4.28 Representación gráfica del pipeline de instrucciones, similar en espíritu a la segmentación de la lavandería de la figura 4.25. Para representar los recursos físicos se usan símbolos con las abreviaciones de las etapas del pipeline usadas a lo largo del capítulo. Los símbolos para las cinco etapas son: *IF* para la etapa de búsqueda de instrucciones (*instruction fetch*), con la caja que representa la memoria de instrucciones; *ID* para la etapa de descodificación de instrucciones y de lectura del banco de registros (*instruction decode*), con el dibujo que muestra la lectura al banco de registros; *EX* para la etapa de ejecución de instrucciones (*instruction execution*), con el dibujo que representa la ALU; *MEM* para la etapa de acceso a memoria (*memory access*), con la caja que representa la memoria de datos; y *WB* para la etapa de escritura de resultado (*write back*), con el dibujo que muestra la escritura en el banco de registros. El sombreado del dibujo indica el elemento que se usa por parte de la instrucción. De este modo, MEM tiene un fondo blanco porque la instrucción `add` no accede a la memoria de datos. El sombreado en la mitad derecha del banco de registros o de la memoria significa que el elemento se lee en esa etapa, y el sombreado en la mitad izquierda del banco de registros significa que se escribe en esa etapa. Por tanto, la mitad derecha de ID está sombreada en la segunda etapa porque se lee el banco de registros, y la mitad izquierda de WB está sombreada en la quinta etapa porque se escribe en el banco de registros.

RESPUESTA

La figura 4.29 muestra la conexión para anticipar el valor de `$s0` después de la etapa de ejecución de la instrucción `add` a la entrada de la etapa de ejecución de la instrucción `sub`.

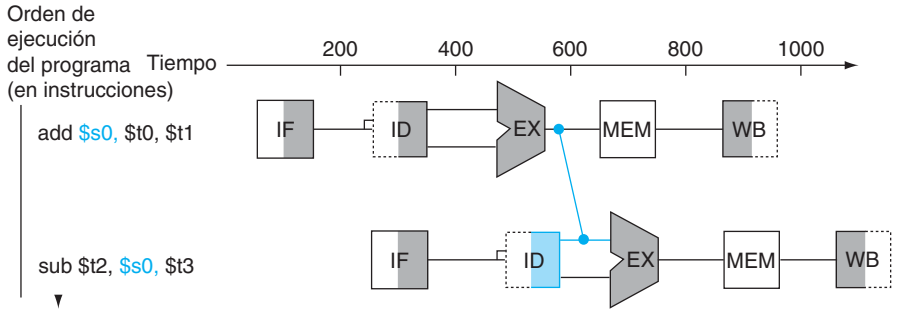


FIGURA 4.29 Representation gráfica de la anticipación de datos. La conexión muestra el camino de anticipación de datos desde la salida de la etapa EX de `add` hasta la entrada de la etapa EX de `sub`, reemplazando el valor del registro `$s0` leído en la segunda etapa de la instrucción `sub`.

En esta representación gráfica de los sucesos, las líneas de anticipación de datos sólo son válidas si la etapa destino está más adelante en el tiempo que la etapa origen. Por ejemplo, no puede establecerse una línea válida de anticipación desde la salida de la etapa de acceso al dato de la primera instrucción hasta la entrada de la etapa de ejecución de la siguiente, puesto que eso significaría ir hacia atrás en el tiempo.

La anticipación funciona muy bien y será descrita en detalle en la sección 4.7. De todos modos, no es capaz de evitar todos los bloqueos. Por ejemplo, suponer que la primera instrucción fuera una carga del registro `$s0`, en lugar de una suma. Como se puede deducir a partir de una mirada a la figura 4.29, el dato deseado sólo estaría dis-

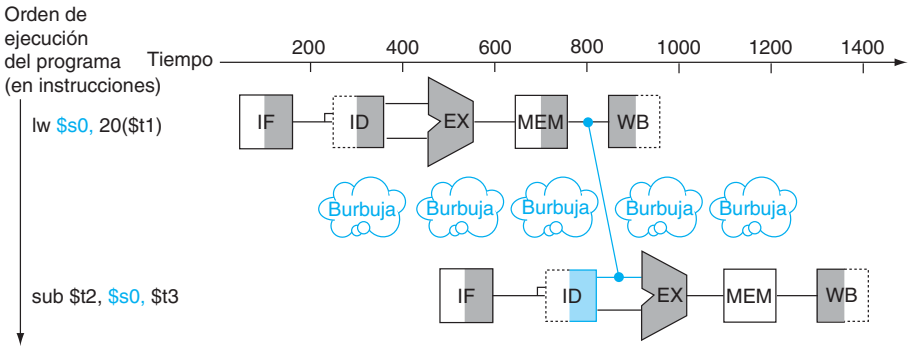


FIGURA 4.30 El bloqueo es necesario incluso con la anticipación de resultados cuando una instrucción de Formato R intenta usar el dato de una instrucción de carga precedente. Sin el bloqueo, la ruta desde la salida de la etapa de acceso a memoria hasta la entrada de la etapa de ejecución sería hacia atrás en el tiempo, lo cual es imposible. Esta figura es en realidad una simplificación, pues hasta después de que la instrucción de resta es buscada y descodificada no se puede conocer si será o no será necesario un bloqueo. La sección 4.7 muestra los detalles de lo que sucede en realidad en el caso de los riesgos.

Riesgo del dato de una carga: forma específica de riesgo en la que el dato de una instrucción de carga no está aún disponible cuando es pedido por otra instrucción.

Bloqueo del pipeline (burbuja): bloqueo iniciado para resolver un riesgo.

ponible *después* de la cuarta etapa de la primera instrucción, lo cual es demasiado tarde para la *entrada* de la etapa EX de la instrucción `sub`. Por lo tanto, incluso con la anticipación de resultados, habría que bloquear durante una etapa en el caso del **riesgo del dato de una carga** (***load-use data hazard***), tal como se puede ver en la figura 4.30. Esta figura muestra un importante concepto sobre la segmentación, oficialmente denominado un **bloqueo del pipeline** (***pipeline stall***), pero que frecuentemente recibe el apodo de **burbuja** (*bubble*). Se verán bloqueos en otros lugares del pipeline. La sección 4.7 muestra cómo se pueden tratar casos difíciles como éstos, bien con un hardware de detección y con bloqueos, bien por software, que reordena el código para evitar los bloqueos debido a riesgos de datos de una carga (*load-use*), como se ilustra en el siguiente ejemplo.

EJEMPLO

Reordenación de código para evitar bloqueos del pipeline

Considere el siguiente segmento de código en lenguaje C:

```
a = b + e;
c = b + f;
```

Aquí se muestra el código MIPS generado para este segmento, suponiendo que todas las variables están en memoria y son direccionables como desplazamientos a partir de `$t0`:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($01)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

Encuentre los riesgos que existen en el segmento de código y reordene las instrucciones para evitar todos los bloqueos del pipeline.

RESPUESTA

Las dos instrucciones `add` tienen un riesgo debido a sus respectivas dependencias con la instrucción `lw` que les precede inmediatamente. Observe que la anticipación de datos elimina muchos otros riesgos potenciales, incluidos la dependencia del primer `add` con el primer `lw` y los riesgos con las instrucciones *store*. Mover hacia arriba la tercera instrucción `lw` elimina ambos riesgos.

```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($01)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

En un procesador segmentado con anticipación de resultados, la secuencia reordenada se completará en dos ciclos menos que la versión original.

La anticipación de resultados conlleva otra característica que hace comprender la arquitectura MIPS, además de las cuatro mencionadas en la página 335. Cada instrucción MIPS escribe como mucho un resultado y lo hace cerca del final del pipeline. La anticipación de resultados es más complicada si hay múltiples resultados que avanzar por cada instrucción, o si se necesita escribir el resultado antes del final de la ejecución de la instrucción.

Extensión: El nombre "anticipación de resultado" (*forwarding*) viene de la idea de que el resultado se pasa hacia adelante, de una instrucción anterior a una instrucción posterior. El término "realimentación" (*bypassing*) viene del hecho de pasar el resultado a la unidad funcional deseada sin tener que copiarlo previamente en el banco de registros.

Riesgos de control

El tercer tipo de riesgo se llama **riesgo de control (*control hazard*)** y surge de la necesidad de tomar una decisión basada en los resultados de una instrucción mientras las otras se están ejecutando.

Supongamos que los trabajadores de nuestra lavandería tienen la feliz tarea de lavar los uniformes de un equipo de fútbol. Dependiendo de lo sucia que esté la colada, se necesitará determinar si el detergente y la temperatura del agua

Riesgo de control (riesgo de saltos): caso en el que la instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque la instrucción que ha sido buscada no es la que se requería; esto es, el flujo de direcciones de instrucciones no es el que el pipeline esperaba.

seleccionadas son suficientemente fuertes para lavar los uniformes, pero no tan fuertes como para que éstos se desgasten y se rompan pronto. En nuestra lavandería segmentada, se tiene que esperar a la segunda etapa de la segmentación para examinar el uniforme seco y comprobar si se tiene que cambiar algunos de los parámetros del lavado. ¿Qué se debe hacer?

A continuación se presenta la primera de dos soluciones posibles a los riesgos de control en la lavandería y sus equivalentes para los computadores.

Bloquear (*stall*): Operar de manera secuencial hasta que la primera carga esté seca, y entonces repetir hasta que se consiga la fórmula correcta.

Esta opción conservadora realmente funciona, pero es lenta.

En un computador, la tarea de decisión equivalente es la instrucción de salto (*branch*). Observe que se debe comenzar a buscar la instrucción que sigue a un salto justo en el siguiente ciclo de reloj. Pero el pipeline puede no conocer cuál es la siguiente instrucción, ya que ¡*justo acaba de obtener* la instrucción de salto de la memoria! Igual que con la lavandería, una posible solución es bloquear inmediatamente después de haber ido a buscar un salto, y esperar a que el pipeline determine el resultado del salto y conozca cuál es la instrucción que se debe ir a buscar.

Supongamos que se emplea suficiente hardware adicional para poder examinar los registros, calcular la dirección del salto y actualizar el registro PC durante la segunda etapa del pipeline (véase la sección 4.8 para más detalles). Incluso con este hardware adicional, el pipeline que trata los saltos condicionales se parecería al de la figura 4.31. La instrucción `lw`, ejecutada si el salto no se toma, se bloquea un ciclo extra de 200 ps antes de empezar.

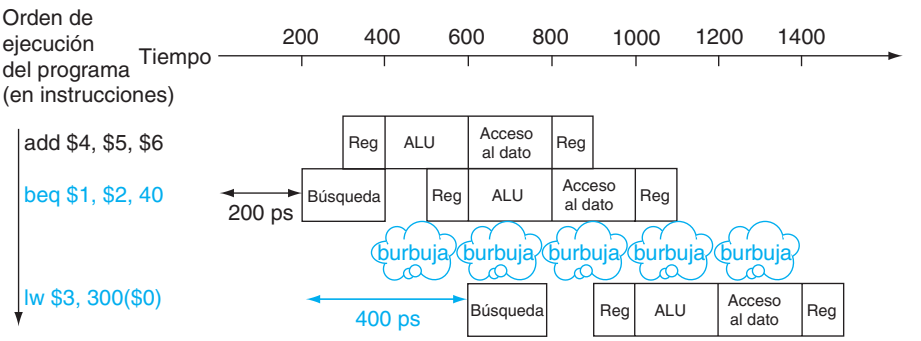


FIGURA 4.31 Pipeline que muestra el bloqueo en cada salto condicional como solución a los riesgos de control. En este ejemplo se supone que el salto condicional es tomado y que la instrucción destino del salto es la instrucción "`OR`". Se produce un bloqueo en el pipeline de una etapa, o burbuja, después del salto. En realidad, el proceso de crear el bloqueo es ligeramente más complicado, tal y como se verá en la sección 4.8. El efecto en las prestaciones, en cambio, sí que es el mismo que se daría si realmente se insertara una burbuja.

Rendimiento de "bloquear los saltos"

EJEMPLO

Estimar el impacto en ciclos de reloj por instrucción (CPI) de bloqueo en los saltos. Suponer que todas las otras instrucciones tienen un CPI de 1.

RESPUESTA

La figura 3.27 del capítulo 3 muestra que los saltos condicionales representan el 17% de las instrucciones ejecutadas en SPECint2006. Dado que la ejecución del resto de las instrucciones tiene un CPI de 1 y los saltos necesitan un ciclo de reloj adicional debido al bloqueo, entonces se observará un CPI de 1.17 y por lo tanto una desaceleración de 1.17 respecto al caso ideal.

Si el salto no se puede resolver en la segunda etapa, como ocurre frecuentemente en pipelines más largos, bloquearse ante la presencia de un salto supondrá una mayor disminución del rendimiento. El coste de esta opción es demasiado alto para la mayoría de los computadores y motiva una segunda solución para los riesgos de control:

Predicción: Si se está bastante seguro de que se tiene la fórmula correcta para lavar los uniformes, entonces basta con predecir que funcionará bien y lavar la segunda colada mientras se espera que la primera se seque.

Cuando se acierta, el rendimiento de la segmentación no se reduce. Sin embargo, cuando se falla, se tiene que rehacer la colada que fue lavada mientras se comprobaba si la decisión era correcta.

Los computadores realmente también usan la *predicción* para tratar los saltos. Una estrategia simple es predecir que siempre se dará un salto no tomado. Cuando se acierta, el pipeline funciona a máxima velocidad. El procesador sólo se bloquea cuando los saltos son tomados. La figura 4.32 muestra este ejemplo.

Una versión más sofisticada de la **predicción de saltos (*branch prediction*)** supondría predecir que algunos saltos saltan (se toman) y que otros no saltan (no se toman). En nuestra analogía, los uniformes oscuros o los de casa podrían necesitar una fórmula y los claros o de calle otra fórmula. En los computadores, los saltos que cierran un lazo saltan hacia atrás hasta el principio del lazo. Ya que probablemente estos saltos serán tomados y saltan hacia atrás, se podría predecir como tomados aquellos saltos cuya dirección destino es una dirección anterior a la de la instrucción de salto.

Predicción de saltos: método de resolver los riesgos de saltos que presupone un determinado resultado para el salto y procede a partir de esta suposición, en lugar de esperar a que se establezca el resultado real.

Este tipo de enfoque tan rígido de la predicción de saltos se basa en un comportamiento estereotipado y no tiene en cuenta las características individuales de cada instrucción de salto específica. Los predictores *dinámicos* realizados en hardware, en claro contraste con los explicados anteriormente, hacen sus predicciones dependiendo del comportamiento de cada salto y pueden cambiar las predicciones para un mismo salto a lo largo de la ejecución de un programa. Siguiendo nuestra analogía, usando predicción dinámica una persona miraría cómo está de sucio el uniforme e intentaría adivinar la fórmula, ajustando la siguiente predicción dependiendo del acierto de las últimas predicciones hechas.

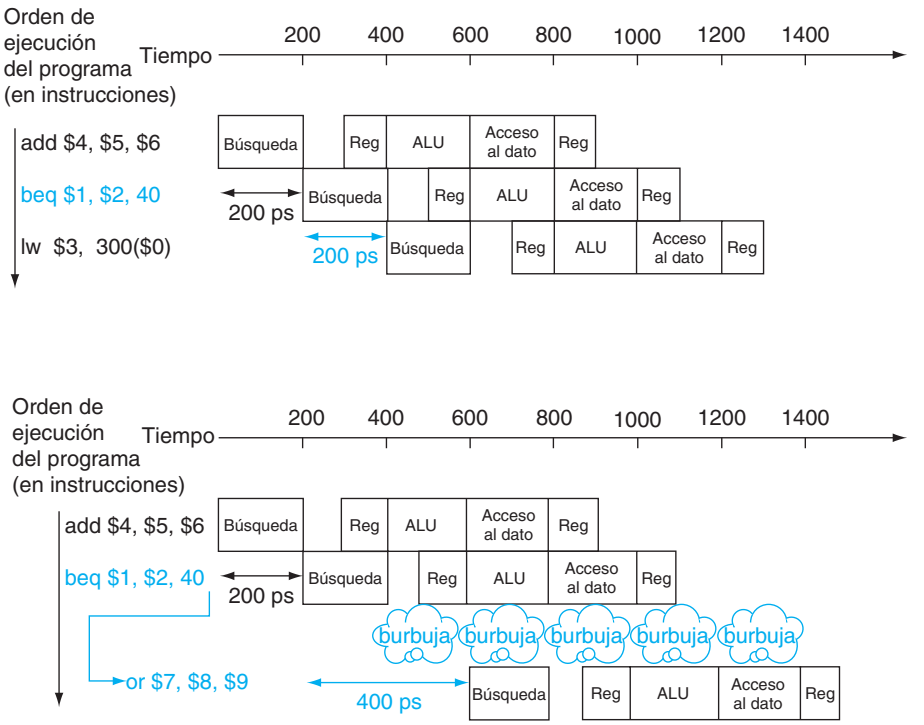


FIGURA 4.32 Predecir que los saltos son no tomados como solución a los riesgos de control. El dibujo de la parte superior muestra el pipeline cuando el salto es no tomado. El dibujo de la parte inferior muestra el pipeline cuando el salto es tomado. Tal como se mostraba en la figura 4.31, la inserción de una burbuja de esta manera simplifica lo que sucede en realidad, al menos durante el primer ciclo de reloj que sigue de forma inmediata al salto. La sección 4.8 revelará los detalles.

Un enfoque muy utilizado en la predicción dinámica de saltos en computadores es mantener una historia para cada salto que indique si ha sido tomado o no tomado, y entonces usar el pasado reciente para predecir el futuro. Tal como se verá más tarde, la cantidad y tipo de historia almacenada ha ido creciendo, con el resultado de que los predictores dinámicos de saltos pueden predecir saltos correctamente con alrededor del 90% de precisión (véase sección 4.8). Cuando la predicción es incorrecta, el control del procesador segmentado debe asegurar que las instrucciones que siguen al salto mal predicho no tienen efecto y debe reinicializar el pipeline desde la dirección correcta de salto. En nuestra analogía de la lavandería, se deben detener las coladas para reiniciar la colada que fue incorrectamente predicha.

Como en el caso de las otras soluciones a los riesgos de control, los pipelines largos agravan el problema, en este caso elevando el coste del fallo de predicción. En la sección 4.8 se describen con más detalle las soluciones a los riesgos de control.

Extensión: Hay un tercer enfoque para los riesgos de control, llamado decisión retardada (*delayed decision*) mencionado anteriormente. En nuestra analogía, cada vez que se vaya a tomar una decisión acerca del lavado se pone en la lavadora ropa que no sea de

fútbol, mientras se espera a que los uniformes de fútbol se sequen. Mientras se tenga suficiente ropa sucia que no se vea afectada por el test, esta solución funciona bien.

En los computadores este enfoque se llama salto retardado (*delayed branch*), y es la solución que se usa en la arquitectura MIPS. El salto retardado siempre ejecuta la siguiente instrucción secuencial, mientras que el salto se ejecuta después del retardo de una instrucción. Esta funcionalidad es oculta al programador de lenguaje ensamblador MIPS, ya que el ensamblador arregla el código automáticamente para que el comportamiento de los saltos sea el que desea el programador. El ensamblador de MIPS pondrá inmediatamente después del salto retardado una instrucción que no dependa del salto, y si el salto es tomado la dirección que se cambiará es la de la instrucción que *sigue* a esta instrucción no dependiente (que es seguro que siempre se va a ejecutar). En el ejemplo, la instrucción `add` que está antes del salto en la figura 4.31 no afecta al salto, y puede moverse después del salto para ocultar completamente su retardo. Como los saltos retardados son útiles cuando los saltos son cortos, ningún procesador usa un salto retardado de más de 1 ciclo. Para retardos mayores de los saltos, generalmente se usa la predicción basada en hardware.

Resumen de la visión general del pipeline

La segmentación es una técnica que explota el paralelismo entre las instrucciones de un flujo secuencial. A diferencia de otras técnicas para incrementar la velocidad del procesador, tiene la ventaja sustancial de ser fundamentalmente invisible al programador.

En las siguientes secciones de este capítulo se analiza el concepto de segmentación utilizando el subconjunto del repertorio de instrucciones MIPS ya utilizado en la implementación de ciclo único de la sección 4.4 y una versión simplificada de su pipeline. Se estudiarán los problemas introducidos por la segmentación y el rendimiento que se puede conseguir en diversas situaciones típicas.

Si el lector desea centrarse más en las implicaciones que suponen la segmentación para el software y para las prestaciones, después de acabar esta sección tendrá suficientes conocimientos básicos como para pasar a la sección 4.10, que introduce conceptos avanzados de la segmentación, como son los procesadores superescalares y la planificación dinámica y la sección 4.11 examina el pipeline de microprocesadores recientes.

Alternativamente, si se está interesado en comprender cómo se implementa el pipeline y los retos de manejar los riesgos, se puede comenzar a examinar el diseño de un camino de datos segmentado y del control básico, que se explican en la sección 4.6. Se debería ser capaz de usar este conocimiento para analizar la implementación de la anticipación de datos y los bloqueos en la sección 4.7. Se puede leer la sección 4.8 para aprender más sobre soluciones para los riesgos de saltos, y después leer en la sección 4.9 cómo se gestionan las excepciones.

Autoevaluación

Para cada una de las secuencias de código mostradas abajo, afirmar si se deben producir bloqueos, si se pueden evitar los bloqueos usando solamente la anticipación de resultados, o si se pueden ejecutar sin bloquear y sin avanzar resultados:

Secuencia 1	Secuencia 2	Secuencia 3

`lw $t0,0($t0)`
`add $t1,$t0,$t0`
	`add $t1,$t0,$t0`
	`addi $t2,$t0,#5`
	`addi $t4,$t1,#5`
		`addi $t1,$t0,#1`
		`addi $t2,$t0,#2`
		`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

Comprender las prestaciones de los programas

Aparte del sistema de memoria, el funcionamiento efectivo del pipeline es generalmente el factor más importante para determinar el CPI del procesador, y por tanto sus prestaciones. Tal como veremos en la sección 4.10, comprender las prestaciones de un procesador segmentado actual con ejecución múltiple de instrucciones es complejo y requiere comprender muchas más cosas de las que surgen del análisis de un procesador segmentado sencillo. De todos modos, los riesgos estructurales, de datos y de control mantienen su importancia tanto en los pipelines sencillos como en los más sofisticados.

En los pipelines actuales, los riesgos estructurales generalmente involucran a la unidad de punto flotante, que no puede ser completamente segmentada, mientras que los riesgos de control son generalmente un problema grande en los programas enteros, que tienden a tener una alta frecuencia de instrucciones de salto además de saltos menos predecibles. Los riesgos de datos pueden llegar a ser cuellos de botella para las prestaciones tanto en programas enteros como de punto flotante. Con frecuencia es más fácil tratar con los riesgos de datos en los programas de punto flotante, porque la menor frecuencia de saltos y un patrón de accesos a memoria más regular facilitan que el compilador pueda planificar la ejecución de las instrucciones para evitar los riesgos. Es más difícil realizar esas optimizaciones con programas enteros que tienen patrones de accesos a memoria menos regulares y que involucran el uso de apuntadores en más ocasiones. Tal como se verá en la sección 4.10, existen técnicas más ambiciosas, tanto por parte del compilador como del hardware, para reducir las dependencias de datos mediante la planificación de la ejecución de las instrucciones.

IDEA clave

La segmentación incrementa el número de instrucciones que se están ejecutando a la vez y la rapidez con que las instrucciones empiezan y acaban. La segmentación no reduce el tiempo que se tarda en completar una instrucción individual, también denominada la **latencia (latency)**. Por ejemplo, el pipeline de cinco etapas todavía necesita que las instrucciones tarden 5 ciclos para ser completadas. Según los términos usados en el capítulo 4, la segmentación mejora la productividad (*throughput*) de instrucciones en vez del *tiempo de ejecución* o *latencia* de cada instrucción.

Los repertorios de instrucciones pueden tanto simplificar como dificultar la tarea de los diseñadores de procesadores segmentados, los cuales tienen ya que hacer frente a los riesgos estructurales, de control y de datos. La predicción de saltos, la anticipación de resultados y los bloqueos ayudan a hacer un computador más rápido y que aún siga produciendo las respuestas correctas.

Latencia (del pipeline): número de etapas en un pipeline, o el número de etapas entre dos instrucciones durante la ejecución.

Aquí hay menos de lo que el ojo puede ver.

Tallulah Bankhead, observación a Alexander Wollcott, 1922

4.6 Camino de datos segmentados y control de la segmentación

La figura 4.33 muestra el camino de datos monociclo de la sección 4.4 con las etapas de segmentación identificadas. La división de una instrucción en cinco pasos implica

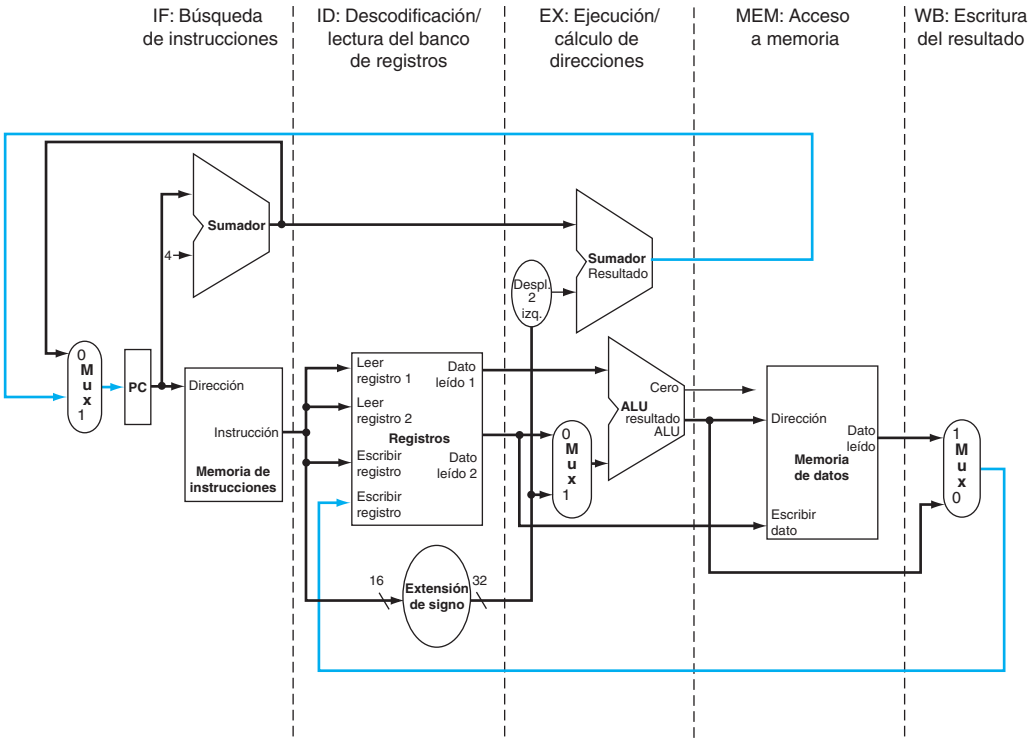


FIGURA 4.33 Camino de datos monociclo extraído de la sección 4.4 (similar al de la figura 4.17). Cada paso de la instrucción se puede situar en el camino de datos de izquierda a derecha. Las únicas excepciones a esta regla son la actualización del registro PC y el paso final de escritura de resultados, mostrados en color. En este último caso, el resultado de la ALU o de la memoria de datos se envía hacia la izquierda para ser escritos en el banco de registros. (Aunque normalmente se utilizan las líneas de color para representar líneas de control, en este caso representan líneas de datos.)

una segmentación de cinco etapas, lo que a su vez significa que durante un ciclo de reloj se estarán ejecutando hasta cinco instrucciones. Por tanto el camino de datos se debe dividir en cinco partes, y cada una de ellas se nombrará haciéndola corresponder con un paso de la ejecución de la instrucción:

1. IF: Búsqueda de instrucciones
2. ID: Descodificación de instrucciones y lectura del banco de registros
3. EX: Ejecución de la instrucción o cálculo de dirección
4. MEM: Acceso a la memoria de datos
5. WB: Escritura del resultado (*write back*)

En la figura 4.33 estas cinco etapas quedan identificadas aproximadamente por la forma como se dibuja el camino de datos. En general, a medida que se va completando la ejecución, las instrucciones y los datos se mueven de izquierda a derecha a través de las cinco etapas. Volviendo a la analogía de la lavandería, la ropa se lava, se seca y se dobla mientras se mueve a través de la línea de segmentación, y nunca vuelve hacia atrás.

Sin embargo, hay dos excepciones a este movimiento de izquierda a derecha de las instrucciones:

- La etapa de escritura de resultados, que pone el resultado en el banco de registros que está situado más atrás, a mitad del camino de datos.
- La selección del siguiente valor del PC, que se elige entre el PC incrementado y la dirección de salto obtenida al final de la etapa MEM.

El flujo de datos de derecha a izquierda no afecta a la instrucción actual. Estos movimientos de datos hacia atrás sólo influyen a las instrucciones que entran al pipeline después de la instrucción en curso. Obsérvese que la primera flecha de derecha a izquierda puede dar lugar a riesgos de datos y la segunda flecha puede dar lugar a riesgos de control.

Una manera de mostrar lo que ocurre en la ejecución segmentada es suponer que cada instrucción tiene su propio camino de datos, y entonces colocar estas rutas de datos en una misma línea de tiempo para mostrar su relación. La figura 4.34 muestra la ejecución de las instrucciones de la figura 4.27 representando en una línea de tiempo común sus rutas de datos particulares. Para mostrar estas relaciones, la figura 4.33 usa una versión estilizada del camino de datos que se había mostrado en la figura 4.34.

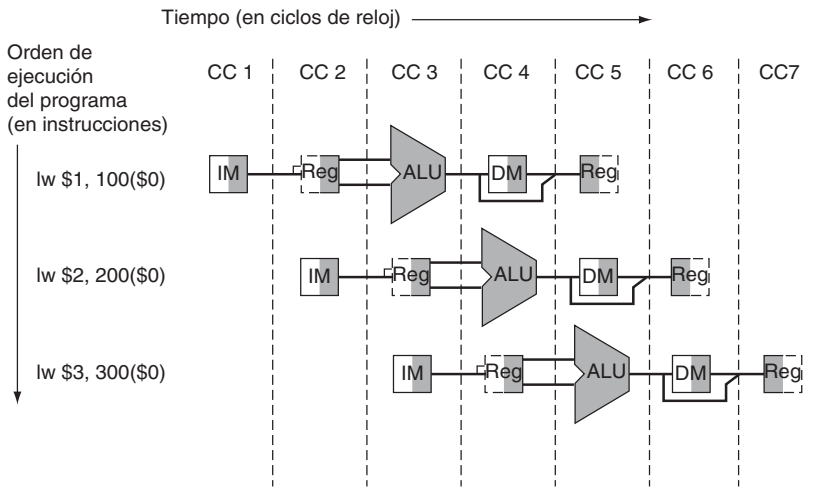


FIGURA 4.34 Ejecución de instrucciones usando el camino de datos monociclo de la figura 4.33, suponiendo que la ejecución está segmentada. De un modo similar a como se hace en las figuras 4.28 a 4.30, esta figura supone que cada instrucción tiene su propio camino de datos, y cada parte del camino de datos está sombreado según su uso. A diferencia de las otras figuras, ahora cada etapa está etiquetada con el recurso físico usado en esa etapa, y que corresponde con la parte del camino de datos mostrado en la figura 4.33. *IM* representa tanto a la memoria de instrucciones como al registro PC de la etapa de búsqueda de instrucciones, *Reg* representa el banco de registros y la unidad de extensión de signo en la etapa de descodificación de instrucciones/lectura del banco de registros (ID), y así todas las demás etapas. Para mantener correctamente el orden temporal, este camino de datos estilizado divide el banco de registros en dos partes lógicas: la lectura de registros durante la etapa ID y la escritura en registro durante la etapa WB. Para representar este doble uso, en la etapa ID se dibuja la mitad izquierda del banco de registros sin sombrear y con líneas discontinuas, ya que no está siendo escrito, mientras que en la etapa WB es la mitad derecha del banco de registros la que se dibuja sin sombrear y con líneas discontinuas, ya que no está siendo leído. Igual que se ha hecho con anterioridad, se supone que la escritura en el banco de registros se hace en la primera mitad del ciclo y la lectura durante la segunda mitad.

La figura 4.34 puede parecer sugerir que tres instrucciones necesitan tres rutas de datos. Así, se añadieron registros para almacenar datos intermedios y permitir así compartir partes del camino de datos durante la ejecución de las instrucciones.

Por ejemplo, tal como muestra la figura 4.34, la memoria de instrucciones sólo se usa durante una de las cinco etapas de una instrucción, permitiendo que sea compartida con otras instrucciones durante las cuatro etapas restantes. Para poder conservar el valor de cada instrucción individual durante las cuatro últimas etapas del pipeline, el valor leído de la memoria de instrucciones se debe guardar en un registro. Aplicando argumentos similares a cada una de las etapas del pipeline, se puede razonar la necesidad de colocar registros en todas las líneas entre etapas de segmentación que se muestran en la figura 4.33. Volviendo a la analogía de la lavandería, se debería tener una cesta entre cada una de las etapas para mantener la ropa producida por una etapa y que debe ser usada por la siguiente etapa.

La figura 4.35 muestra el camino de datos segmentado en la que se han resaltado los registros de segmentación. Durante cada ciclo de reloj todas las instrucciones avanzan de un registro de segmentación al siguiente. Los registros tienen el nombre de las dos etapas que separan. Por ejemplo, el registro de segmentación entre las etapas IF e ID se llama IF/ID.

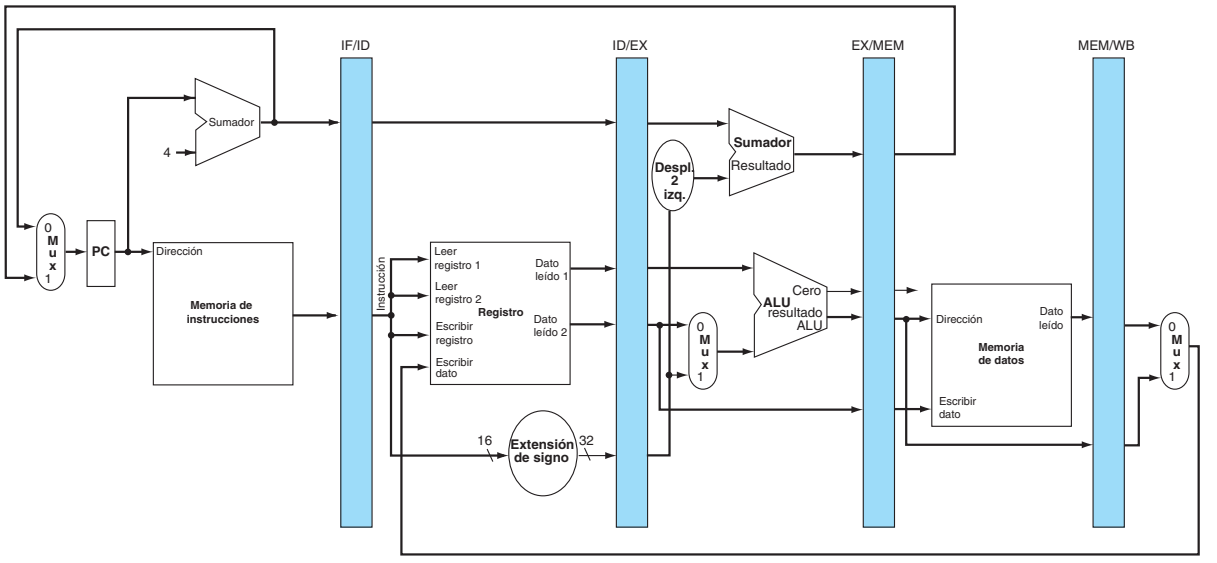


FIGURA 4.35 Versión segmentada del camino de datos de la figura 4.33. Los registros de segmentación, en color, separan cada una de las etapas. Están etiquetados con las etapas que separan; por ejemplo, el primero está etiquetado como IF/ID ya que separa las etapas de búsqueda de instrucciones y de descodificación. Los registros han de ser suficientemente anchos para almacenar todos los datos que corresponden con las líneas que pasan a través de ellos. Por ejemplo, el registro IF/ID debe ser de 64 bits de ancho, ya que debe guardar tanto los 32 bits de la instrucción leída de memoria como los 32 de la dirección obtenida del PC e incrementada. Aunque a lo largo de este capítulo estos registros se ampliarán, de momento supondremos que los otros tres registros de segmentación contienen 128, 97 y 64 bits, respectivamente.

Obsérvese que no hay registro de segmentación al final de la etapa de escritura de resultado. Todas las instrucciones deben actualizar algún estado de la máquina –el banco de registros, la memoria o el registro PC– por lo que sería redundante usar un

registro de segmentación específico para un estado que ya se está actualizando. Por ejemplo, una instrucción de carga guarda su resultado en uno de los 32 registros generales, y cualquier instrucción posterior que necesite el dato puede leerlo directamente de ese registro concreto.

Todas las instrucciones actualizan el registro PC, bien sea incrementando su valor o bien modificando su valor con la dirección destino de un salto. El registro PC se puede considerar un registro de segmentación: el que se utiliza para alimentar la etapa IF del pipeline. Sin embargo, a diferencia de los registros de segmentación mostrados de forma sombreada en la figura 4.35, el registro PC forma parte del estado visible de la arquitectura; es decir, su contenido debe ser guardado cuando ocurre una excepción, mientras que el contenido del resto de registros de segmentación puede ser descartado. En la analogía de la lavandería, se puede considerar que el PC corresponde con la cesta que contiene la carga inicial de ropa sucia antes de la etapa de lavado.

Para mostrar el funcionamiento de la segmentación, durante este capítulo se usarán secuencias de figuras que ilustran la operación sobre el pipeline a lo largo del tiempo. Quizás parezca que se requiere mucho tiempo para comprender estas páginas adicionales. No se debe tener ningún temor: sólo se trata de comparar las secuencias entre sí para identificar los cambios que ocurren en cada ciclo de reloj y ello requiere menos esfuerzo de comprensión del que podría parecer. La sección 4.7 describe lo que ocurre cuando se producen riesgos de datos entre las instrucciones segmentadas, así que de momento pueden ser ignorados.

Las figuras 4.36 a 4.38, que suponen la primera secuencia, muestran resaltadas las partes activas del pipeline a medida que una instrucción de carga avanza a través de las cinco etapas de la ejecución segmentada. Se muestra en primer lugar una instrucción de carga porque está activa en cada una de las cinco etapas. Igual que en las figuras 4.28 a 4.30, se resalta la *mitad derecha* del banco de registros o de la memoria cuando están siendo *leídos* y se resalta la *mitad izquierda* cuando están siendo *escritos*.

En cada figura se muestra la abreviación de la instrucción, `lw`, junto con el nombre de la etapa que está activa. Las cinco etapas son las siguientes:

1. *Búsqueda de instrucción*: La parte superior de la figura 4.36 muestra cómo se lee la instrucción de memoria usando la dirección del PC y después se coloca en el registro de segmentación IF/ID. La dirección del PC se incrementa en 4 y se escribe de nuevo en el PC para prepararse para el siguiente ciclo de reloj. Esta dirección incrementada se guarda también en el registro IF/ID por si alguna instrucción, como por ejemplo `beq`, la necesita con posterioridad. El computador no puede conocer el tipo de instrucción que se está buscando hasta que ésta es descodificada, por lo que debe estar preparado ante cualquier posible instrucción, pasando la información que sea potencialmente necesaria a lo largo del pipeline.

2. *Descodificación de instrucción y lectura del banco de registros*: La parte inferior de la figura 4.36 muestra la parte del registro de segmentación IF/ID donde está guardada la instrucción. Este registro proporciona el campo inmediato de 16 bits, que es extendido a 32 bits con signo, y los dos identificadores de los registros que se deben leer. Los tres valores se guardan, junto con la dirección del PC incrementada, en el registro ID/EX. Una vez más se transfiere todo lo que pueda necesitar cualquier instrucción durante los ciclos posteriores.

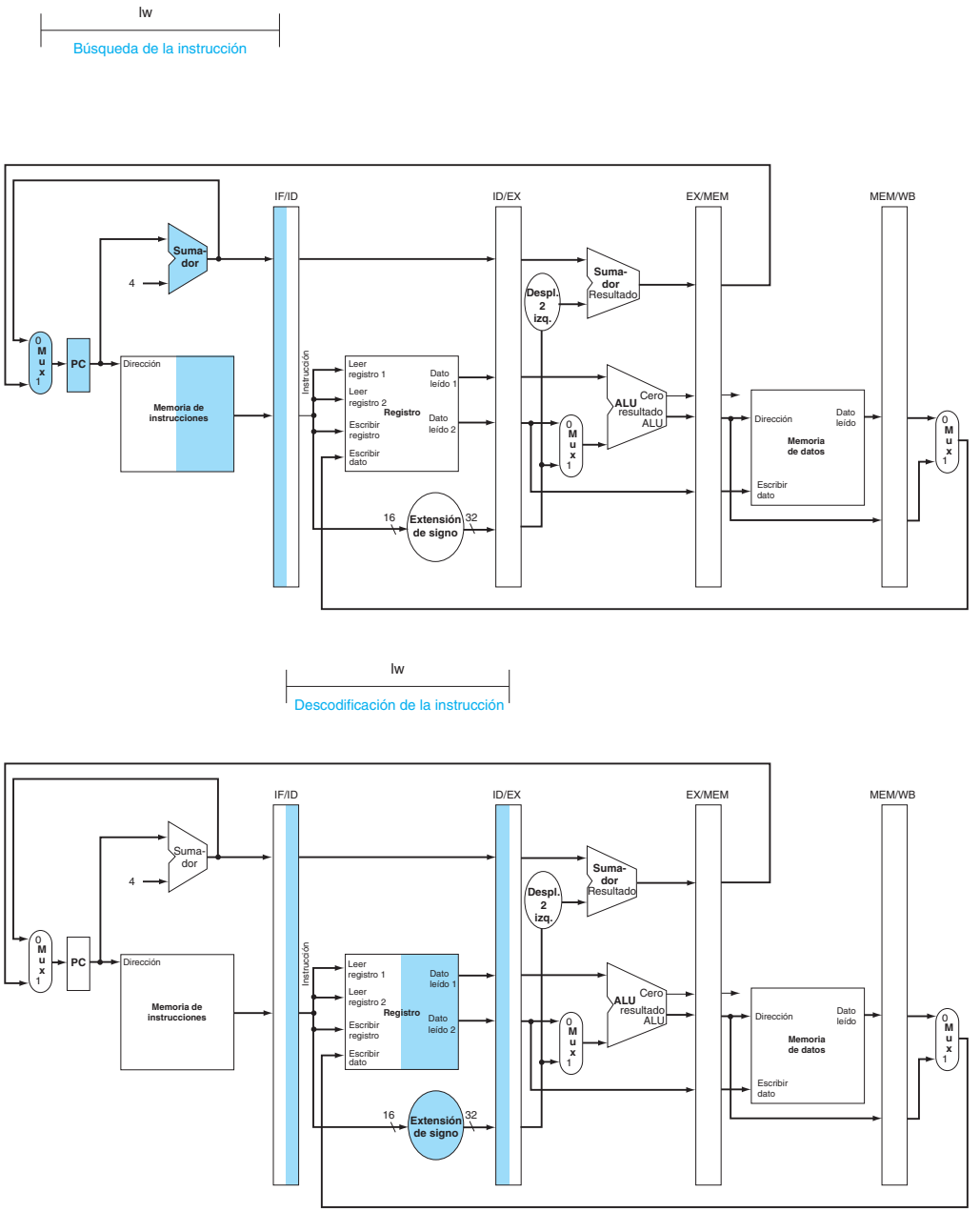


FIGURA 4.36 IF e ID: primera y segunda etapa de segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en estas dos etapas. La convención para resaltar los elementos del camino de datos es la misma que se usó en la figura 4.28. Igual que en la sección 4.2, no hay confusión al leer y escribir registros porque el contenido de éstos cambia sólo con la transición de la señal de reloj. Aunque la instrucción de carga en la etapa 2 sólo necesita el registro de arriba, el procesador no sabe qué instrucción se está descodificando, así que extiende el signo de la constante de 16 bits obtenida de la instrucción y lee ambos registros de entrada y los tres valores se almacenan sobre el registro de segmentación ID/EX. Seguro que no se necesitan los tres operandos, pero disponer de los tres simplifica el control.

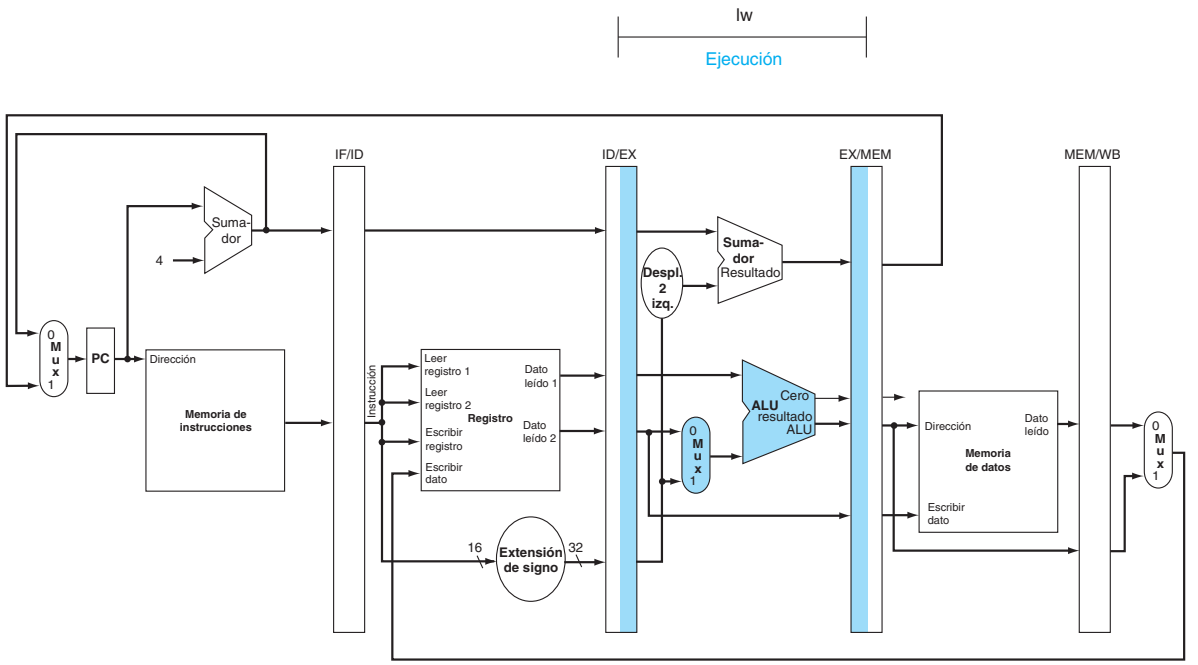


FIGURA 4.37 EX: tercera etapa de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. El registro se suma al valor inmediato con el signo extendido, y el resultado de la suma se coloca en el registro de segmentación EX/MEM.

3. *Ejecución o cálculo de dirección*: La figura 4.38 muestra que la instrucción de carga lee del registro IF/ID el contenido del registro 1 y el valor inmediato con el signo extendido, y los suma usando la ALU. El resultado de esta suma se coloca en el registro de segmentación EX/MEM.
4. *Acceso a memoria*: La parte superior de la figura 4.38 muestra la instrucción de carga cuando usa la dirección obtenida del registro EX/MEM para leer un dato de memoria y después guardar el dato leído en el registro de segmentación MEM/WB.
5. *Escritura de resultado*: La parte inferior de la figura 4.38 muestra el paso final: la lectura del resultado guardado en el registro MEM/WB y la escritura de este resultado en el banco de registros mostrado en el centro de la figura.

Este recorrido de la instrucción de carga indica que toda la información que se pueda necesitar en etapas posteriores del pipeline se debe pasar a cada etapa mediante los registros de segmentación. El recorrido de una instrucción de almacenamiento es similar en la manera de ejecutarse y en la manera de pasar la información a las etapas posteriores del pipeline. A continuación se muestran las cinco etapas de un almacenamiento:

1. *Búsqueda de la instrucción*: Se lee la instrucción de la memoria usando la dirección del PC y se guarda en el registro IF/ID. Esta etapa ocurre antes de

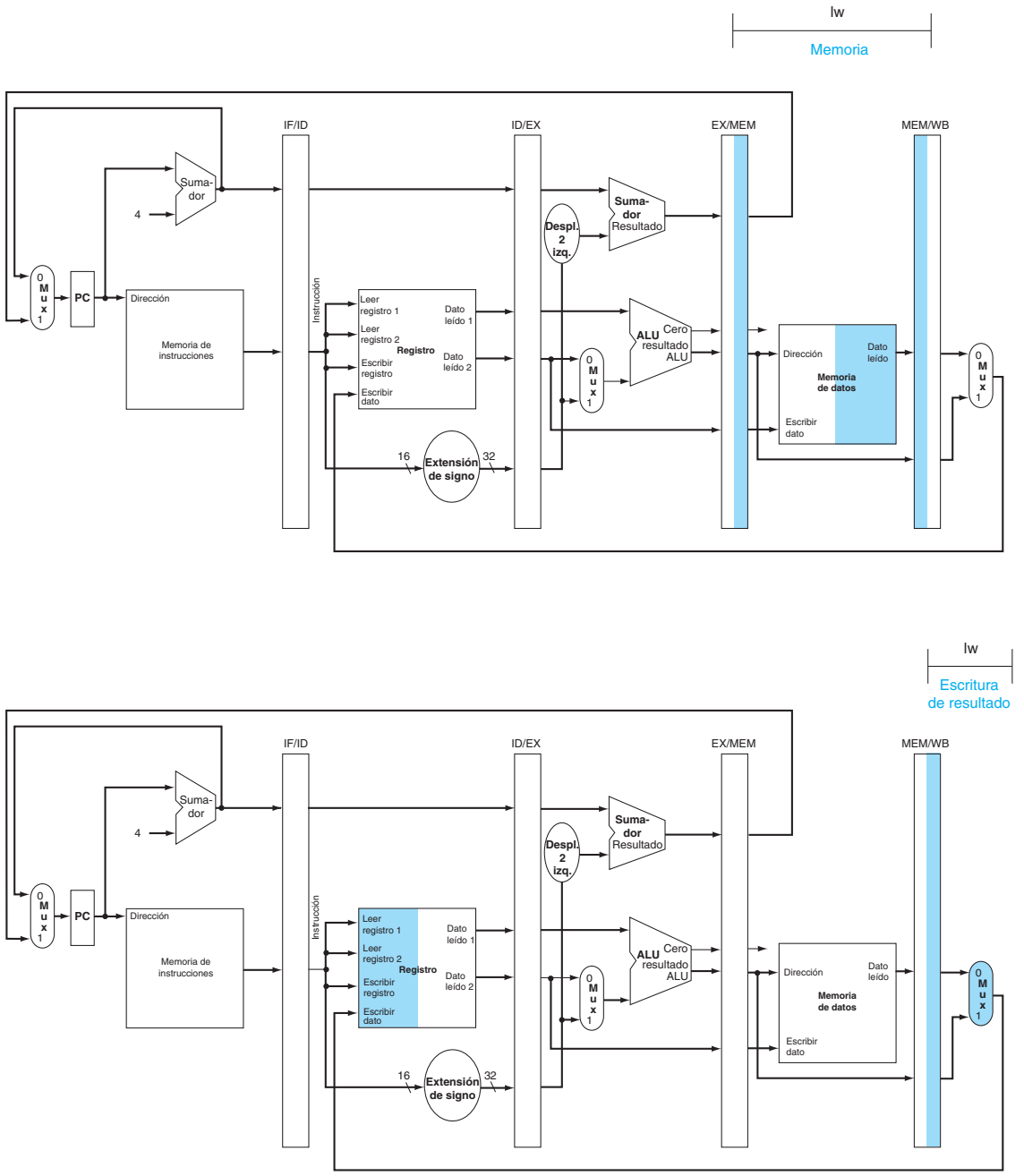


FIGURA 4.38 MEM y WB: cuarta y quinta etapas de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. Se lee la memoria de datos usando la dirección contenida en el registro EX/MEM, y el dato leído se guarda en el registro de segmentación MEM/WB. A continuación, este dato, guardado en el registro de segmentación MEM/WB, se escribe en el banco de registros, que se encuentra en el medio del camino de datos. Nota: hay un error en este diseño que se corrige en la figura 4.41.

que se identifique la instrucción, por lo que la parte superior de la figura 4.36 sirve igual tanto para cargas como para almacenamientos.

2. *Descodificación de instrucción y lectura del banco de registros*: La instrucción guardada en el registro IF/ID proporciona los identificadores de los dos registros que deben ser leídos y proporciona el valor inmediato de 16 bits cuyo signo ha de ser extendido. Estos tres valores de 32 bits se guardan en el registro de segmentación ID/EX. La parte inferior de la figura 4.36 para las instrucciones de carga también muestra las operaciones a realizar en la segunda etapa de las instrucciones de almacenamiento. En realidad, estos dos primeros pasos se ejecutan siempre igual para todas las instrucciones, ya que es demasiado pronto para que se conozca el tipo de la instrucción
3. *Ejecución o cálculo de dirección*: La figura 4.39 muestra el tercer paso; la dirección efectiva se coloca en el registro de segmentación EX/MEM.
4. *Acceso a memoria*: La parte superior de la figura 4.40 muestra el dato que se está escribiendo en memoria. Observe que el registro que contiene el dato que se tiene que guardar en memoria fue leído en una etapa anterior y guardado en el registro ID/EX. La única manera de hacer que el dato esté disponible durante la etapa MEM es que se coloque en el registro de segmentación EX/MEM durante la etapa EX, de la misma manera que también se ha guardado la dirección efectiva.
5. *Escritura de resultado*: La parte inferior de la figura 4.40 muestra el paso final del almacenamiento. En esta etapa no ocurre nada para esta instrucción. Puesto que todas las instrucciones posteriores al almacenamiento ya están en progreso, no hay manera de acelerarlas aprovechando que la instrucción de almacenamiento no tiene nada que hacer. En general, las instrucciones pasan a través de todas las etapas aunque en ellas no tengan nada que hacer, ya que las instrucciones posteriores ya están progresando a la máxima velocidad.

La instrucción de almacenamiento ilustra una vez más que, para pasar datos de una etapa del pipeline a otra posterior, la información se debe colocar en un registro de segmentación; si no se hiciera así, la información se perdería cada vez que llegase la siguiente instrucción. Para ejecutar el almacenamiento se ha necesitado pasar uno de los registros leídos en la etapa ID a la etapa MEM, donde se guarda en memoria. El dato se ha tenido que guardar primero en el registro de segmentación ID/EX y después se ha tenido que pasar al registro de segmentación EX/MEM.

Las cargas y almacenamientos ilustran un segundo punto importante: cada componente lógico del camino de datos —como la memoria de instrucciones, los puertos de lectura del banco de registros, la ALU, la memoria de datos y el puerto de escritura en el banco de registros— pueden usarse solamente dentro de una única etapa de la segmentación. De otra forma se tendría un *riesgo estructural* (véase la página 335). Por lo tanto, estos componentes y su control pueden asociarse a una sola etapa de la segmentación.

En este momento ya se puede destapar un error en el diseño de la instrucción de carga. ¿Lo ha descubierto? ¿Qué registro se modifica en la última etapa de la carga? Más específicamente, ¿qué instrucción proporciona el identificador del registro de escritura? La instrucción guardada en el registro de segmentación

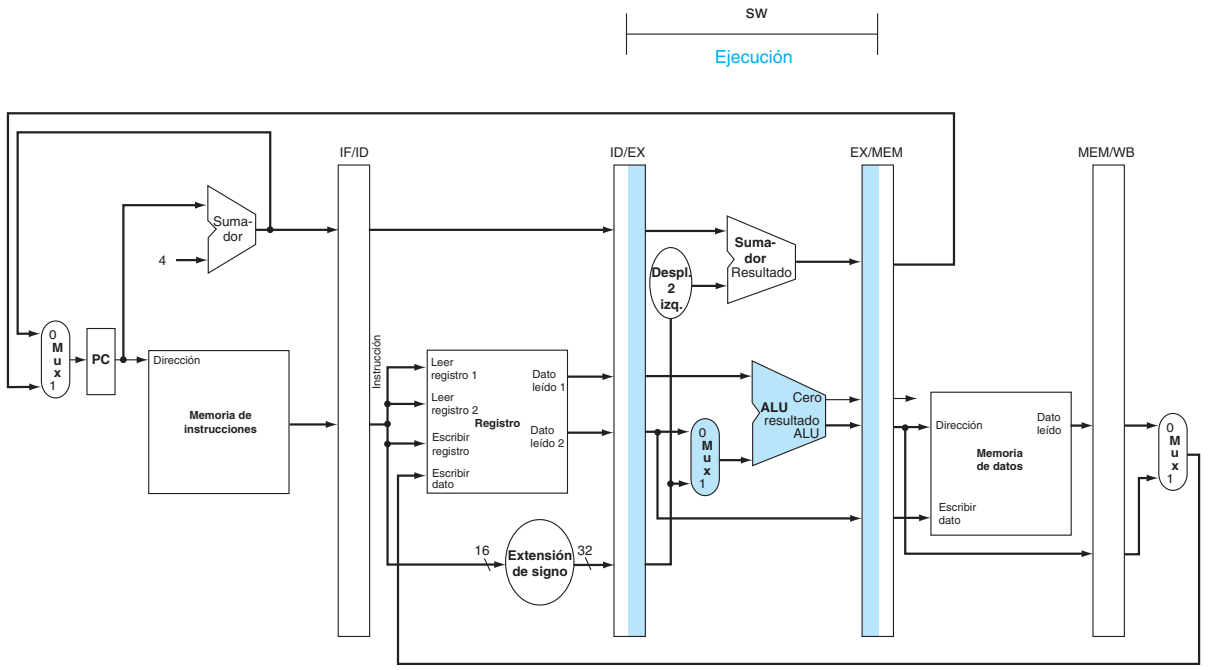


FIGURA 4.39 EX: tercera etapa de la segmentación de una instrucción de almacenamiento. A diferencia de la tercera etapa de la instrucción de carga en la figura 4.37, el valor del segundo registro es cargado en el registro de segmentación EX/MEM para ser usado en la siguiente etapa. Aunque no importaría mucho si siempre se escribiera este segundo registro en el registro de segmentación EX/MEM, para hacer el pipeline más fácil de entender sólo se escribirá en el caso de las instrucciones store.

IF/ID proporciona el número del registro sobre el que se va a escribir, ¡pero en el orden de ejecución, esta instrucción es bastante posterior a la carga que debe hacer la escritura!

Por lo tanto, para la instrucción de carga es necesario conservar el identificador del registro destino. Así como el almacenamiento ha pasado el contenido del registro que se tenía que guardar en memoria desde el registro de segmentación ID/EX al registro EX/MEM para poder ser usado posteriormente en la etapa MEM, también la carga debe pasar el identificador de registro destino desde ID/EX a través de EX/MEM y hasta MEM/WB para que pueda ser usado correctamente en la etapa WB. Otra manera de interpretar el paso del número del registro es que, para poder compartir el camino de datos segmentado, es necesario preservar la información de la instrucción que se ha leído en la etapa IF, de modo que cada registro de segmentación contiene las partes de la instrucción necesarias tanto para esa etapa como para las siguientes.

La figura 4.41 muestra la versión correcta del camino de datos, en la que se pasa el número del registro de escritura primero al registro ID/EX, después al EX/MEM y finalmente al MEM/WB. Este identificador del registro se usa durante la etapa WB para especificar el registro sobre el que se debe escribir. La figura 4.42 representa el camino de datos correcto en un solo dibujo, resaltando el hardware que se usa en cada una de las cinco etapas de la carga de las figuras 4.36 a 4.38. La explicación de cómo lograr que la instrucción de salto funcione tal y como se espera se dará en la sección 4.8.

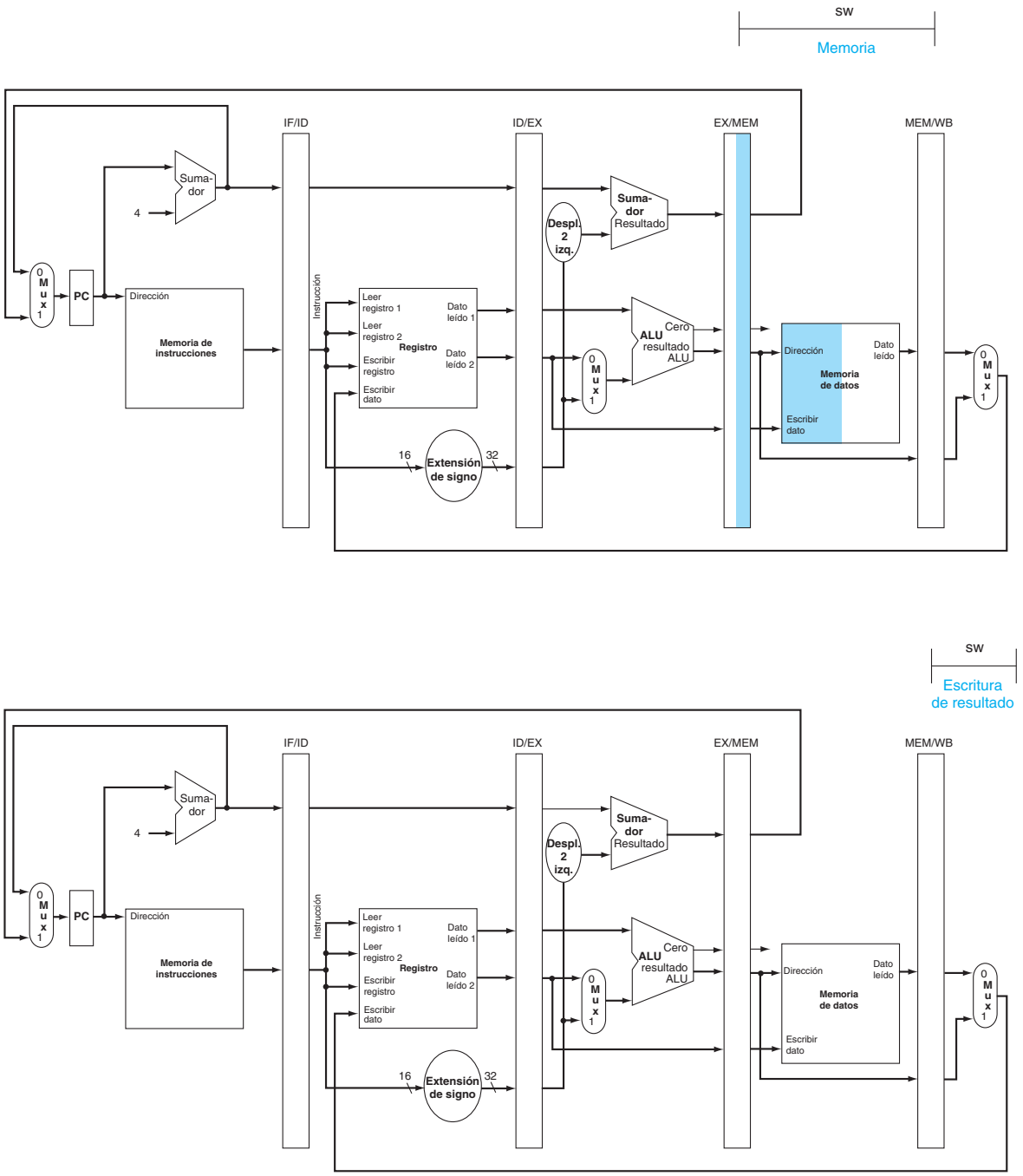


FIGURA 4.40 MEM y WB: cuarta y quinta etapas en la ejecución de un almacenamiento. En la cuarta etapa, el dato se escribe en la memoria de datos. Observe que el dato viene del registro de segmentación EX/MEM y que no se cambia nada en el registro de segmentación MEM/WB. Una vez escrito el dato en memoria, no hay nada más que hacer para el almacenamiento, por lo que en la etapa 5 no ocurre nada.

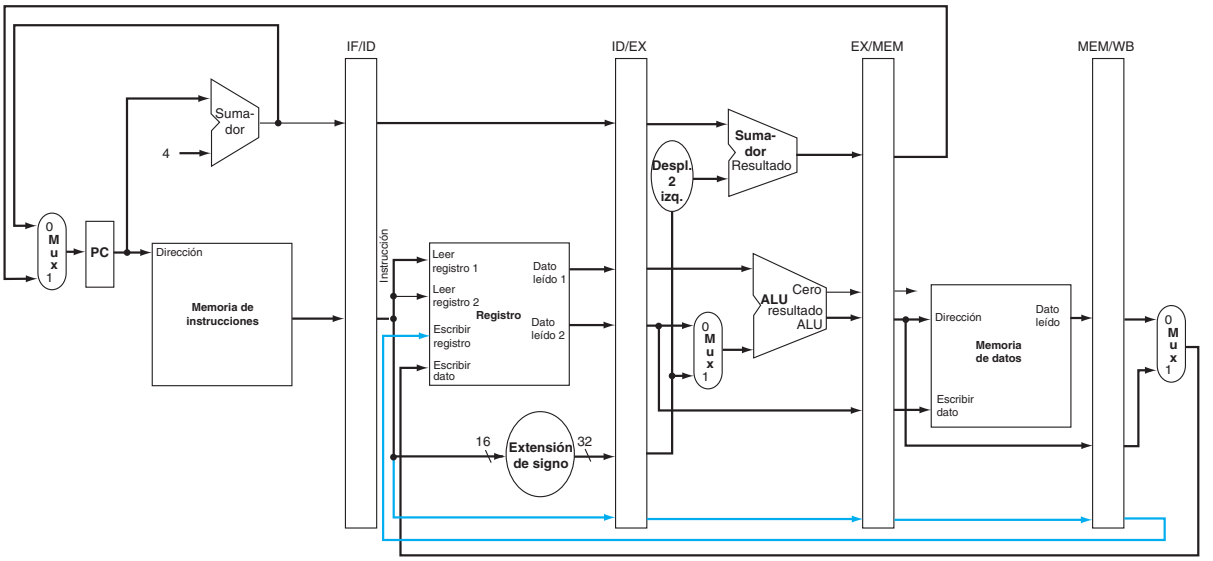


FIGURA 4.41 Camino de datos correctamente modificado para gestionar debidamente la instrucción de carga. Ahora el identificador del registro de escritura viene, junto con el dato a escribir, del registro segmentado MEM/WB. Este identificador se pasa desde la etapa ID hasta que llega al registro de segmentación MEM/WB, añadiendo 5 bits más a los tres últimos registros de segmentación. Este nuevo camino se muestra coloreado.

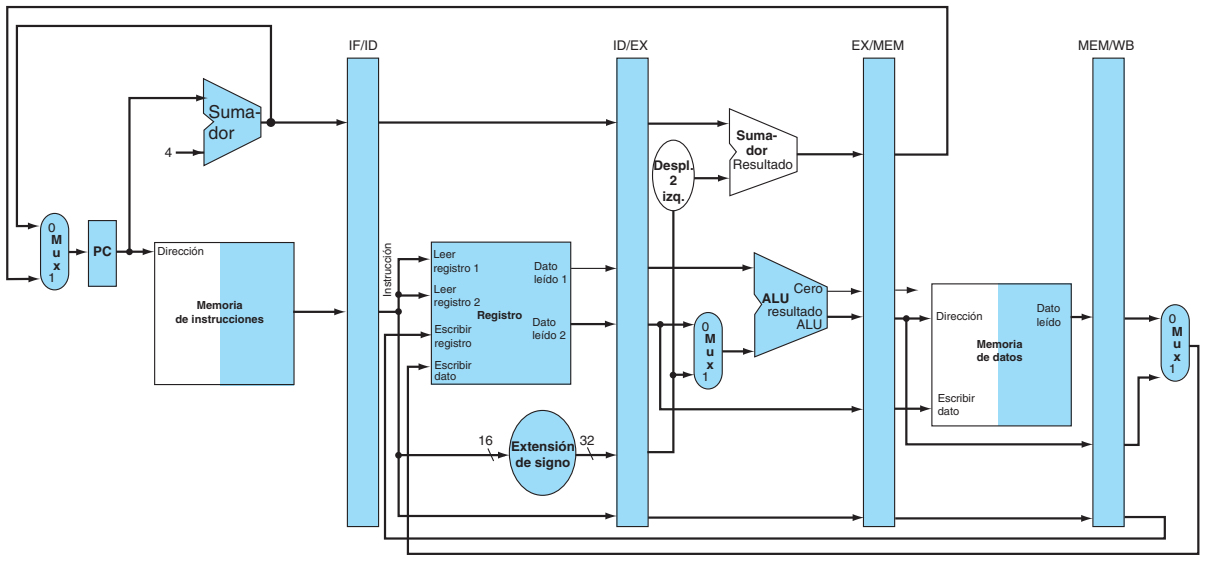


FIGURA 4.42 Porción del camino de datos de la figura 4.41 que es usada por las cinco etapas de la instrucción de carga.

Representación gráfica de la segmentación

La segmentación puede ser difícil de entender, ya que en cada ciclo de reloj hay varias instrucciones ejecutándose simultáneamente en la mismo camino de datos. Para ayudar a entenderla, el pipeline se dibuja usando dos estilos básicos: *diagramas multiciclo de segmentación*, como el de la figura 4.34 de la página 346, y *diagramas monociclo de segmentación*, como los de las figuras 4.36 a 4.40. Los diagramas multiciclo son más simple, pero no contienen todos los detalles. Por ejemplo, consideremos la siguiente secuencia de cinco instrucciones:

```
lw     $10, 20($1)
sub    $11, $2, $3
add    $12, $3, $4
lw     $13, 24($1)
add    $14, $5, $6
```

La figura 4.43 muestra el diagrama multiciclo de segmentación para estas instrucciones. El tiempo avanza de izquierda a derecha a lo largo de la página y las instrucciones avanzan desde la parte superior a la parte inferior de la página, de una forma parecida a como se representaba la segmentación de la lavandería de la figura 4.25. En cada fila del eje de las instrucciones, a lo largo de él, se coloca una representación de las etapas del pipeline, ocupando los ciclos que sean necesarios. Estas rutas de datos estilizadas representan las cinco etapas de nuestro pipeline, pero un rectángulo con el nombre de cada etapa funciona igual de bien. La figura 4.44 muestra la versión más tradicional del diagrama multiciclo de la segmentación. Debe notarse que la figura 4.43 muestra los recursos físicos usados en cada etapa, mientras que la figura 4.44 emplea el nombre de cada etapa.

Los diagramas monociclo muestran el estado del camino de datos completo durante un ciclo de reloj, y las cinco instrucciones que se encuentran en las cinco etapas diferentes del pipeline normalmente se identifican con etiquetas encima de las respectivas etapas. Se usa este tipo de figura para mostrar con más detalle lo que está ocurriendo durante cada ciclo de reloj dentro del pipeline. Habitualmente los diagramas se agrupan para mostrar las operaciones de la segmentación a lo largo de una secuencia de ciclos. Usaremos los diagramas multiciclo para dar visiones generales de las distintas circunstancias de la segmentación. (Si se quieren ver más detalles de la figura 4.43, en la sección 4.12 se pueden encontrar más diagramas monociclo). Un diagrama monociclo representa un corte vertical de un diagrama multiciclo, mostrando la utilización del camino de datos por cada una de las instrucciones que se encuentran en el pipeline durante un determinado ciclo de reloj. Por ejemplo, la figura 4.45 muestra el diagrama monociclo que corresponde al quinto ciclo de las figuras 4.43 y 4.44. Obviamente, los diagramas monociclo incluyen más detalles y requieren un espacio significativamente mayor para mostrar lo que ocurre durante un cierto número de ciclos. Los ejercicios le pedirán que cree estos diagramas para otras secuencias de código.

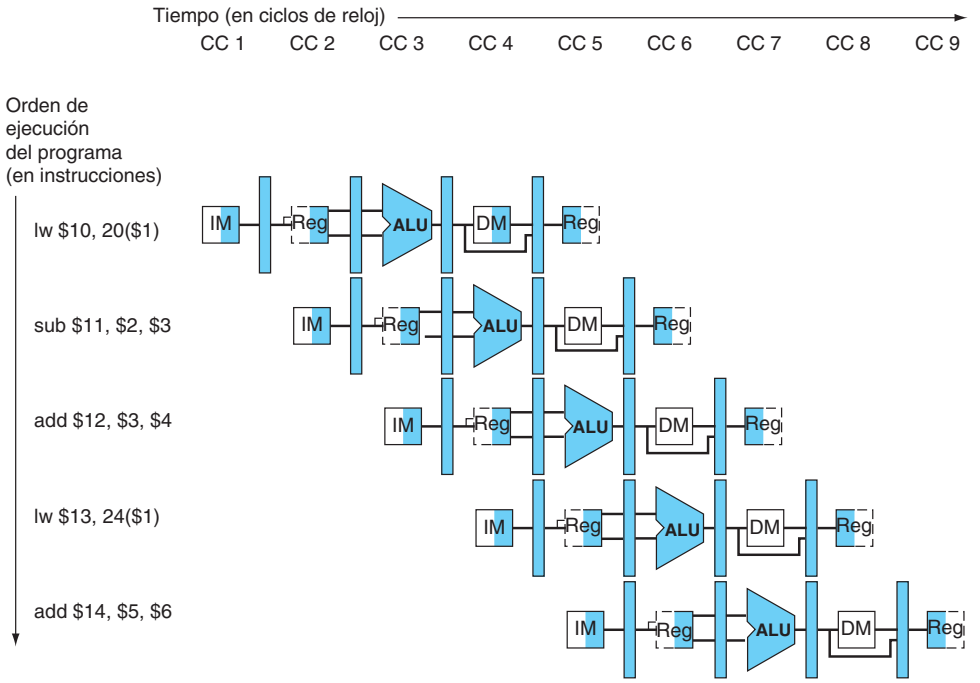


FIGURA 4.43 Diagrama multiciclo de la segmentación de cinco instrucciones. Este estilo de representación del pipeline muestra la ejecución completa de las instrucciones en una sola figura. La relación de instrucciones se hace en orden de ejecución desde la parte superior a la inferior, y los ciclos de reloj avanzan de izquierda a derecha. Al contrario de la figura 4.28, aquí se muestran los registros de segmentación entre cada etapa. La figura 4.44 muestra la manera tradicional de dibujar este diagrama.

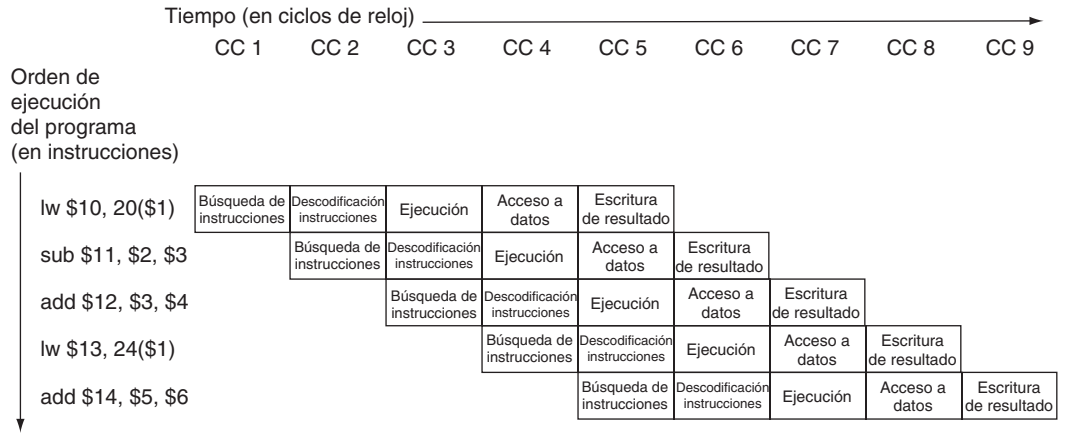


FIGURA 4.44 Versión tradicional del diagrama multiciclo de la segmentación de cinco instrucciones que se muestra en la figura 4.43.

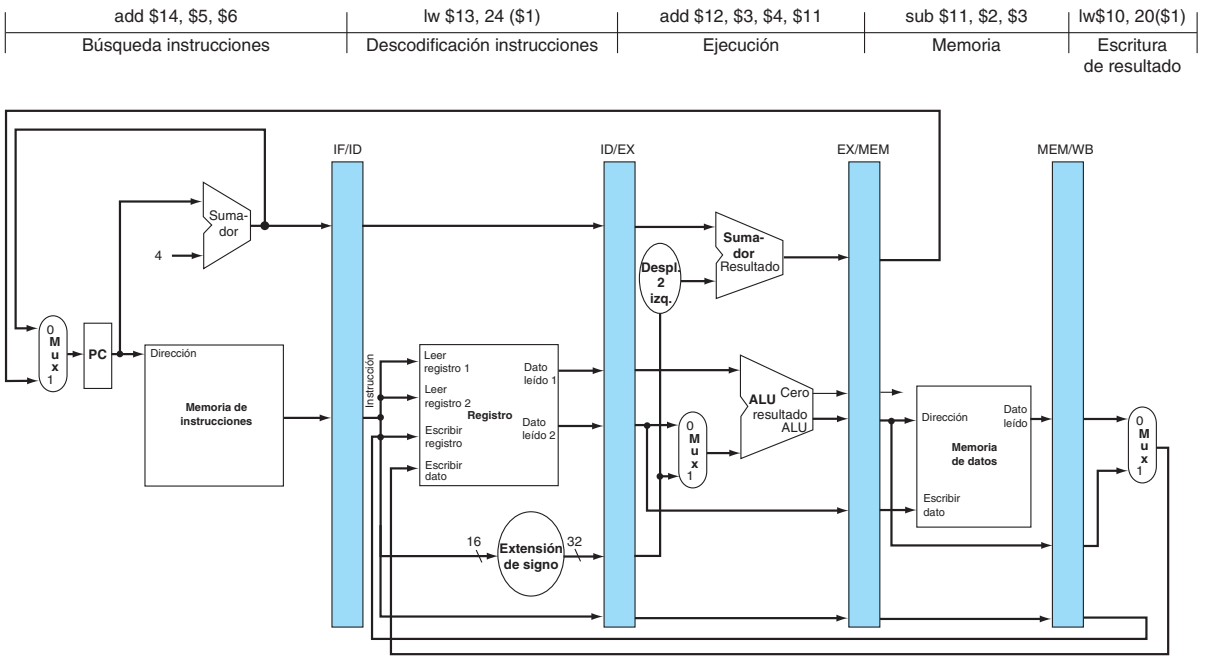


FIGURA 4.45 Diagrama monociclo correspondiente al ciclo 5 del pipeline de las figuras 4.43 y 4.44. Como puede verse, la figura monociclo es una porción vertical del diagrama multiciclo.

Autoevaluación

Un grupo de estudiantes estaba debatiendo sobre la eficiencia de un pipeline de cinco etapas, cuando uno de ellos se dio cuenta de que no todas las instrucciones están activas en cada una de las etapas del pipeline. Después de decidir ignorar el efecto de los riesgos, los estudiantes hicieron las siguientes afirmaciones. ¿Cuáles son correctas?

1. Permitiendo que las instrucciones de salto condicional e incondicional y que las instrucciones que usan la ALU tarden menos ciclos que los cinco requeridos por la instrucción de carga incrementará las prestaciones en todos los casos.
2. Intentar que algunas instrucciones tarden menos ciclos en el pipeline no ayuda, ya que la productividad viene determinada por el ciclo de reloj; el número de etapas del pipeline que requiere cada instrucción afecta a la latencia y no a la productividad.
3. No se puede hacer que las instrucciones que usan la ALU tarden menos ciclos debido a la escritura del resultado final, pero las instrucciones de salto condicional e incondicional sí que pueden tardar menos ciclos, y por tanto hay alguna oportunidad de mejorar.
4. En lugar de tratar que las instrucciones tarden menos ciclos, deberíamos explorar la posibilidad de hacer que el pipeline fuera más largo, de forma que las instrucciones tardaran más ciclos, pero que los ciclos fueran más cortos. Esto podría mejorar las prestaciones.

Control de la segmentación

En el Computador 6600, quizás aún más que en cualquier computador anterior, es el sistema de control el que marca la diferencia.

James Thornton, *Design of a Computer: The Control Data 6600,* 1970

Del mismo modo que en la sección 4.3 añadimos el control a un camino de datos de ciclo único, ahora añadiremos el control a un camino de datos segmentado. Comenzaremos con un diseño simple en el que el problema se verá a través de una gafas con cristales de color rosa. En las secciones 4.7 a 4.9, se prescindirá de estas gafas para desvelar así los riesgos presentes en el mundo real.

El primer paso consiste en etiquetar las líneas de control en el nuevo camino de datos. La figura 4.46 muestra estas líneas. El control del camino de datos sencillo de la figura 4.17 se ha reutilizado lo máximo posible. En concreto, se usa la misma lógica de control para la ALU, la misma lógica de control para los saltos, el mismo multiplexor para los identificadores de registro destino, y las mismas líneas de control. Estas funciones se definieron en las figuras 4.12, 4.16 y 4.18. Para que el texto que sigue sea más fácil de seguir, las figuras 4.47 a 4.48 reproducen la misma información clave.

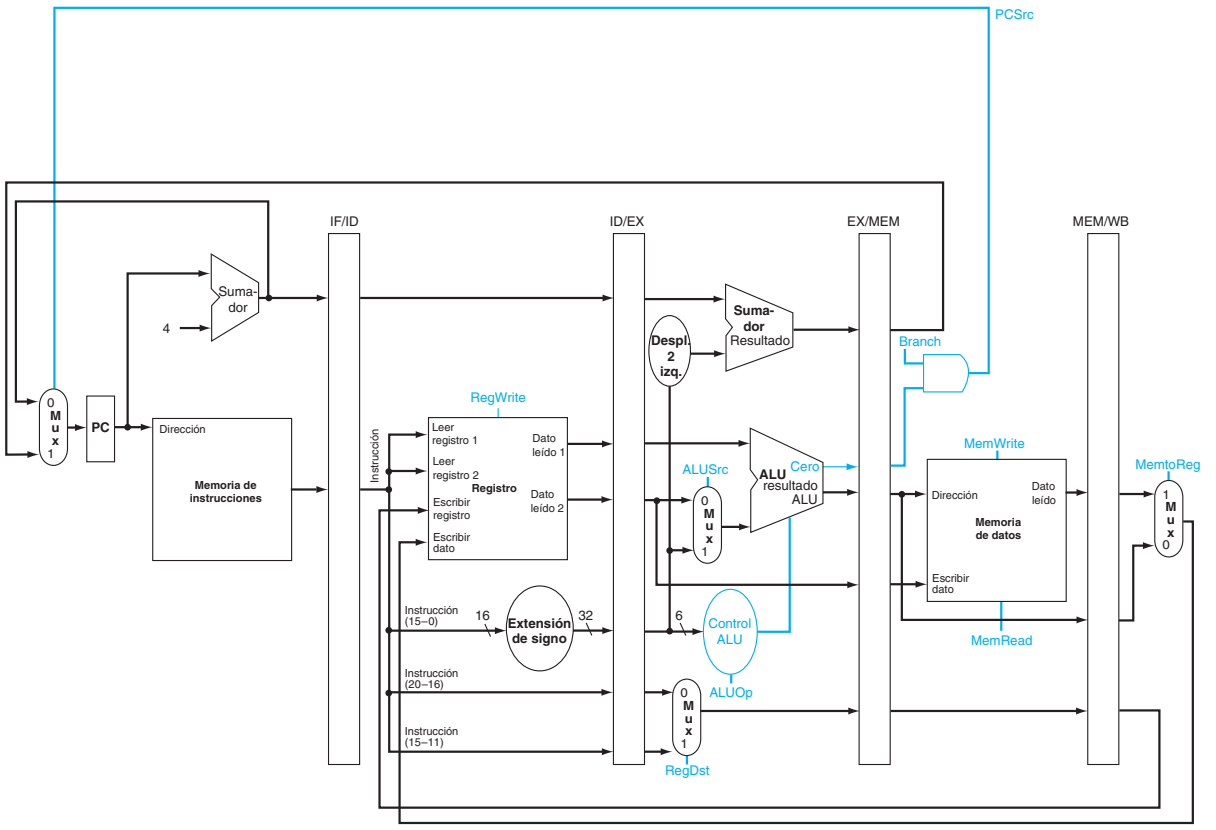


FIGURA 4.46 El camino de datos segmentado de la figura 4.41 en la que se identifican las señales de control. Este camino de datos toma prestada la lógica de control de la sección 4.4 para el PC, el identificador de registro fuente y destino, y el control de la ALU. Debe notarse que en la etapa EX, como entrada de control de la ALU, son ahora necesarios los 6 bits del campo funct (código de función) de la instrucción, y por tanto estos bits también deben ser incluidos en el registro de segmentación ID/EX. Recuerde que ya que estos 6 bits también pueden representar los 6 bits menos significativos del campo inmediato de la instrucción, el registro de segmentación ID/EX los proporciona como parte del campo inmediato, pues la extensión del signo mantiene el valor de los bits.

Código de operación	ALUOp	Operación	Código Función	Acción deseada en ALU	Entrada Control ALU
LW	00	cargar palabra	XXXXXX	sumar	0010
SW	00	almacenar palabra	XXXXXX	sumar	0010
Saltar si igual	01	saltar si igual	XXXXXX	restar	0110
tipo R	10	sumar	100000	sumar	0010
tipo R	10	restar	100010	restar	0110
tipo R	10	AND	100100	Y-lógica	0000
tipo R	10	OR	100101	O-lógica	0001
tipo R	10	iniciar si menor que	101010	iniciar si menor que	0111

FIGURA 4.47 Copia de la figura 4.12. Esta figura muestra cómo activar los bits de control de la ALU dependiendo de los bits de control de ALUOp y de los diferentes códigos de función de las instrucciones de tipo R.

Nombre de señal	Efecto cuando desactiva (0)	Efecto cuando activa (1)
RegDst	El identificador del registro destino para la escritura a registro viene del campo rt (bits 20:16).	El identificador del registro destino para la escritura a registro viene del campo rd (bits 15:11).
RegWrite	Ninguno.	El registro se escribe con el valor de escritura.
ALUSrc	El segundo operando de la ALU proviene del segundo registro leído del banco de registros.	El segundo operando de la ALU son los 16 bits de menor peso de la instrucción con el signo extendido.
PCSrc	El PC es reemplazado por su valor anterior más 4 (PC + 4).	El PC es reemplazado por la salida del sumador que calcula la dirección destino del salto.
MemRead	Ninguno.	El valor de la posición de memoria designada por la dirección se coloca en la salida de lectura.
MemWrite	Ninguno.	El valor de la posicón de memoria designada por la dirección se reemplaza por el valor de la entrada de datos.
MemtoReg	El valor de entrada del banco de registros proviene de la ALU.	El valor de entrada del banco de registros proviene de la memoria.

FIGURA 4.48 Copia de la figura 4.16. Se define la función de cada una de las siete señales de control. Las líneas de control de la ALU (ALUOp) se definen en la segunda columna de la figura 4.47. Cuando se activa el bit de control de un multiplexor de dos entradas, éste selecciona la entrada correspondiente a 1. En caso contrario, si el control está desactivado, el multiplexor selecciona la entrada 0. Obsérvese que en la figura 4.46, PCSrc se controla mediante una puerta AND. Si la señal Branch y la señal Cero de la ALU están activadas, entonces la señal PCSrc es 1; en caso contrario es 0. El control activa la señal Branch sólo para una instrucción `beq`; en otro caso PCSrc se pone a 0.

Instrucción	Ejecución / cálculo de dirección líneas de control				Acceso a memoria líneas de control			Escritura de resultado líneas de control	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
Formato R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

FIGURA 4.49 Los valores de las líneas de control son los mismos que en la figura 4.18, pero han sido distribuidas en tres grupos que corresponden a las tres últimas etapas de la segmentación.

Igual que para la implementación monociclo, se supondrá que el PC se escribe en cada ciclo y que no es necesaria una línea separada para controlar la escritura en el PC. Por el mismo motivo, no existen señales de escritura separadas para los registros de segmentación (IF/ID, ID/EX, EX/MEM y MEM/WB), en los cuales también se escribe en cada ciclo de reloj.

Para especificar el control en el pipeline sólo se necesita activar los valores de control durante cada etapa de la segmentación. Puesto que cada línea de control se asocia con un componente activo en una única etapa, las líneas de control se pueden dividir en cinco grupos según las etapas de la segmentación:

1. *Búsqueda de instrucción*: Las señales de control para leer de la memoria de instrucciones y para escribir el PC están siempre activadas, por lo que el control en esta etapa no tiene nada de especial.

2. *Descodificación de instrucción y lectura del banco de registros:* Aquí pasa lo mismo que en la etapa anterior, por lo que no hay líneas de control opcionales que activar.

3. *Ejecución / cálculo de dirección*: Las señales a activar son RegDst, ALUOp y ALUSrc (véanse las figuras 4.47 y 4.48). Estas señales seleccionan el registro de resultado, la operación de la ALU, y seleccionan como entrada de la ALU o bien el dato leído del segundo registro o bien el valor inmediato con signo extendido.

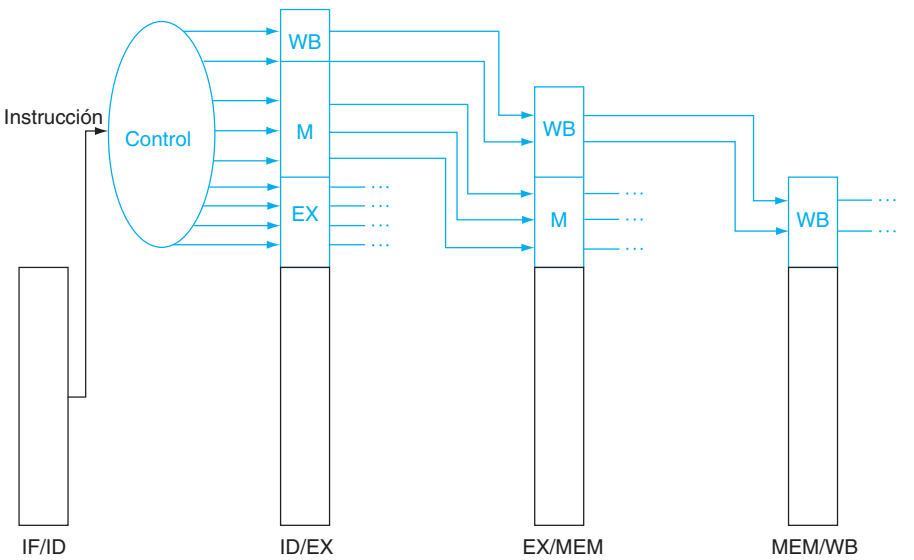


FIGURA 4.50 Líneas de control para las tres etapas finales. Observe que cuatro de las nueve líneas de control se usan en la etapa EX, mientras que las cinco restantes pasan al registro de segmentación EX/MEM, que ha sido extendido para poder almacenar las líneas de control; tres se usan durante la etapa MEM, y las dos últimas se pasan al registro MEM/WB para ser usadas en la etapa WB.

4. *Acceso a memoria*: Las líneas de control que se activan en esta etapa son Branch, MemRead y MemWrite. Estas señales se activan para las instrucciones `beq`, `load`, y `store` respectivamente. Recuerde que PCSrc en la figura 4.48 selecciona la dirección siguiente en orden secuencial a no ser que la lógica de control active la señal Branch y el resultado de la ALU sea cero.

5. *Escritura de resultado*: Las dos líneas de control son MemtoReg, la cual decide entre escribir en el banco de registros o bien el resultado de la ALU o bien el valor leído de memoria, y RegWrite, que escribe el valor escogido.

Ya que al segmentar el camino de datos no cambia el significado de las líneas de control, se pueden emplear los mismos valores para el control que antes. La figura 4.49 presenta los mismos valores que en la sección 4.4, pero ahora las nueve líneas de control se agrupan por etapas de segmentación.

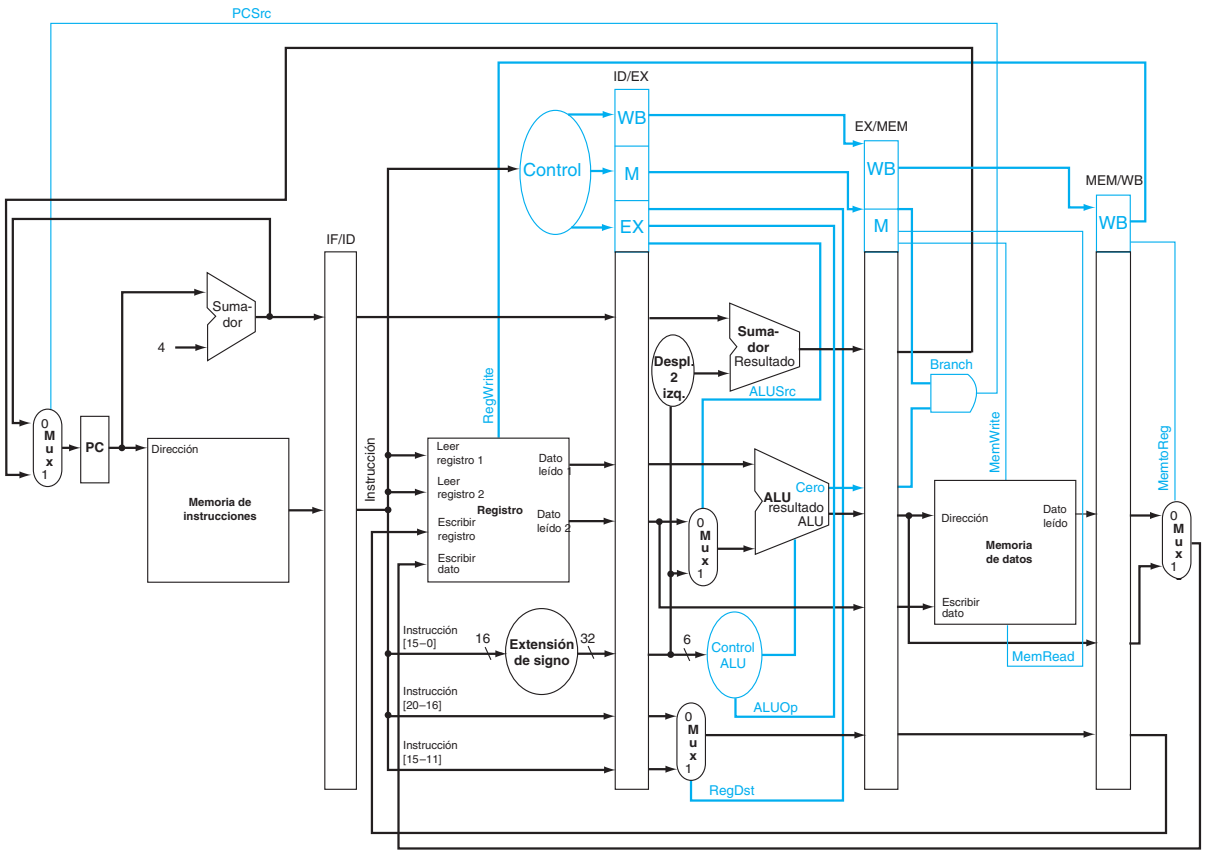


FIGURA 4.51 Camino de datos segmentado de la figura 4.40, con las señales de control conectadas a la parte de control de los registros de segmentación. Los valores de control de las tres últimas etapas se crean durante la descodificación de la instrucción y son escritos en el registro de segmentación ID/EX. En cada etapa de segmentación se usan ciertas líneas de control, y las líneas restantes se pasan a la etapa siguiente.

Realizar el control significa activar las nueve líneas de control a estos valores en cada etapa para cada instrucción. La manera más simple de hacerlo es extendiendo los registros de segmentación para incluir la información de control.

Ya que las líneas de control empiezan en la etapa EX, se puede crear la información de control durante la descodificación de la instrucción. La figura 4.50 muestra que estas señales de control se usan en la etapa de segmentación adecuada mientras la instrucción avanza por el pipeline, tal y como avanza el identificador de registro destino de las cargas en la figura 4.41. La figura 4.51 muestra el camino de datos completo con los registros de segmentación extendidos y con las líneas de control conectadas a la etapa correcta. (Si se quieren más detalles, la sección 4.12 tiene más diagramas monociclo con ejemplos de ejecución de códigos MIPS en el pipeline).

4.7 Riesgos de datos: anticipación frente a bloqueos

¿Qué quieres decir, por qué se debería construir? Es un baipás. Debes construir realimentaciones.

Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

Los ejemplos de la sección anterior muestran la potencia de la ejecución segmentada y cómo el hardware realiza esta tarea. Ahora es el momento de quitarse las gafas con cristales de color rosa y mirar qué pasa en programas reales. Las instrucciones de las figuras 4.43 a 4.45 eran independientes; ninguna de ellas usaba los resultados calculados por alguna de las anteriores. En cambio, en la sección 4.5 se vio que los riesgos de datos suponían un obstáculo para la ejecución segmentada.

Vamos a ver una secuencia de instrucciones con varias dependencias, mostradas en color.

```
sub   $2, $1,$3     # sub escribe en registro $2
and   $12,$2,$5     # 1er operando($2) depende de sub
or    $13,$6,$2     # 2º operando($2) depende de sub
add   $14,$2,$2     # 1er($2) y 2º($2) depende de sub
sw    $15,100($2)   # Base ($2) depende de sub
```

Las últimas cuatro instrucciones dependen todas del resultado de la primera instrucción, guardado en el registro `$2`. Si este registro tenía el valor 10 antes de la instrucción de resta y el valor -20 después, la intención del programador es que el valor –20 sea usado por las instrucciones posteriores que referencian al registro `$2`.

¿Cómo se ejecutaría esta secuencia de instrucciones en el procesador segmentado? La figura 4.52 ilustra la ejecución de estas instrucciones usando una representación multiciclo. Para mostrar la ejecución de esta secuencia de instrucciones en el pipeline, la parte superior de la figura 4.52 muestra el valor del registro `$2`, que cambia en la mitad del quinto ciclo, cuando la instrucción `sub` escribe su resultado.

Uno de los riesgos potenciales se puede resolver con el propio diseño del hardware del banco de registros: ¿qué ocurre cuando un registro se lee y se escribe en el mismo ciclo de reloj? Se supone que la escritura se hace en la primera mitad del ciclo y la lectura se hace en la segunda mitad, por lo que la lectura proporciona el valor que acaba de ser escrito. Como esto ya lo hacen muchas implementaciones de bancos de registros, en este caso entenderemos que no hay riesgo de datos.

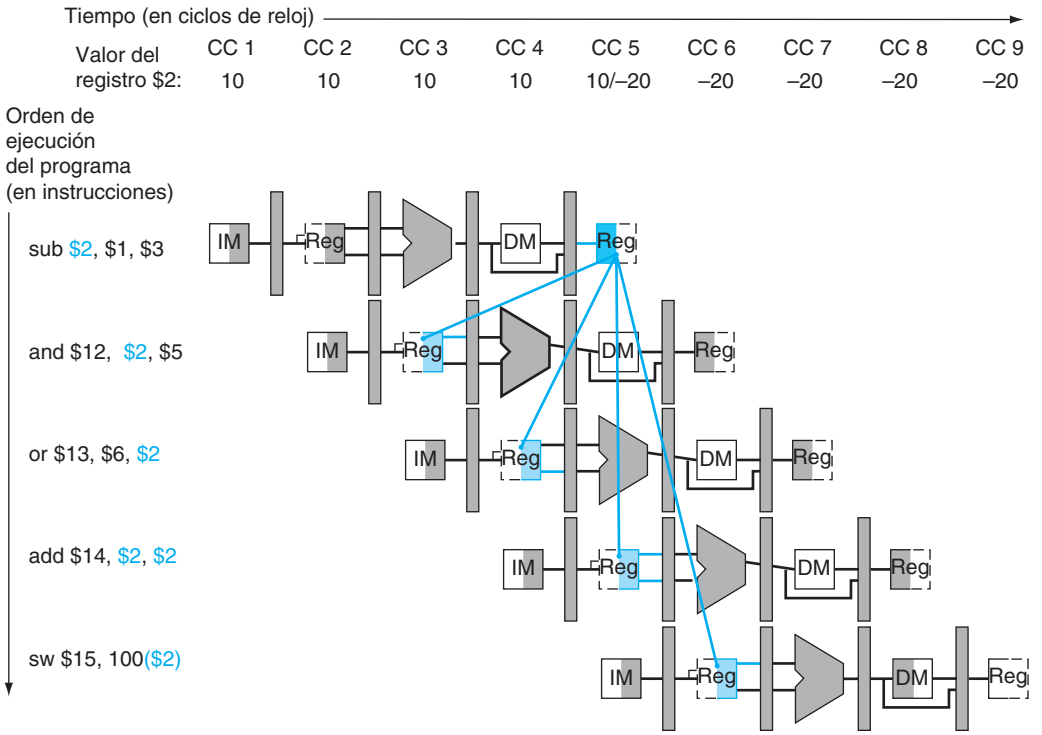


FIGURA 4.52 Dependencias en la segmentación de la ejecución de la secuencia de cinco instrucciones usando caminos de datos simplificados para mostrar las dependencias. Todas las acciones dependientes se muestran en color y "CC *i*" en la parte superior de la figura representa el ciclo de reloj *i*. La primera instrucción escribe en `$2`, y todas las siguientes instrucciones leen de `$2`. Este registro se escribe en el ciclo 5, por lo que el valor correcto no está disponible antes del ciclo 5. (La lectura de un registro durante un ciclo de reloj retornará el valor escrito al final de la primera mitad del ciclo, si es que esa escritura se produce). Las líneas coloreadas desde la parte superior del camino de datos a la parte inferior muestran las dependencias. Aquellas que deben ir hacia atrás en el tiempo constituyen los *riesgos de datos en el pipeline.*

La figura 4.52 muestra que los valores que se leen del registro `$2` *no* representarían el resultado de la instrucción `sub` a menos que la lectura del registro se hiciera durante el ciclo 5 o después. Las instrucciones que obtendrían el valor correcto de -20 son `add` y `sw`; las instrucciones `and` y `or` obtendrían el valor 10, que es incorrecto. Al utilizar este estilo de dibujo estos problemas se ven claramente porque una línea de dependencia va hacia atrás en el tiempo.

Como se ha mencionado en la sección 4.5, el resultado está disponible al final de la etapa EX, o lo que es lo mismo al final del tercer ciclo. ¿Cuándo se necesita realmente ese dato para las instrucciones `and` y `or`? Al principio de la etapa EX, o lo que es lo mismo en los ciclos 4 y 5, respectivamente. Por tanto, podemos ejecutar este segmento sin bloqueos si simplemente anticipamos los datos tan pronto como estén disponibles a cualquiera de las unidades que necesiten el dato antes de que esté disponible en el banco de registros para ser leído.

¿Cómo funciona la anticipación de resultados? Por simplicidad, en el resto de esta sección se considerará la posibilidad de anticipar datos sólo a las operaciones que están en la etapa EX, que pueden ser operaciones de tipo ALU o el cálculo de

una dirección efectiva. Esto significa que cuando una instrucción trata de usar en su etapa EX el registro que una instrucción anterior intenta escribir en su etapa WB, lo que realmente se necesita es disponer de su valor como entrada a la ALU.

Una notación que ponga nombre a los campos de los registros de segmentación permite una descripción más precisa de las dependencias. Por ejemplo, "ID/EX.RegisterRs" se refiere al identificador de un registro cuyo valor se encuentra en el registro de segmentación ID/EX; esto es, el que viene del primer puerto de lectura del banco de registros. La primera parte del nombre, a la izquierda, es el identificador del registro de segmentación; la segunda parte es el nombre del campo de ese registro. Usando esta notación, existen dos parejas de condiciones para detectar riesgos:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

El primer riesgo en la secuencia de la página 363 se encuentra en el registro `$2`, entre el resultado de `sub $2, $1, $3` y el primer operando de lectura de la instrucción `and $12, $2, $5`. Este riesgo se puede detectar cuando la instrucción `and` está en la etapa EX y la otra instrucción está en la etapa MEM, por lo que corresponde con la condición 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs = `$2`

Detección de dependencias

EJEMPLO

Clasificar las dependencias en esta secuencia de la página 363:

```
sub    $2,    $1, $3   # Registro $2 escrito por sub
and    $12,   $2, $5   # 1er operando($2) escrito por sub
or     $13,   $6, $2   # 2º operando($2) escrito por sub
add    $14,   $2, $2   # 1er($2) y 2º($2) escrito por sub
sw     $15,   100($2) # Índice($2) escrito por sub
```

RESPUESTA

Tal como se ha mencionado anteriormente, el riesgo `sub-add` es de tipo 1a. Los riesgos restantes son:

- El riesgo `sub-or` es de tipo 2b:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = `$2`

- Las dos dependencias en `sub-add` no son riesgos ya que el banco de registros proporciona el valor correcto durante la etapa ID de `add`.
- No hay riesgo de datos entre `sub` y `sw` ya que `sw` lee `$2` un ciclo de reloj *después* que `sub` escriba `$2`.

Ya que algunas instrucciones no escriben en registros, esta política es poco precisa; algunas veces se anticiparía un valor innecesario. Una solución consiste sencillamente en comprobar si la señal RegWrite estará activa: para ello se examina el campo de control WB del registro de segmentación durante las etapas de EX y MEM. Además, MIPS requiere que cada vez que se use como operando el registro `$0`, se debe producir un valor para el operando igual a cero. En el caso que una instrucción tenga como destino `$0` (por ejemplo, `sll $0, $1, $2`), se debe evitar la anticipación de un resultado que posiblemente sea diferente de cero. El no anticipar resultados destinados a `$0` libera de restricciones al programador de ensamblador y al compilador para usar `$0` como registro destino. Por lo tanto, las condiciones mencionadas antes funcionarán correctamente siempre que se añada EX/MEM.RegisterRd ≠ 0 a la primera condición de riesgo y MEM/WB.RegisterRd ≠ 0 a la segunda condición.

Una vez detectados los riesgos, la mitad del problema está resuelto, pero todavía falta anticipar el dato correcto.

La figura 4.53 muestra las dependencias entre los registros de segmentación y las entradas de la ALU para la misma secuencia de código de la figura 4.52. El cambio radica en que la dependencia empieza en un registro de segmentación en lugar de esperar a que en la etapa WB se escriba en el banco de registros. Por lo tanto, el dato que se tiene que adelantar ya está disponible en los registros de segmentación con tiempo suficiente para las instrucciones posteriores.

Si se pudieran obtener las entradas de la ALU de *cualquier* registro de segmentación en vez de sólo del registro de segmentación ID/EX, entonces se podría adelantar el dato correcto. Bastaría con añadir multiplexores adicionales en la entrada de la ALU, y el control apropiado para poder ejecutar el pipeline a máxima velocidad en presencia de estas dependencias.

Por ahora, supondremos que las únicas instrucciones que necesitan avanzar su resultado son las cuatro de tipo R: `add`, `sub`, `and` y `or`. La figura 4.54 muestra un primer plano de la ALU y de los registros de segmentación antes y después de añadir la anticipación de datos. La figura 4.55 muestra los valores de las líneas de control de los multiplexores de la ALU que seleccionan bien los valores del banco de registros, o bien uno de los valores anticipados.

El control de la anticipación estará en la etapa EX, ya que los multiplexores de anticipación previos a la ALU se encuentran en esta etapa. Por lo tanto se deben pasar los identificadores de los registros fuente desde la etapa ID a través del registro de segmentación ID/EX para determinar si se deben adelantar los valores. El campo rt ya se tiene (bits 20-16). Antes de la anticipación, el registro ID/EX no necesitaba incluir espacio para guardar el campo rs. Por lo tanto se debe añadir rs (bits 25-21) al registro ID/EX.

Ahora escribiremos tanto las condiciones para detectar riesgos como las señales de control para resolverlos:

1. Riesgo EX:

```
si (EX/MEM.RegWrite
y (EX/MEM.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
si (EX/MEM.RegWrite
y (EX/MEM.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

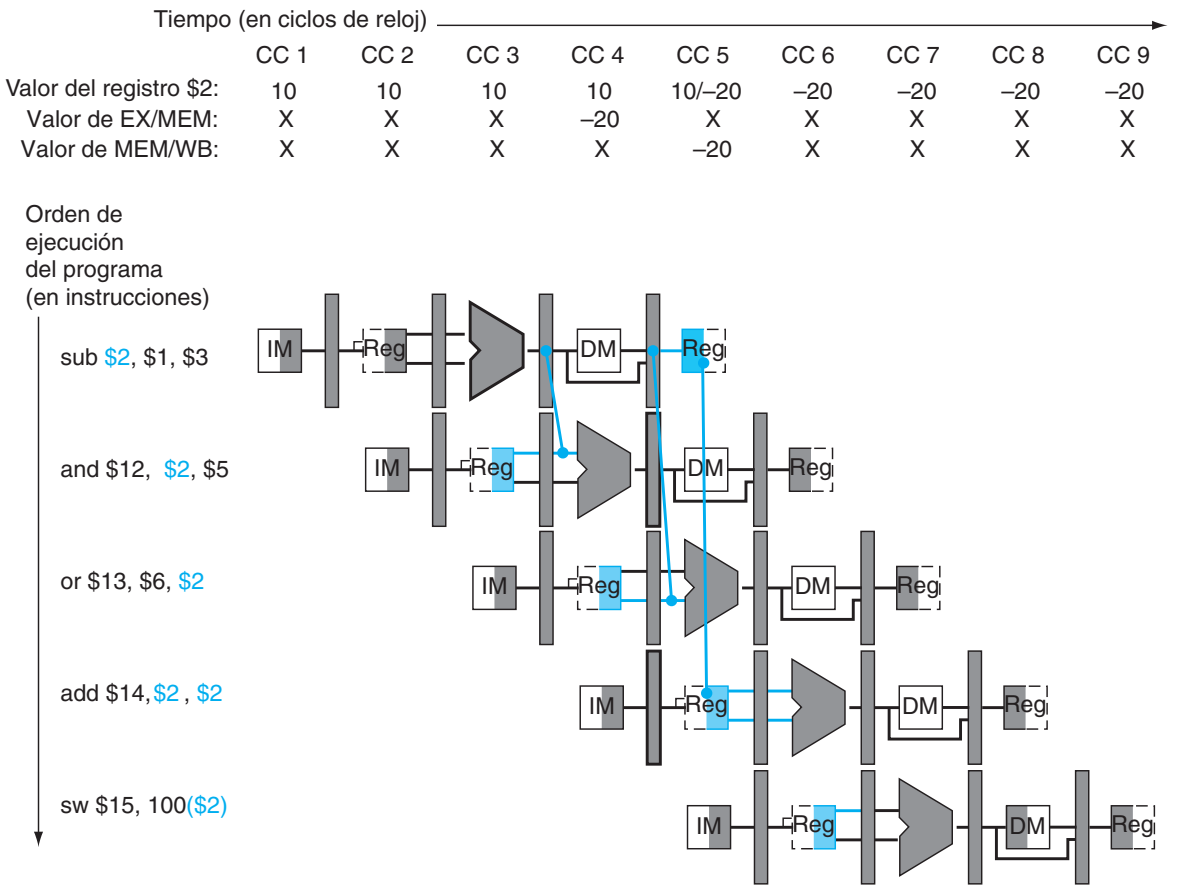


FIGURA 4.53 Las dependencias entre los registros de segmentación se mueven hacia adelante en el tiempo, por lo que es posible proporcionar a la ALU los operandos que necesitan las instrucciones `and` y `or` anticipando los resultados que se encuentran en dichos registros. Los valores en los registros de segmentación muestran que el valor deseado está disponible antes de que se escriba en el banco de registros. Se supone que el banco de registros adelanta el valor que se lee y escribe durante el mismo ciclo de reloj, por lo que la instrucción `add` no se debe bloquear, ya que los valores le vienen desde el banco de registros en vez de desde un registro de segmentación. La anticipación de datos dentro del banco de registros -esto es, la lectura obtiene el valor que se escribe en ese mismo ciclo- explica por qué el ciclo 5 muestra al registro `$2` con el valor 1 al principio y con el valor -20 al final del ciclo de reloj. Como en el resto de la sección, consideraremos todos los casos de anticipación de datos excepto para los valores que deben ser guardados en memoria en el caso de las instrucciones de almacenamiento.

Observe que el campo EX/MEM.RegisterRd es el registro destino tanto para una instrucción ALU (que proviene del campo Rd de la instrucción) como para una carga (que proviene del campo Rt).

Este caso adelanta el resultado desde una instrucción anterior a cualquiera de las entradas de la ALU. Si la instrucción previa va a escribir en el banco de registros y el identificador del registro destino es igual al identificador de registro de lectura A o B en la entrada de la ALU, suponiendo que no es el registro 0, entonces se hace que el multiplexor tome el valor del registro de segmentación EX/MEM.

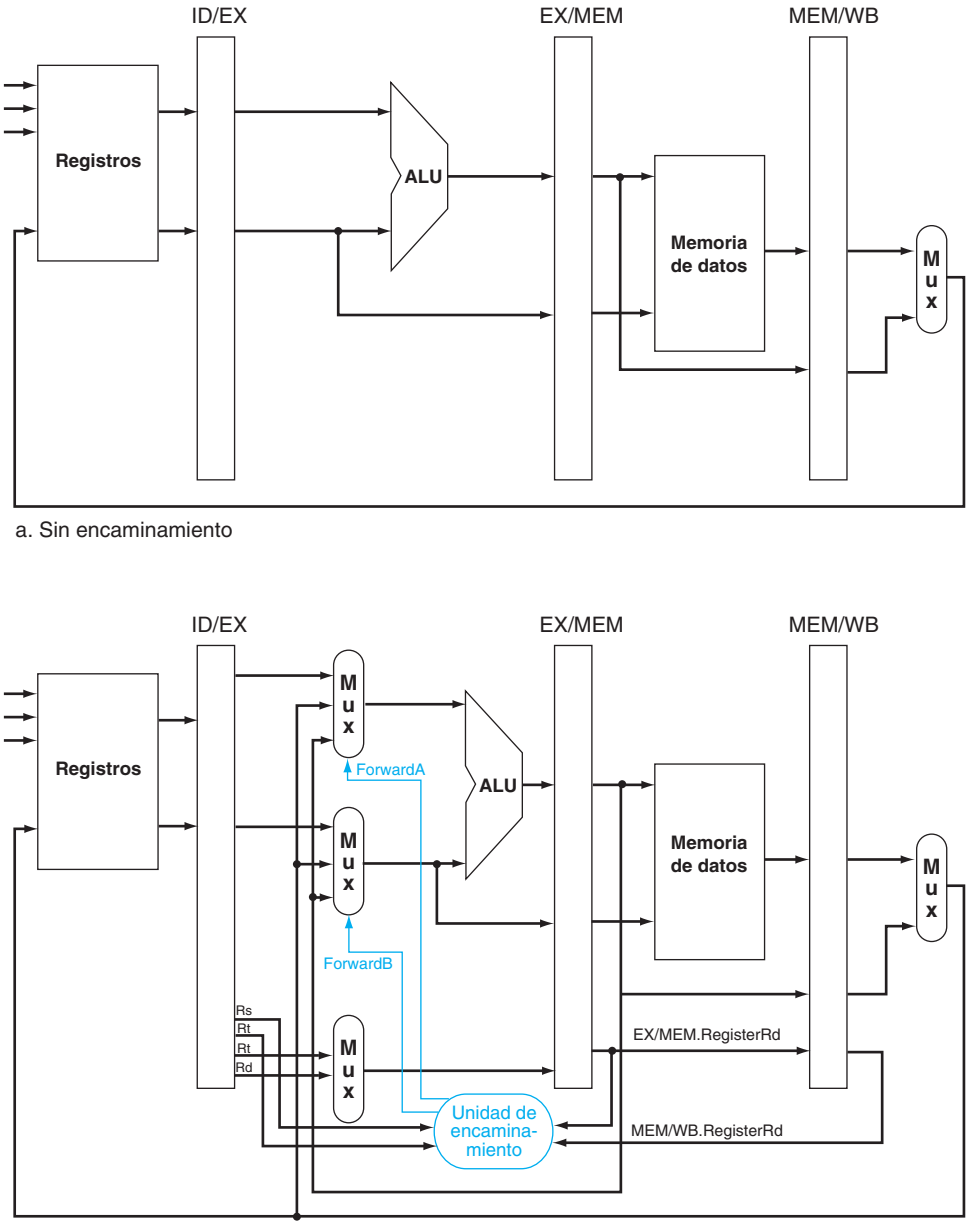


FIGURA 4.54 En la parte superior se encuentra la ALU y los registros de segmentación antes de añadir la anticipación de resultados. En la parte inferior, los multiplexores se han expandido para añadir los caminos de anticipación, y también se muestra la unidad de anticipación. El hardware nuevo se muestra en color. Esta figura es, sin embargo, un dibujo simplificado que deja fuera detalles del camino de datos completo, como por ejemplo el hardware de extensión de signo. Observe que el campo ID/EX.RegisterRt se muestra dos veces, una vez conectado al multiplexor y otra vez conectado a la unidad de anticipación, pero es una sola señal. Como en la discusión anterior, se ignora la anticipación de datos para el valor que debe ser guardado en memoria en el caso de las instrucciones de almacenamiento. Obsérvese que esta técnica funciona también para instrucciones `slt`.

2. Riesgo MEM:

```
si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Como ya hemos mencionado anteriormente, no hay riesgo en la etapa WB, ya que se supone que el banco de registros proporciona el valor correcto si la instrucción en la etapa ID lee el mismo registro que escribe la instrucción situada en la etapa WB. Un banco de registros de tal funcionalidad realiza otro tipo de anticipación, pero ocurre dentro del propio banco de registros.

Una posible complicación consiste en tener riesgos de datos potenciales entre el resultado de la instrucción situada en WB, el resultado de la instrucción en la etapa MEM y el operando fuente de la instrucción en la etapa de ALU. Por ejemplo, cuando se suma un vector de números en un único registro, todas las instrucciones de la secuencia leerán y escribirán en el mismo registro:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

En este caso, el resultado se anticipa desde la etapa MEM ya que el resultado en esta etapa es el más reciente. Por lo tanto, el control del riesgo en la etapa MEM sería (con los cambios adicionales resaltados):

```
si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
y (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
y (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

La figura 4.56 muestra el hardware necesario para implementar la anticipación de datos con las operaciones que usan resultados en la etapa EX.(20) Observe que el campo EX/MEM.RegisterRd es el registro destino de tanto de una instrucción de

Control del multiplexor	Fuente	Explicación
ForwardA = 00	ID/EX	EL primer operando de la ALU viene del banco de registros
ForwardA = 10	EX/MEM	EL primer operando de la ALU se anticipa del resultado anterior de la ALU
ForwardA = 01	MEM/WB	EL primer operando de la ALU se anticipa de la memoria de datos o de un resultado de la ALU anterior
ForwardB = 00	ID/EX	EL segundo operando de la ALU viene del banco de registros
ForwardB = 10	EX/MEM	EL segundo operando de la ALU se anticipa del resultado anterior de la ALU
ForwardB = 01	MEM/WB	EL segundo operando de la ALU se anticipa de la memoria de datos o de un resultado de la ALU anterior

FIGURA 4.55 Los valores de control para los multiplexores de anticipación de datos de la figura 4.54. El valor inmediato con signo que es otra entrada de la ALU se describe en la Extensión que se encuentra al final de esta sección.

la ALU (que se obtiene del campo Rd de la instrucción) como de una carga (que se obtiene del campo Rt).

Para más detalles, la sección 4.12 en el CD muestra dos fragmentos de código MIPS con riesgos y anticipación, ilustrados con diagramas monociclo.

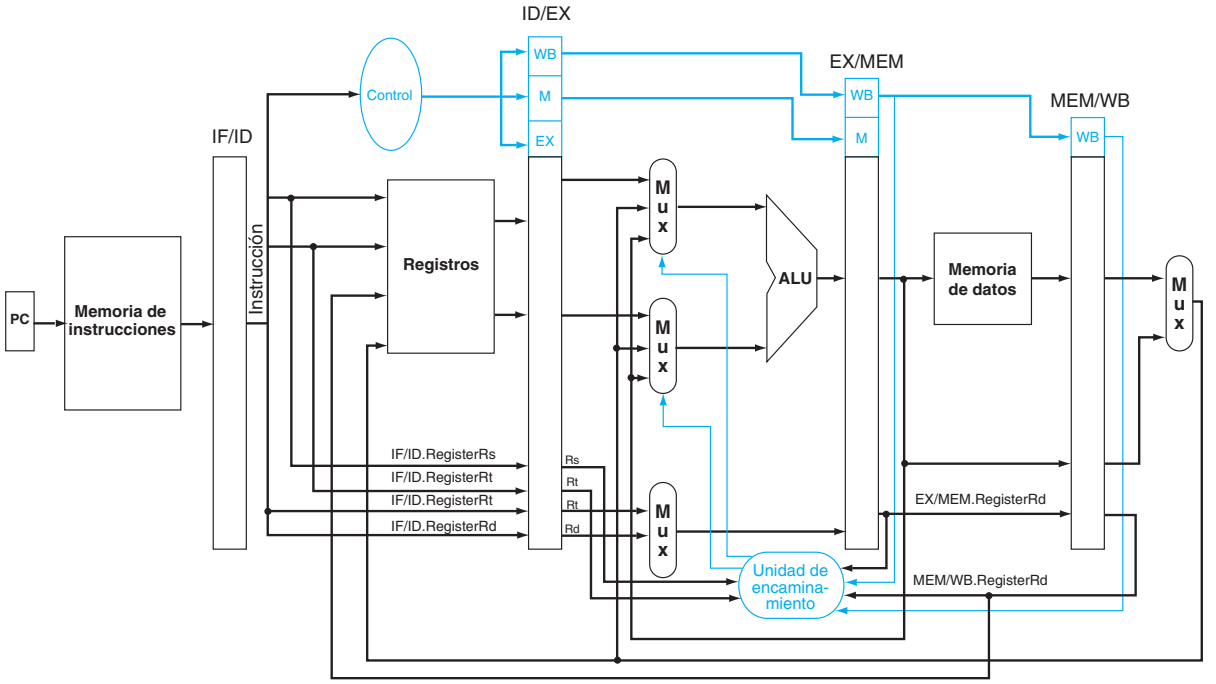


FIGURA 4.56 Camino de datos modificado para resolver los riesgos a través la anticipación de resultados. Comparado con el camino de datos de la figura 4.51, los cambios adicionales son los multiplexores en las entradas de la ALU. Sin embargo, esta figura simplificada deja fuera detalles del camino de datos completo, como por ejemplo el hardware para los saltos y para la extensión de signo.

Extensión: La anticipación de datos también puede ayudar con los riesgos cuando las instrucciones de almacenamiento en memoria dependen de otras instrucciones. En este caso la anticipación es sencilla, ya que las instrucciones sólo usan un valor durante la etapa MEM. Pero considere cargas seguidas inmediatamente de almacenamientos, útiles para realizar copias memoria-a-memoria en la arquitectura MIPS. Se necesita añadir más hardware de anticipación de datos para hacer que se ejecuten más rápido. Si se dibujara otra vez la figura 4.53, reemplazando las instrucciones `sub` y `and` por `lw` y `sw`, se vería que es posible evitar el bloqueo, ya que el dato está en el registro MEM/WB de la instrucción de carga con tiempo para que sea usado en la etapa MEM de la instrucción de almacenamiento. Para esta opción, se necesitaría añadir el hardware de anticipación de datos en la etapa de acceso a memoria. Se deja esta modificación en el camino de datos como ejercicio.

Por otra parte, el valor inmediato con signo extendido que necesitan las cargas y los almacenamientos en la entrada de la ALU no está en el camino de datos de la figura 4.56. Debido a que el control central decide entre el valor del registro o el inmediato, y debido a que la unidad de anticipación de datos elige el registro de segmentación que irá a la entrada de la ALU, la solución más fácil es añadir un multiplexor 2:1 que escoja entre la salida del multiplexor ForwardB y el valor inmediato con signo. La figura 4.57 muestra este cambio adicional.

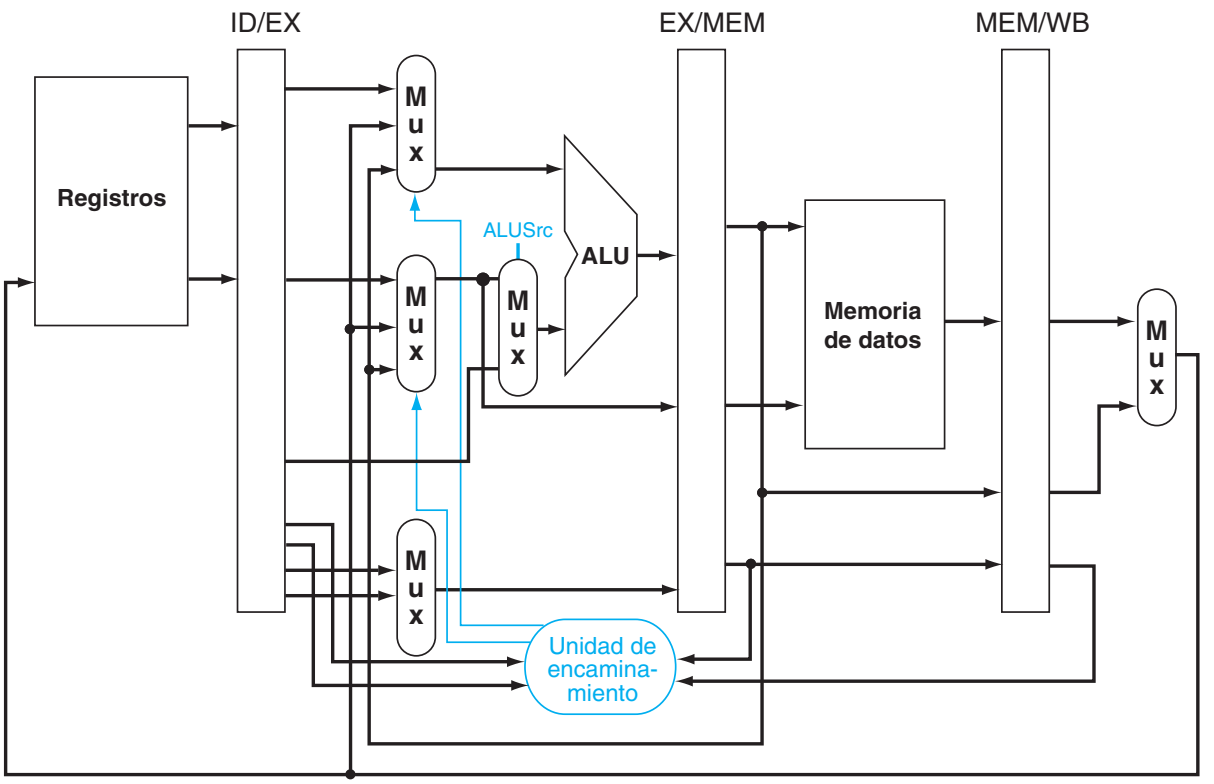


FIGURA 4.57 El zoom del camino de datos de la figura 4.54 muestra un multiplexor 2:1, añadido para seleccionar como entrada de la ALU al valor inmediato con signo.

Si al principio no tienes éxito, redefine el éxito.
Anónimo

Riesgos de datos y bloqueos

Como se ha dicho en la sección 4.5, un caso donde la anticipación de resultados no nos puede resolver la papeleta es cuando una instrucción situada después de una carga intenta leer el registro en el que escribe la carga. La figura 4.58 ilustra el problema. El dato todavía se está leyendo de memoria en el ciclo 4 mientras la ALU está ejecutando la operación de la siguiente instrucción. Es necesario que se bloquee el pipeline para la combinación de una carga seguido de una instrucción que lee su resultado.

Por lo tanto, además de una unidad de anticipación de datos, se necesita una unidad de detección de riesgos. Debe funcionar durante la etapa ID de tal manera que pueda insertar un bloqueo entre el *load* y la instrucción que lo usa. El control para la detección de riesgos cuando se tiene que comprobar una instrucción *load* es la siguiente condición:

```
si (ID/EX.MemRead y
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) o
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      bloquear el pipeline
```

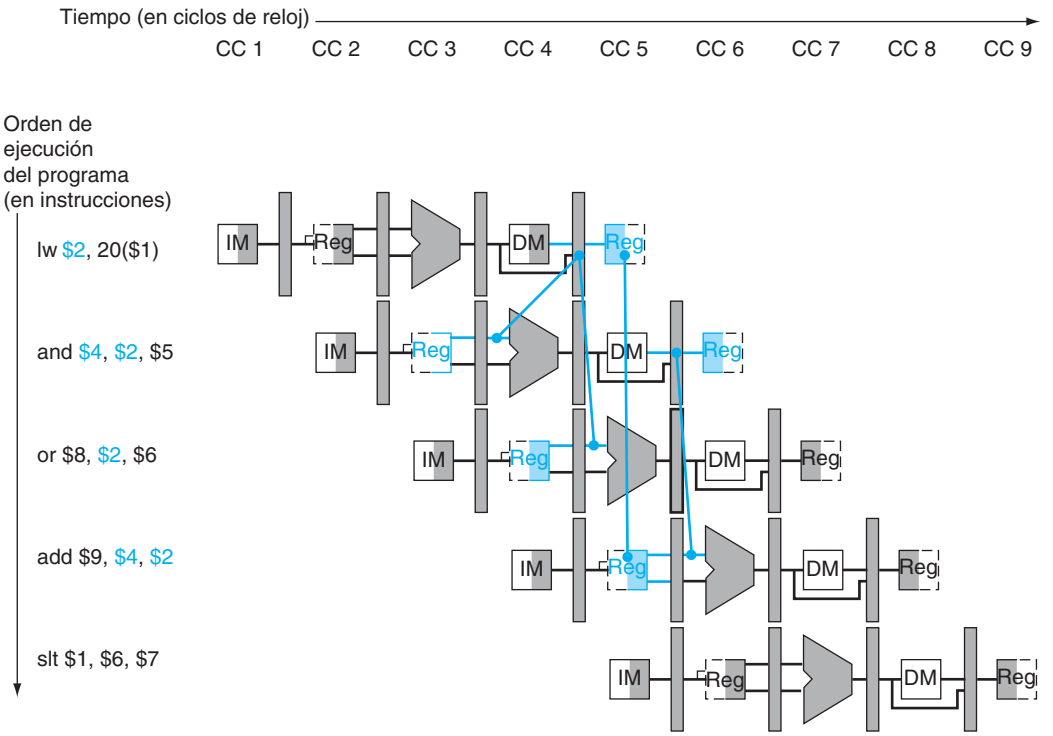


FIGURA 4.58 Secuencia de instrucciones segmentadas. Como la dependencia entre la carga y la instrucción siguiente (`and`) va hacia atrás en el tiempo, este riesgo no puede ser resuelto mediante la anticipación de datos. Por tanto, esta combinación debe resultar en un bloqueo generado por la unidad de detección de riesgos.

La primera línea comprueba si la instrucción es una carga: ésta es la única instrucción que lee de memoria. Las dos líneas siguientes comprueban si el campo de registro destino de la carga en la etapa EX es igual a cualquiera de los dos registros fuente de la instrucción en ID. Si la condición se cumple, la instrucción se bloquea durante 1 ciclo. Después de este bloqueo de un ciclo, la lógica de anticipación de datos puede resolver la dependencia y la ejecución continúa. (Si no hubiera anticipación de datos, entonces las instrucciones en la figura 4.58 tendrían que bloquearse otro ciclo).

Si la instrucción situada en la etapa ID se bloquea, entonces también debe bloquearse la que esté en IF; de no hacerse así se perdería la instrucción buscada de memoria. Evitar que estas dos instrucciones progresen en el procesador supone simplemente evitar que cambien el registro PC y el registro de segmentación IF/ID. Si estos valores no cambian, la instrucción en la etapa IF continuará leyendo de memoria usando el mismo PC y se volverán a leer los registros en la etapa ID usando los mismos campos de instrucción en el registro de segmentación IF/ID. Volviendo a nuestra analogía favorita, es como si la lavadora volviera a empezar con la misma ropa y se dejara a la secadora que diera vueltas vacía. Por supuesto, igual que con la secadora, la mitad posterior de pipeline que comienza en la etapa EX debe hacer alguna cosa. Lo que hace es ejecutar instrucciones que no tienen ningún efecto: **nops**.

nops: instrucción que no realiza ninguna operación para cambiar el estado. Viene de "no operación".

¿Cómo se pueden insertar estos nops, que actúan como burbujas en el pipeline? En la figura 4.49 se observa que negando las nueve señales de control (poniéndolas a 0) en las etapas EX/MEM y WB se creará una instrucción "no hacer nada" o nop. Al identificar el riesgo en la etapa ID, se puede insertar una burbuja en el pipeline poniendo a 0 los campos de control de EX, MEM y WB en el registro de segmentación ID/EX. Estos nuevos valores de control se propagan hacia adelante en cada ciclo con el efecto deseado: si los valores de control valen todos 0 no se escribe sobre memoria o sobre registros.

La figura 4.59 muestra lo que realmente sucede en el hadware: el segmento de ejecución asociado a la instrucción `and` se convierte en un nop, y todas las instrucciones a partir de `and` son retrasadas un ciclo. Como una burbuja de aire en una tubería de agua, una burbuja de bloqueo retrasa todo lo que viene detrás, y avanza por el pipeline hasta salir por su extremo final. El riesgo fuerza a que las instrucciones `and` y `or` repitan en el ciclo 4 lo que hicieron en el 3: `and` lee los registros y se descodifica, `or` se busca de nuevo en la memoria de instrucciones. Es realmente esta repetición de trabajo lo que realiza un bloqueo, pero su efecto es alargar el tiempo de las instrucciones `and` y `or`, y retrasar la búsqueda de la instrucción `add`.

La figura 4.60 resalta las conexiones en el pipeline tanto para la unidad de detección de riesgos como para la unidad de anticipación de datos. Como antes, la unidad de anticipación controla los multiplexores de la ALU para reemplazar el valor del registro de propósito general por el valor del registro de segmentación adecuado. La unidad de detección de riesgos controla la escritura sobre los registros PC e IF/ID, además del multiplexor que elige entre los valores de control reales o todo ceros. La unidad de riesgos bloquea y desactiva los campos de control si es cierta la comprobación del riesgo de uso de una carga (*load-use hazard*) antes

mencionado. Para más detalles, la sección 4.12 en el CD muestra un fragmento de código MIPS con riesgos que causan bloqueos del pipeline, ilustrado con diagramas monociclo.

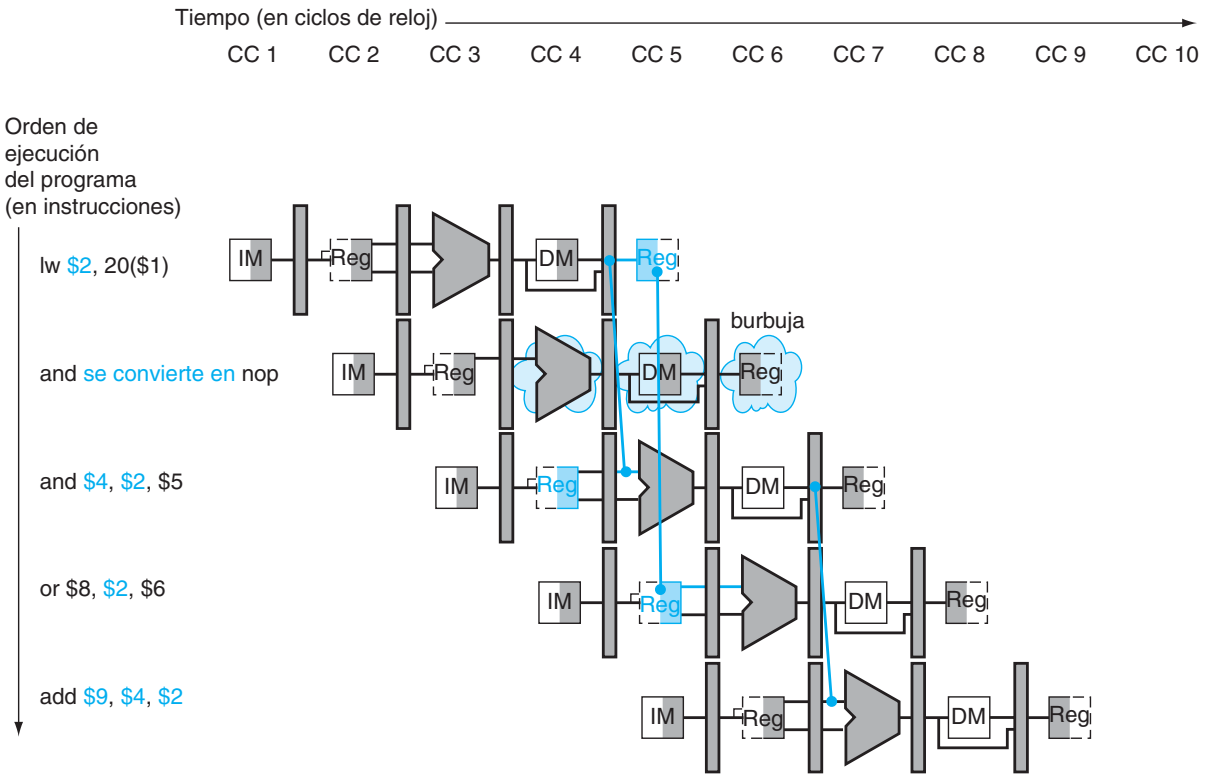


FIGURA 4.59 Como se insertan realmente los bloqueos en el pipeline. Se inserta una burbuja al principio del ciclo 4 cambiando la instrucción `and` por un `nop`. Observe que la instrucción `and` es realmente buscada y descodificada en los ciclos 2 y 3, pero su etapa EX es retrasada hasta el ciclo 5 (en lugar del ciclo 4 cuando no se produce bloqueo). De un modo similar, la instrucción `or` se busca en el ciclo 4, pero su etapa IF es retrasada hasta el ciclo 5 (en lugar del ciclo 4 cuando no se produce bloqueo). Después de insertar la burbuja, todas las dependencias avanzan en el tiempo y no ocurren más riesgos de datos.

IDEA clave

El hardware puede depender del compilador o no para resolver los riesgos de dependencias y asegurar una ejecución correcta. Sea como sea, para conseguir las mejores prestaciones es necesario que el compilador comprenda el pipeline. De lo contrario, los bloqueos inesperados reducirán las prestaciones del código compilado.

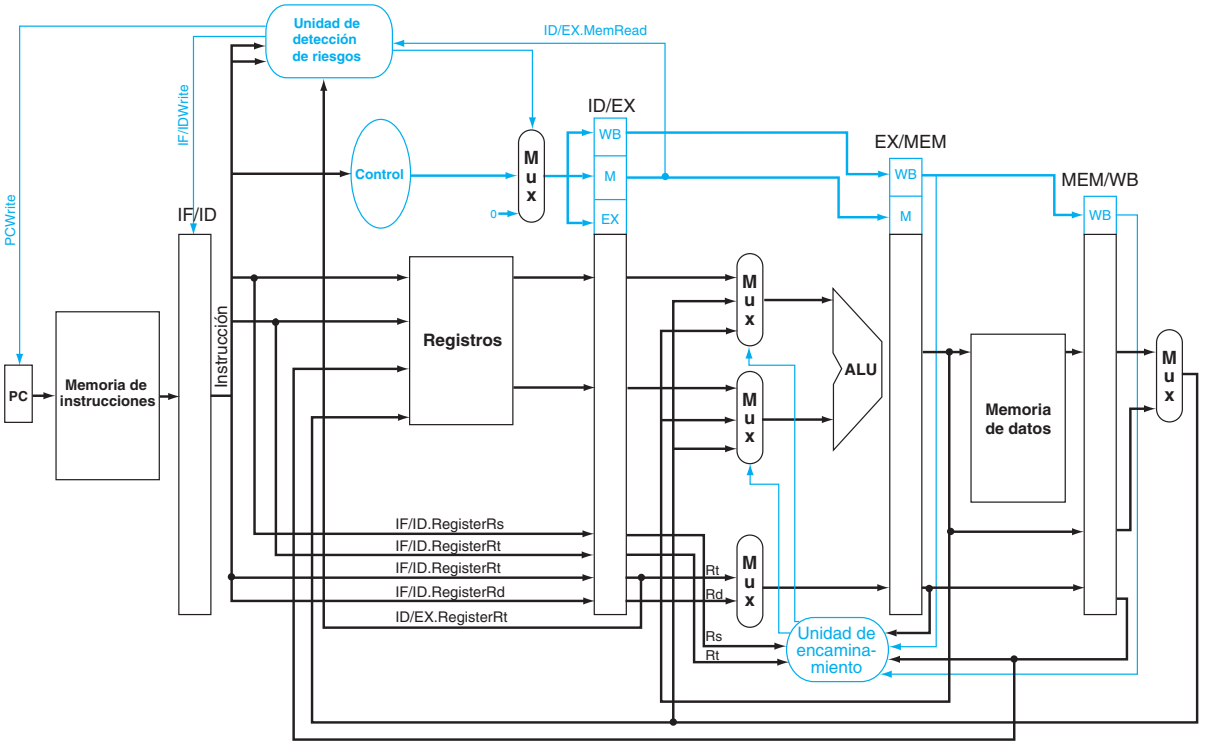


FIGURA 4.60 Perspectiva general del control de la segmentación, mostrando los dos multiplexores para la anticipación de datos, la unidad de detección de riesgos y la unidad de anticipación de datos. A pesar de que las etapas ID y EX se han simplificado (faltan la extensión de signo del valor inmediato y lógica de saltos) este dibujo muestra los requerimientos esenciales para el hardware de anticipación de datos.

Extensión: Con respecto al comentario anterior acerca de poner las líneas de control a 0 para evitar la escritura sobre registros o memoria: sólo se necesita que valgan 0 las señales RegWrite y MemWrite, mientras que el resto de las señales de control pueden tener un valor "indiferente".

4.8 Riesgos de control

Hay miles que atacan las ramas del mal por cada uno que golpea en su raíz.

Henry David Thoreau, *Walden*, 1854

Hasta ahora hemos centrado nuestra atención en los riesgos que implican operaciones aritméticas y de transferencia de datos. Pero como ya vimos en la sección 4.5, también existen riesgos en el pipeline en los que están involucrados los saltos. La figura 4.61 muestra una secuencia de instrucciones y señala qué pasa cuando entra un salto en el pipeline. En cada ciclo de reloj se debe buscar una nueva instrucción para poder mantener el pipeline ocupado, pero en nuestro

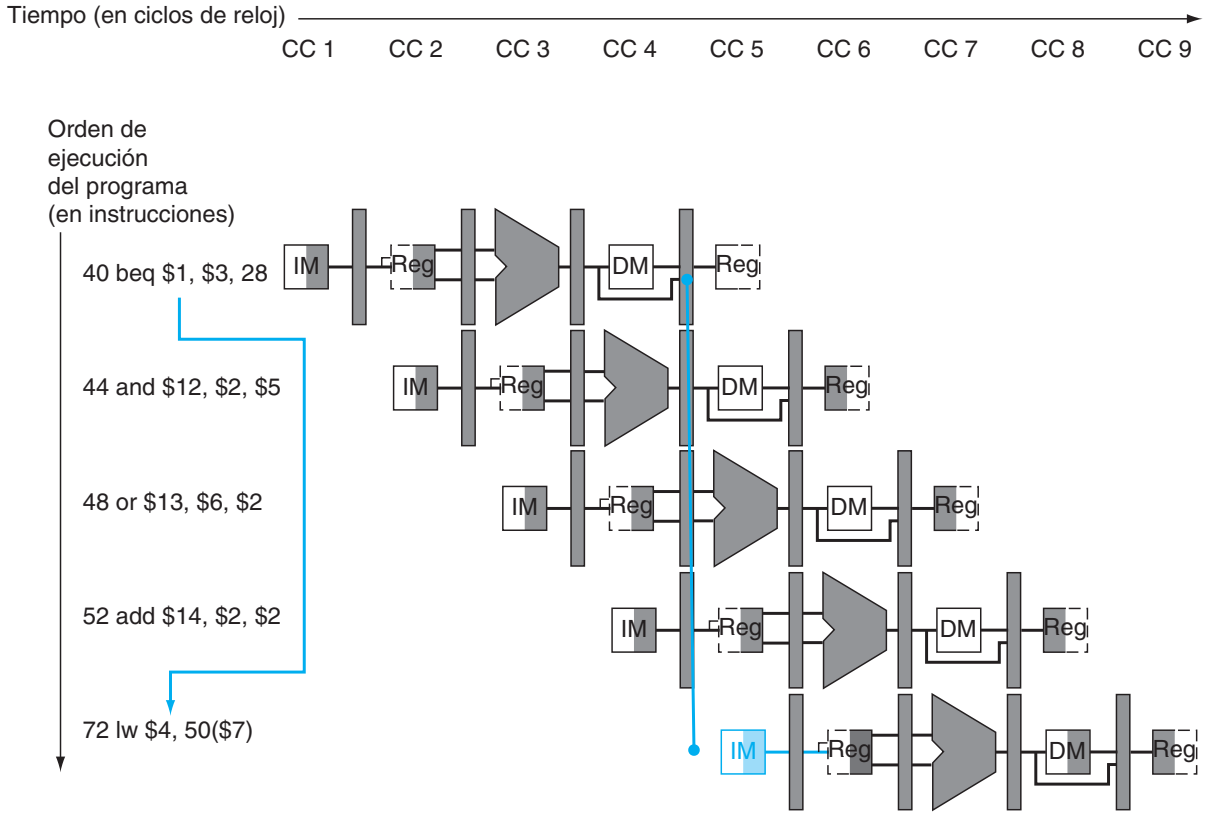


FIGURA 4.61 Impacto de la segmentación en las instrucciones de salto. Los números a la izquierda de las instrucciones (40, 44,...). son sus direcciones. Como una instrucción de salto no toma la decisión de saltar o no saltar hasta la etapa MEM (ciclo 4 para la instrucción `beq` de arriba), las tres instrucciones que siguen al salto serán leídas de memoria y se empezará su ejecución. Si no se interviene de algún modo, esas tres instrucciones empezarán la ejecución antes que `beq` salte a la instrucción `lw` de la posición 72. (En la figura 4.31 se suponía la existencia de circuitería adicional para reducir los riesgos de control a un único ciclo; esta figura utiliza el camino de datos original, sin optimizar).

diseño no se conoce la decisión del salto hasta la etapa MEM. Como se ha mencionado en la sección 4.5, este retardo a la hora de determinar la instrucción correcta que se tiene que buscar se llama riesgo de control o riesgo de saltos, en contraposición con los riesgos de datos que se acaban de estudiar.

Esta sección, que trata sobre los riesgos de control, es más corta que las secciones previas sobre los riesgos de datos. Las razones son que los riesgos de control son relativamente más simples de entender, que aparecen con menos frecuencia que los riesgos de datos y que no existe un mecanismo tan efectivo contra los riesgos de control como lo es la anticipación de resultados para los riesgos de datos. Por tanto, se usarán esquemas más simples. Se estudiarán dos esquemas para resolver los riesgos de control y una optimización para mejorar estos esquemas.

Suponer el salto como no tomado

Como vimos en la sección 4.5, bloquear el procesador hasta que se complete el salto es demasiado lento. Para evitar el bloqueo en los saltos, habitualmente se realiza una mejora que es suponer que el salto no será tomado y por lo tanto la ejecución continúa siguiendo el flujo secuencial de instrucciones. Si el salto es finalmente tomado, se deberá descartar las instrucciones que se están buscando y descodificando en ese momento, y la ejecución continuará a partir del destino del salto. Si la mitad de las veces los saltos son no tomados, y si descartar instrucciones cuesta poco, esta optimización reduce el coste de los riesgos de control a la mitad.

Para descartar instrucciones, basta con cambiar el valor de las señales de control orginales y ponerlas a cero, de una forma muy parecida a como se bloquearon las instrucciones para evitar el riesgo de datos de tipo *load-use*. La diferencia estriba en que cuando el salto llega a la etapa de MEM se deben cambiar las señales de control para tres instrucciones, las que hay en las etapas IF, ID y EX; en cambio, en el caso de los bloqueos por uso del dato de una carga, sólo se ponía el control a 0 en la etapa ID y se le dejaba propagar por el pipeline. Por lo tanto, en este caso **descartar intrucciones (*flush instructions*)** significa desechar las instrucciones que haya en las etapas IF, ID y EX.

Descartar instrucciones: descartar instrucciones de un pipeline, generalmente debido a un suceso imprevisto.

Reducción del retardo de los saltos

Una manera de mejorar el rendimiento de los saltos es reducir el coste de los saltos tomados. Hasta ahora se ha supuesto que el siguiente valor de PC en el caso de un salto se selecciona en la etapa de MEM, pero si se mueve la ejecución del salto a una etapa anterior del pipeline, entonces se tienen que eliminar menos instrucciones. La arquitectura MIPS fue diseñada para dar soporte a saltos rápidos de un único ciclo de reloj que pudieran ser segmentados con una penalización pequeña. Los diseñadores observaron que muchos de los saltos dependían de comprobaciones muy simples (igualdad o signo, por ejemplo) y que estas comprobaciones no requieren una operación completa en la ALU y pueden hacerse con muy pocas puertas lógicas. Cuando se necesita tomar una decisión de salto más compleja, entonces se debe utilizar una instrucción distinta que realice la comparación requerida mediante la ALU; una situación que es similar al uso de códigos de condición para saltos (véase capítulo 2).

Mover la decisión del salto hacia arriba en el pipeline requiere que dos acciones se realicen antes: el cómputo de la dirección destino del salto, y la evaluación de la decisión de salto. La parte más sencilla de este cambio es adelantar el cálculo de la dirección de salto. En el registro IF/ID ya se dispone del valor del PC y del campo inmediato, por lo que basta sólo con mover el sumador de direcciones desde la etapa MEM a la etapa ID; por supuesto, aunque el cálculo de la dirección destino del salto se haga para todas las instrucciones, sólo se usará cuando realmente sea necesario.

La parte más complicada es la propia decisión de saltar. Para la comprobación de "saltar si igual" se deben comparar los dos registros leídos durante la etapa ID y ver si son iguales. La igualdad se puede verificar haciendo la operación or-exclusiva de los bits respectivos de cada uno de los valores leídos y haciendo la operación OR con estos resultados. Mover la verificación del salto a la etapa ID implica la necesidad de circuitería nueva para la detección de riesgos y para la anticipación de resultados, ya que una instrucción *branch* que depende de un resultado previo que está todavía en el pipeline debe seguir funcionado correctamente aunque se implemente la optimización. Por ejemplo, para

implementar la operación "saltar si igual" (y su inversa), se necesitará anticipar los resultados de una instrucción anterior a la lógica de comprobación que opera durante la etapa ID. Existen dos factores que complican las cosas:

1. Durante ID se debe descodificar la instrucción, decidir si se necesita la anticipación a la unidad de comprobación de igualdad, y completar esta comprobación para que si la instrucción es un salto entonces se pueda modificar el registro PC con la dirección destino del salto. La anticipación de los operandos de los saltos era anteriormente manejado por la lógica de anticipación de resultados de la ALU, pero la introducción de la unidad de comprobación de igualdad en la etapa ID requiere su propia lógica de anticipación de resultados. Observe que los operandos fuente que son anticipados a un salto pueden proceder tanto del registro de segmentación EX/MEM como del MEM/WB.
2. Puesto que los valores de una comparación de salto se necesitan en ID pero pueden ser producidos más tarde, es posible que ocurra un riesgo de datos y sea necesario un bloqueo. Por ejemplo, si una instrucción de ALU a la que le sigue inmediatamente una instrucción de salto produce uno de los operandos que necesita la comparación del salto, entonces se necesitará un bloqueo, ya que la etapa EX de la instrucción de tipo ALU se realiza después del ciclo ID que corresponde al salto. Por extensión, cuando a una carga le sigue inmediatamente un salto condicional que chequea el resultado de la carga, se produce un bloqueo durante dos ciclos, porque el resultado de la carga no está disponible hasta el final del ciclo MEM, pero se necesita al principio del ciclo ID del salto.

A pesar de estas dificultades, mover la ejecución de los saltos a la etapa ID es una mejora, ya que cuando el salto es tomado reduce la penalización de los saltos a solamente una instrucción, la que se está buscando en memoria en ese momento. Los ejercicios exploran los detalles de implementación de la ruta de anticipación de resultados y de la detección del riesgo de control.

Para eliminar instrucciones en la etapa de búsqueda (IF), se añade una línea de control llamada IF.Flush, la cual pone a cero el campo de instrucción del registro de segmentación IF/ID. Al poner este registro a cero se transforma la instrucción leída en una instrucción *nop*, una instrucción que no realiza ninguna acción ni cambia ningún estado.

Salto segmentado

EJEMPLO

Mostrar qué ocurre cuando en la siguiente secuencia de instrucciones el salto es tomado, suponiendo que el pipeline está optimizado para los saltos no tomados y que se ha movido la ejecución del salto a la etapa ID:

```
36 sub $10, $4, $8
40 beq  $1, $3, 7 # salto relativo a PC a 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw  $4,  50($7)
```

RESPUESTA

La figura 4.62 muestra lo que ocurre cuando el salto es tomado. Al contrario de la figura 4.61, al tomar el salto sólo aparece una burbuja en el pipeline.

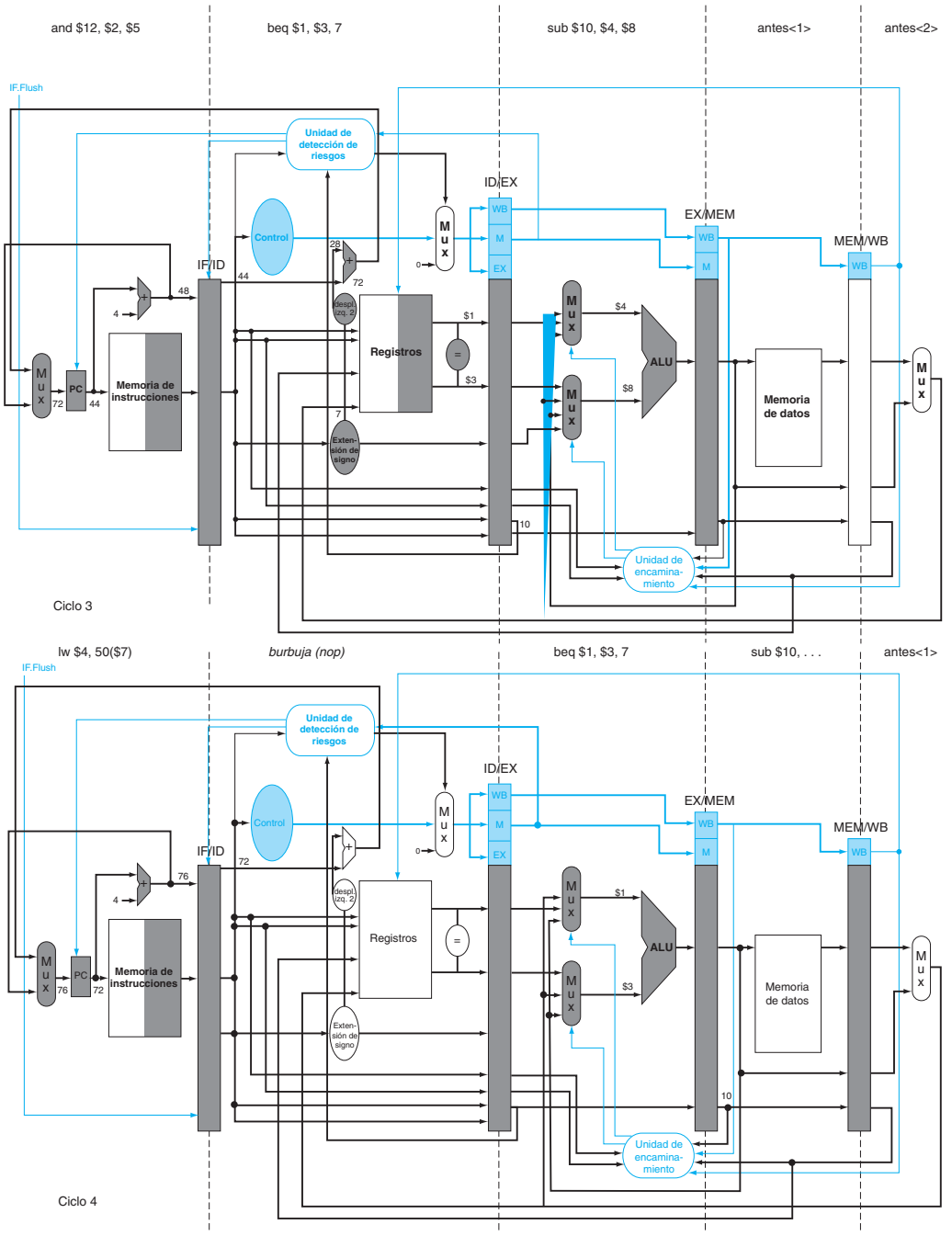


FIGURA 4.62 La etapa ID del ciclo 3 determina si el salto se debe tomar, de modo que selecciona el valor 72 como siguiente PC y pone a cero la instrucción que ha sido buscada para ser descodificada en el siguiente ciclo. El ciclo 4 muestra que se está buscando la instrucción situada en la posición 72 y que ha aparecido una burbuja o `nop` en el pipeline como resultado del salto tomado. (Ya que en realidad la instrucción `nop` corresponde con la instrucción `sll $0, $0, $0` es discutible si en el ciclo 4 se debería o no resaltar la etapa ID)

Predicción dinámica de saltos

Suponer que los saltos no son tomados es una forma simple de *predicción de saltos*. En este caso, la predicción es que los saltos no se toman, y se vacía el pipeline cuando la predicción es incorrecta. En el caso del pipeline simple de cinco etapas, esta estrategia, en muchas ocasiones realizada en colaboración con el compilador, es probablemente adecuada. En cambio, con pipelines más profundos (mayor número de etapas), la penalización de los fallos de la predicción de saltos, si se mide en ciclos de reloj, es mayor. De modo similar, la ejecución múltiple de instrucciones (véase la sección 4.10) incrementa la penalización en términos de oportunidades perdidas de ejecución de instrucciones. Esta combinación significa que en un diseño más agresivo del pipeline, un esquema simple de predicción estática no alcanzará unas buenas prestaciones. Tal como se mencionó en la sección 4.5, dedicando más hardware es posible tratar de predecir el comportamiento de los saltos durante la propia ejecución del programa.

Una estrategia es fijarse en la dirección de las instrucciones para ver si se trata de un salto que fue tomado la última vez que la instrucción se ejecutó, y en ese caso, comenzar a buscar la siguiente instrucción en el mismo sitio que se hizo la vez anterior. Esta técnica se denomina **predicción dinámica de saltos (*dynamic branch prediction*)**.

Predicción dinámica de saltos: predicción de saltos en tiempo de ejecución que usa la información que se va generando durante la ejecución.

Una manera de realizar esta estrategia es utilizar un **búfer de predicción de saltos (*branch prediction buffer*) o una tabla de historia de saltos (*branch history table*)**. Un búfer de predicción de saltos es una memoria pequeña indexada con la parte menos significativa de la dirección de la instrucción de salto. Esta memoria contiene un bit en cada entrada que dice si el salto ha sido recientemente tomado o no.

Búfer de predicción de saltos (tabla de historia de saltos): pequeña memoria que se indexa con la porción menos significativa de los bits de la dirección de la instrucción de salto y que contiene uno o más bits que indican si el salto ha saltado recientemente o no.

Esta es la clase más sencilla de búfer: en realidad no se sabe si la predicción es la correcta, ya que esa entrada podría contener la información de otro salto que coincida en los bits menos significativos de la dirección. Pero este problema no afecta a la corrección del mecanismo. La predicción es simplemente una apuesta que se supone que va a ser acertada, así que la búsqueda se inicia en la dirección predicha. Si la apuesta finalmente resulta ser falsa, entonces se eliminan las instrucciones predichas de forma incorrecta, se invierte el bit de predicción y se vuelve a guardar en la tabla, y a continuación se busca y ejecuta la secuencia correcta de instrucciones.

Este sencillo mecanismo de predicción basado en 1 bit tiene una desventaja en en cuanto a las prestaciones: aunque un salto sea tomado prácticamente siempre, es muy probable que cada vez que no sea tomado se hagan dos predicciones incorrectas en lugar de una. El siguiente ejemplo muestra este dilema.

EJEMPLO

Lazos y predicción

Considere un salto de final de lazo que salta nueve veces seguidas para iterar dentro del lazo hasta que la última vez deja de saltar para salir del lazo. ¿Cuál es la precisión de la predicción para este salto, suponiendo que el bit de predicción de este salto se mantiene en el búfer?

RESPUESTA

Durante el estado estable de la ejecución del programa, la predicción siempre fallará tanto en la primera como en la última iteración del lazo. Fallar en la última iteración es inevitable, ya que el bit de predicción siempre indicará que se debe tomar el salto, ya que en ese momento el salto ha sido tomado nueve veces seguidas. El fallo de predicción en la primera iteración del lazo se debe a que el bit se cambió justo al ejecutar el salto en la última iteración del lazo, ya que para salir del lazo el salto tuvo que ser no tomado. Por lo tanto, la precisión de la predicción de este salto, que es tomado el 90% de las veces, es sólo del 80% (dos predicciones incorrectas y ocho correctas).

Idealmente, para estos saltos tan regulares, la precisión del predictor tendría que ser igual a la frecuencia de saltos tomados. Para remediar el punto débil de este esquema con frecuencia se usan esquemas de predicción de 2 bits. Usando este esquema, la predicción ha de ser incorrecta 2 veces seguidas antes de cambiarla. La figura 4.63 muestra la máquina de estados finitos para un esquema de predicción de 2 bits.

Un búfer de predicción de saltos se puede implementar como un pequeño búfer especial al que se accede con la dirección de la instrucción durante la etapa IF. Si se predice la instrucción como tomada, la búsqueda de instrucciones empieza en la dirección destino del salto tan pronto como se conozca el PC; y esto, tal como se menciona en la página 377, puede ser como muy pronto en la etapa ID. En caso contrario, la búsqueda y ejecución secuencial de instrucciones continúa. Si la predicción resulta ser incorrecta, se cambian los bits de predicción tal como se muestra en la figura 4.63.

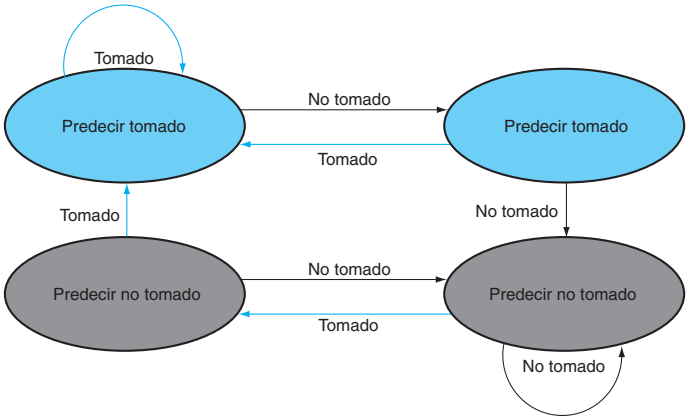


FIGURA 4.63 Estados en el esquema de predicción de 2 bits. Usando 2 bits en vez de 1, un salto que tiene una gran tendencia o bien a ser tomado o bien a ser no tomado (como hacen muchos saltos) será predicho incorrectamente sólo una vez. Los dos bits se utilizan para codificar los cuatro estados del sistema. El esquema de dos bits en un ejemplo general de los predictores basados en un contador, que se incrementa cuando la predicción es adecuada y se decrementa en el caso contrario, y el punto medio del rango de valores que puede tomar el contador se utiliza para tomar la decisión de predicción.

Extensión: Tal como se describió en la sección 4.5, en un pipeline de cinco etapas se puede evitar el problema de los riesgos de control redefiniendo el salto. Se define el salto retardado de forma que siempre se ejecuta la instrucción que hay después del salto, y es la segunda instrucción que sigue al salto la que será afectada por el salto.

Los compiladores y ensambladores intentan colocar después del salto, en el **hueco del salto retardado (*branch delay slot*)**, una instrucción que siempre se deba ejecutar. La tarea del software es conseguir que las instrucciones que suceden a los saltos retardados sean

Hueco de salto retardado: hueco que se encuentra directamente después de una instrucción de salto retardado, y que en la arquitectura MIPS es rellenado por una instrucción que no afecta al salto.

tanto válidas como útiles. La figura 4.64 muestra tres maneras de planificar el uso del hueco del salto retardado.

Las limitaciones de la planificación del uso de los saltos retardados provienen de (1) las restricciones en las instrucciones que se colocan en el hueco del salto retardado y de (2) la capacidad que se tiene en tiempo de compilación para predecir si un salto será tomado o no.

La utilización de saltos retardados fue una solución simple y efectiva para los procesadores segmentados en cinco etapas que ejecutaban una instrucción por ciclo. Desde que los procesadores se diseñan con pipelines más largos y con la capacidad para ejecutar más instrucciones por ciclo (véase la sección 4.10), el retardo de los saltos se hace mayor, y tener un único hueco de salto retardado resulta insuficiente. Además, el uso del salto retardado pierde popularidad en comparación con las estrategias dinámicas, que son más costosas pero más flexibles. Simultáneamente, el crecimiento en el número de transistores disponibles en el chip ha hecho que la predicción dinámica sea relativamente más barata.

Extensión: Un predictor de saltos nos indica si un salto debe o no ser tomado, pero todavía es necesario calcular la dirección destino del salto. En el pipeline de cinco etapas, este

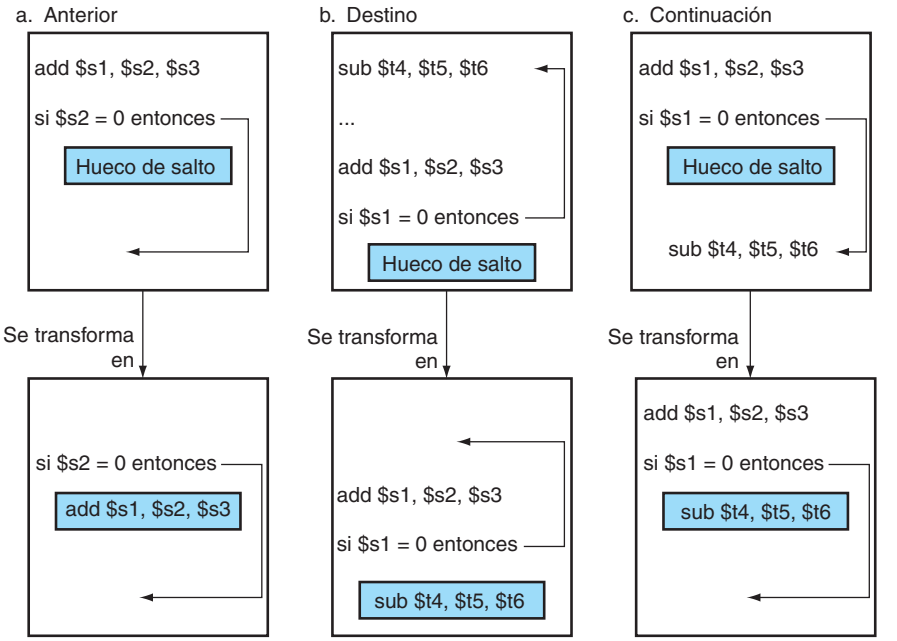


FIGURA 4.64 Planificación del hueco del salto retardado. El recuadro de la parte superior muestra el código antes de la planificación, mientras que el recuadro de la parte inferior muestra el código después de planificar la utilización del hueco. En el caso (a), en el hueco de salto retardado se coloca una instrucción independiente anterior al salto. Si es posible, ésta es la mejor elección, en caso contrario se emplean las estrategias (b) y (c). En las secuencias de código para (b) y (c), el uso del registro `$s1` en la condición del salto evita que la instrucción `add` (cuyo destino es `$s1`) se pueda mover al hueco del salto retardado. En (b) es la instrucción destino del salto la que se coloca en el hueco, y normalmente se necesitará mantener una copia de esta instrucción destino en su lugar de origen, ya que se puede llegar a ella también desde otros caminos. Esta estrategia es la preferida si el salto es tomado con mucha frecuencia, como es el caso del salto de un lazo. Finalmente, se puede colocar en el hueco una instrucción del camino que sigue al salto cuando éste no es tomado, como se muestra en (c). Para que las optimizaciones (b) y (c) funcionen correctamente, debe ser correcta la ejecución de la instrucción `sub` cuando el salto toma la dirección no esperada. Esto significa que se desperdicia trabajo, pero que el programa se ejecuta correctamente. Este sería el caso, por ejemplo, si `$t4` fuera un registro temporal que no tiene ningún uso cuando el salto va a la dirección no esperada.

cálculo necesita 1 ciclo, lo cual significa que los saltos tomados van a tener una penalización de un ciclo. Los saltos retardados representan un estrategia para eliminar esta penalización. Otra estrategia es usar una caché para almacenar la dirección destino del salto, o la instrucción destino del salto, usando un **búfer de destinos de salto (*branch target buffer*)**.

El esquema de predicción dinámica con 2 bits utiliza la información de cada salto en particular. Los investigadores se dieron cuenta de que el uso de la información de un salto en particular junto con la información global sobre el comportamiento de los saltos ejecutados recientemente logra una precisión mucho mayor utilizando la misma cantidad total de bits de predicción. Este tipo de predictores se denominan **predictores de correlación (*correlating predictors*)**. Un predictor de correlación típico puede dedicar dos predictores de 2 bits a cada salto permitiendo escoger entre los dos predictores en función de si el último salto ejecutado (no necesariamente el salto que se quiere predecir) ha sido tomado o no. De este modo, el comportamiento global de los saltos se puede interpretar como añadir más bits al índice que se utiliza para el acceso a la predicción dentro de una tabla.

Una innovación aún más reciente en la predicción de saltos es el uso de predictores de torneo. Un **predictor de torneo (*tournament predictor*)** utiliza múltiples predictores, y además toma nota, para cada salto particular, de cuál de los predictores básicos proporciona mejores resultados. Un predictor de torneo típico podría generar dos predicciones para cada índice de saltos: uno basado en información local y otro basado en el comportamiento global de los saltos. Para cada predicción un selector se encargaría de escojer entre uno de los dos predictores. El selector puede funcionar de forma similar a un predictor de 1 o 2 bits, favoreciendo al predictor que haya sido más preciso. Muchos de los microprocesadores avanzados más recientes hacen uso de estos predictores tan elaborados.

Búfer de destinos de salto: estructura que almacena las direcciones destino o las instrucciones destino de los saltos. Generalmente se organiza como una caché con etiquetas, lo cual la hace más costosa que un simple búfer de predicción.

Predictor de correlación: predictor de saltos que combina el comportamiento local de un salto en particular con la información global del comportamiento de un cierto número de saltos ejecutados recientemente.

Predictor de torneo: predictor de saltos que genera múltiples predicciones para un mismo salto y que para cada salto concreto utiliza un mecanismo de selección que escoge cuál de estas predicciones habilitar.

Extensión: Una forma de reducir el número de instrucciones de salto condicional es añadir instrucciones de *copia condicional* (*conditional move*). En lugar de cambiar el PC con un salto condicional, la instrucción cambia condicionalmente el registro destino de la copia. Si la condición falla, la copia se comporta como un `nop`. Por ejemplo, una versión de la arquitectura del repertorio de instrucciones MIPS tiene dos nuevas instrucciones llamadas `movn` (copia si no es cero) y `movz` (copia si es cero). Así, `movn $8, $11, $4` copia el contenido del registro 11 en el registro 8 si el valor del registro 4 es distinto de cero; en caso contrario no hace nada.

El repertorio de instrucciones ARM tiene un campo de condición en la mayoría de las instrucciones. De este modo, los programas de ARM podrían tener menos saltos condicionales que los programas MIPS.

Resumen de la segmentación

Este capítulo comenzó en una sala de lavado, donde se mostraban los principios de la segmentación en una tarea cotidiana. Usando esta analogía como guía, explicamos la segmentación de instrucciones paso a paso, comenzando con el camino de datos monociclo y después añadiendo registros de segmentación, caminos de anticipación de resultados, lógica de detección de riesgos, predicción de saltos, y la capacidad de vaciar el pipeline de instrucciones en el caso de excepciones. La figura 4.65 muestra la evolución final del camino de datos y de su control. Ahora estamos listos para otro riesgo de control: la cuestión peliaguda de las excepciones.

Autoevaluación

Considere los tres esquemas de predicción siguiente: predecir siempre no tomado, predecir siempre tomado, y predicción dinámica. Suponga que los tres tienen una penalización de cero ciclos cuando su predicción es correcta, y de 2 ciclos cuando es errónea. Suponga que la precisión promedio del predictor dinámico es del 90%. ¿Qué predictor supone la mejor elección en los siguientes casos?

1. Los saltos se toman con una frecuencia del 5%
2. Los saltos se toman con una frecuencia del 95%
3. Los saltos se toman con una frecuencia del 70%

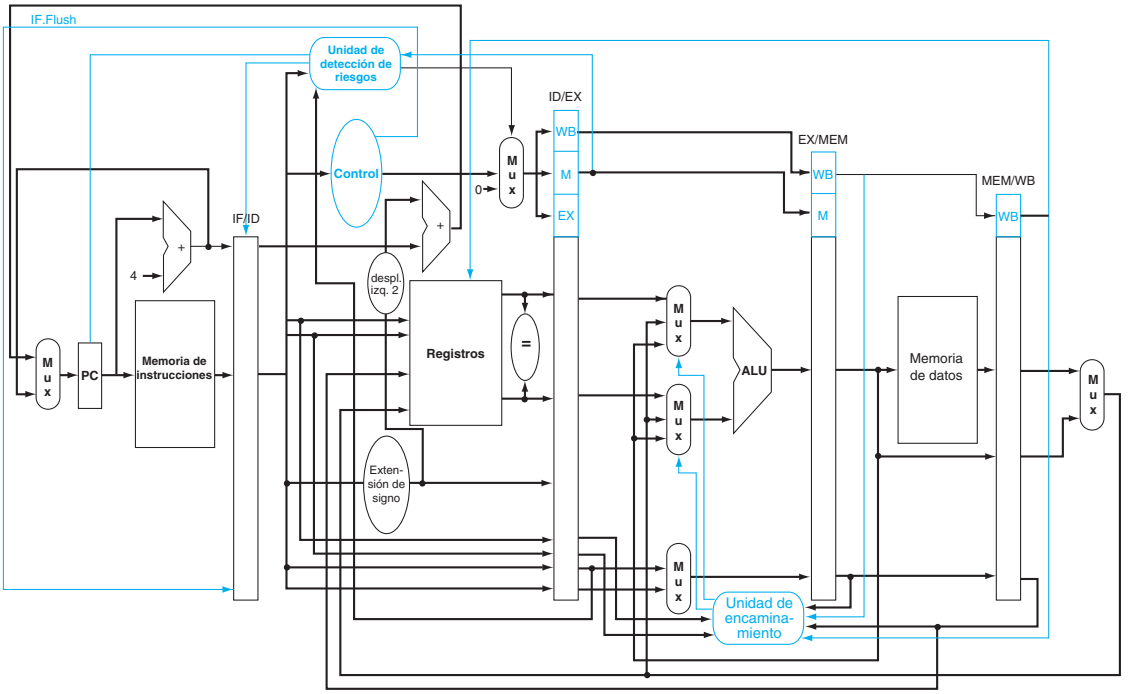


FIGURA 4.65 El resultado final del camino de datos y del control para este capítulo. Obsérvese que ésta es una figura simplificada en lugar de un camino de datos detallado, de modo que no está representado ni el multiplexor ALUsrc de la figura 4.57 ni las señales de control de los multiplexores de la figura 4.51.

Hacer que un computador dotado de un sistema automático de interrupción del programa funcione [de manera secuencial] no fue una tarea fácil, ya que cuando se produce una señal de interrupción el número de instrucciones que se encuentran en diferentes etapas de su procesamiento puede llegar a ser muy grande.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

4.9 Excepciones

El control es el aspecto más exigente del diseño del procesador: es la parte más difícil de construir, pues no sólo ha de funcionar bien, sino que también ha de hacerlo rápido. Una de las partes más problemáticas del control es la implementación de las **excepciones** y las **interrupciones** (sucesos, aparte de los saltos, que cambian el flujo normal de ejecución de las instrucciones). Se crearon inicialmente para procesar sucesos inesperados en el procesador, tales como el desbordamiento aritmético. Este mismo mecanismo básico fue extendido para la comunicación de los dispositivos de E/S con el procesador, como se verá en el capítulo 6.

Muchas arquitecturas y autores no distinguen entre interrupciones y excepciones, y a menudo denominan interrupciones a los dos tipos de sucesos. Por ejemplo, el x86 de Intel usa interrupción para todos los sucesos. Siguiendo con la convención del MIPS, se utiliza el término excepción para referirse a cualquier

cambio inesperado en el flujo de control sin distinguir si la causa es interna o externa; y se usa el término interrupción sólo cuando el suceso es externo. Aquí hay algunos ejemplos que muestran si la situación es generada internamente por el procesador o externamente:

Excepción (interrupción): suceso no planificado que interrumpe la ejecución del programa; se utiliza para detectar el desbordamiento

Interrupción: excepción que proviene del exterior del procesador. (Algunas arquitecturas utilizan el término *interrupción* para todas las excepciones).

Tipo de suceso	¿De dónde?	Terminología MIPS
Demanda de dispositivos de entrada y salida	Externo	Interrupción
Invocar al sistema operativo desde el programa del usuario	Interno	Excepción
Desbordamiento aritmético	Interno	Excepción
Utilizar una instrucción no definida	Interno	Excepción
Errores hardware	Cualquiera	Excepción o interrupción

Una gran parte de la necesidad de soportar excepciones proviene de las situaciones específicas que las causan. Por lo tanto, se volverá a este tema en el capítulo 5, cuando se aborden las jerarquías de memoria, y en el capítulo 6, cuando se vean los dispositivos de entrada y salida, y se entenderá mejor la motivación de las capacidades adicionales en el mecanismo de excepciones. En esta sección se estudiará la implementación del control para detectar dos tipos de excepciones que ya se han tratado.

La detección de situaciones excepcionales y la respuesta mediante la acción apropiada está a menudo en el camino crítico del tiempo del procesador, que determina el tiempo de ciclo de reloj y sus prestaciones. Si no se ha prestado atención a las excepciones durante el diseño de la unidad de control, el intento de añadirlas a posteriori a una implementación complicada puede reducir las prestaciones significativamente, así como complicar la tarea de conseguir un diseño correcto.

Cómo se tratan las excepciones en la arquitectura MIPS

Los dos tipos de excepciones que la implementación actual puede generar son la ejecución de una instrucción no definida y un desbordamiento aritmético. En las siguientes páginas, usaremos el desbordamiento aritmético en la instrucción `add $1, $2, $1`, como ejemplo de excepción. La acción básica que la máquina debe realizar cuando ocurre una excepción es guardar la dirección de la instrucción responsable (*offending*) en el contador de programa de excepción (EPC) y entonces transferir el control al sistema operativo en alguna dirección específica.

Entonces, el sistema operativo puede efectuar la acción apropiada, que puede incluir el darle algún servicio al programa de usuario, tomar alguna acción predefinida en respuesta a un desbordamiento, parar la ejecución del programa o mostrar un mensaje de error. Después de realizar la acción necesaria causada por la excepción, el sistema operativo puede finalizar el programa o continuar su ejecución, utilizando el EPC para saber dónde se ha de retomar la ejecución. En el capítulo 5 se analiza detalladamente el tema del restablecimiento de la ejecución.

Para que el sistema operativo trate la excepción, debe conocer las razones por las que se ha producido, además de la instrucción que la ha causado. Hay dos métodos principales para comunicar la causa de una excepción. El método utilizado en la arquitectura MIPS es incluir un registro de estado denominado registro de causa (*cause register*), que contiene un campo que indica la razón de la excepción.

Interrupción vectorizada: interrupción para la cual la dirección a la que se transfiere el control viene determinada por la causa de la excepción.

Un segundo método es utilizar **interrupciones vectorizadas**. En una interrupción vectorizada, la dirección a la que se transfiere el control viene determinada por la causa de la excepción. Por ejemplo, para conformar los dos tipos de excepción antes mencionados se podrían definir las siguientes direcciones en los vectores de excepción:

Tipo de excepción	Dirección del vector de excepciones (en hexadecimal)
Instrucción no definida	$8000\ 0000_{hex}$
Desbordamiento aritmético	$8000\ 0180_{hex}$

El sistema operativo conoce la razón de la excepción por la dirección donde se inicia. Las direcciones están separadas por 32 bytes u 8 instrucciones, y el sistema operativo debe guardar la causa de la excepción y ejecutar algún proceso limitado a esta secuencia. Cuando la excepción no está vectorizada, se puede utilizar un único punto de entrada para todas las excepciones, y el sistema operativo descodificaría el registro de estado para encontrar la causa.

Se puede realizar el proceso requerido por las excepciones añadiendo algunos registros extra y algunas señales de control a nuestra implementación básica y ampliando ligeramente el control. Suponga que se está implementando el sistema de excepciones utilizado en la arquitectura MIPS (la implementación de excepciones vectorizadas no es más difícil). Se tendrán que añadir dos registros adicionales al camino de datos:

- *EPC*: un registro de 32 bits utilizado para guardar la dirección de la instrucción afectada (este registro se necesita aún cuando las excepciones están vectorizadas).
- *Causa*: un registro para almacenar la causa de la excepción. En la arquitectura MIPS, este registro es de 32 bits, aunque algunos de estos bits no se utilizan normalmente. Se supone que existe un campo de cinco bits para codificar las dos posibles fuentes de excepciones mencionadas anteriormente, con 10 representando una instrucción no definida y 12 representando un desbordamiento aritmético.

Excepciones en una implementación segmentada

Una implementación segmentada trata las excepciones como cualquier otro riesgo de control. Por ejemplo, supóngase que hay un desbordamiento aritmético en una instrucción de suma. Tal como se ha hecho en las secciones previas para un salto tomado, se descartan las instrucciones que han entrado en el pipeline a continuación de la suma y se comienza la búsqueda de instrucciones a partir de la nueva dirección. Para la excepciones usaremos el mismo esquema que se ha utilizado para los saltos tomados, pero en esta ocasión la excepción hace que se desactiven las señales de control.

Al gestionar los fallos de predicción de saltos ya vimos cómo eliminar la instrucción en la etapa IF transformándola en una `nop`. Para eliminar instrucciones en la etapa ID se usa el multiplexor que ya existe en la etapa ID y que se usaba para poner a cero las señales de control en caso de bloqueos. Una nueva señal de control, llamada ID.Flush, y la señal de bloqueo producida por la unidad de detección de riesgos se combinan con una O-lógica para eliminar la instrucción

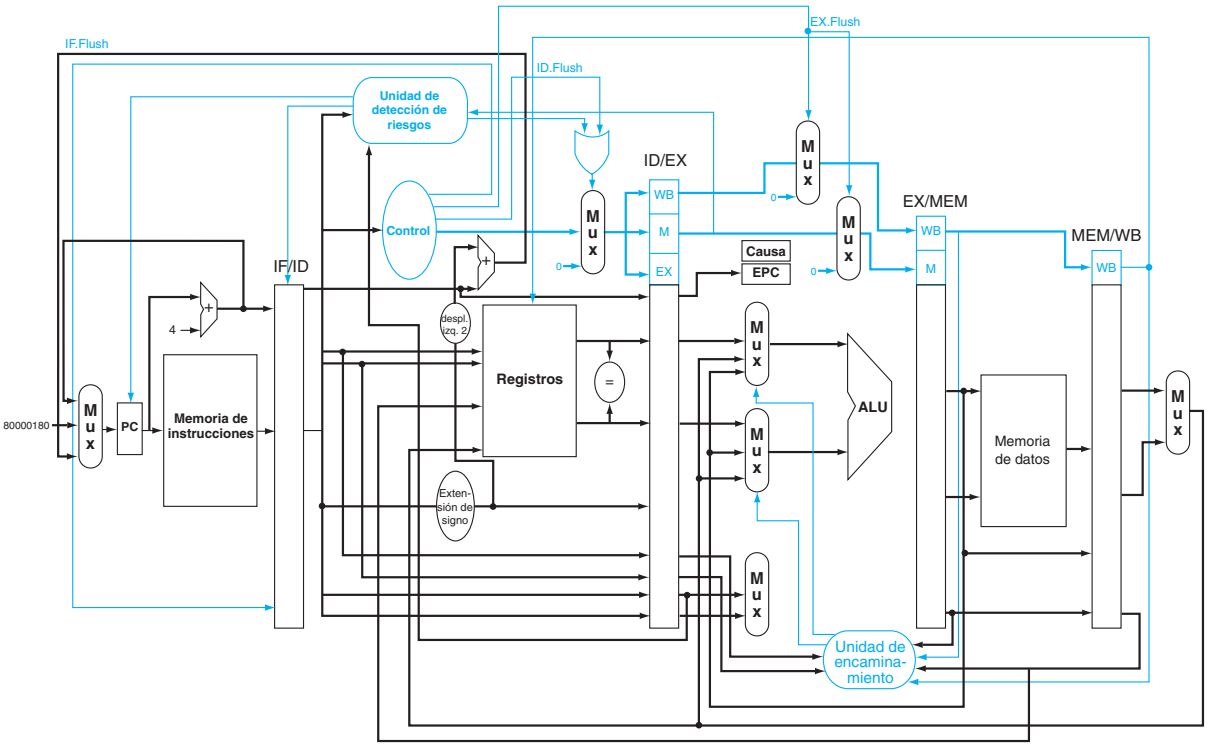


FIGURA 4.66 Camino de datos con el control para gestionar excepciones. Los cambios clave en el diseño incluyen una nueva entrada con el valor 8000 0180_{hex} en el multiplexor que proporciona el nuevo valor del PC; un registro de Causa para registrar el motivo o causa de la excepción, y un registro PC de excepciones (*Exception PC*) para guardar la dirección de la instrucción que ha provocado la excepción. La entrada 8000 00180_{hex} al multiplexor es la dirección inicial desde donde se debe empezar la búsqueda de instrucciones cuando ocurre una excepción. A pesar de que no se muestra, la señal de desbordamiento de la ALU es una entrada a la unidad de control.

en la etapa ID. Para eliminar la instrucción situada en la etapa EX se usará una nueva señal, llamada EX.Flush, que hará que nuevos multiplexores pongan a cero las líneas de control. Para empezar la búsqueda de instrucciones en la posición 8000 0180_{hex}, que es la localización de la excepción en el caso de un desbordamiento aritmético, basta con añadir una entrada adicional al multiplexor del PC que envíe el valor 8000 0180_{hex} al PC. La figura 4.66 muestra estos cambios.

Este ejemplo nos muestra un problema que ocurre con las excepciones: si no se para la ejecución de la instrucción a la mitad, el programador no podrá ver el valor del registro `$1` que ha ayudado a causar la excepción, ya que este valor se perderá al ser `$1` el registro destino de la propia instrucción `add`. Mediante una cuidadosa planificación se consigue detectar el desbordamiento en la etapa EX, de modo que se podrá usar la señal EX.Flush para evitar que la instrucción que se encuentra en la etapa EX escriba su resultado en la etapa de WB. Muchas excepciones requieren que se acabe completando la ejecución de la instrucción que causó la excepción como si se hubieran ejecutado de forma normal. La forma más sencilla de llevar esto a cabo es primero anular la ejecución de la instrucción vaciando del pipeline, y después de que la excepción haya sido ya gestionada reiniciar su ejecución desde el principio.

El paso final consiste en guardar el PC de la instrucción que ha causado la excepción en el registro PC de excepciones (EPC). En realidad, se guarda la dirección de la instrucción + 4, por lo que la rutina de gestión de la excepción deberá primero restar el valor 4 al valor guardado. La figura 4.66 muestra una versión simplificada del camino de datos, que incluye la circuitería para saltos y los cambios que son necesarios para gestionar excepciones.

EJEMPLO

Excepciones en un computador segmentado

Dada la siguiente secuencia de instrucciones,

```
40hex     sub   $11, $2, $4
44hex     and   $12, $2, $5
48hex     or    $13, $2, $6
4Chex     add    $1, $2, $1
50hex     slt   $15, $6, $7
54hex     lw    $16, 50($7)
...
```

suponga que las instrucciones que se han de invocar cuando se produce una excepción empiezan así:

```
80000180hex     sw      $24, 1000($0)
80000184hex     sw      $26, 1004($0)
...
```

Mostrar qué ocurre en el pipeline si se produce una excepción de desbordamiento en la instrucción `add`.

RESPUESTA

La figura 4.67 muestra los sucesos que se producen, empezando con la instrucción `add` situada en la etapa EX. Durante esta fase se detecta el desbordamiento, y se fuerza al PC a que tome el valor 8000 0180_{hex}. El ciclo 7 muestra cómo se eliminan tanto la instrucción `add` como las siguientes, y se busca la primera instrucción del código de excepción. Observe que se guarda la dirección de la instrucción que *sigue* a `add`: $4C_{hex} + 4 = 50_{hex}$.

En la página 385 se han mencionado cinco ejemplos de excepciones y veremos algunos más en los capítulo 5 y 6. Con cinco instrucciones activas en cada ciclo de reloj, el reto consiste en asociar la excepción con la instrucción apropiada. Además, pueden aparecer múltiples excepciones de forma simultánea en un mismo ciclo. La solución más corriente es dar prioridad a las excepciones de tal manera que sea fácil determinar cuál de ellas se ha de servir primero. En la mayoría de las implementaciones de procesadores MIPS, el hardware ordena las excepciones de modo que sea la instrucción más antigua la que se vea interrumpida.

Las peticiones de un dispositivo de E/S y el mal funcionamiento del hardware no están asociados a una instrucción específica, por lo que hay cierta flexibilidad a la hora de tratarlas, como al decidir el momento en el que se debe interrumpir el pipeline. Así, usar el mismo mecanismo que se usa para las otras excepciones funciona perfectamente.

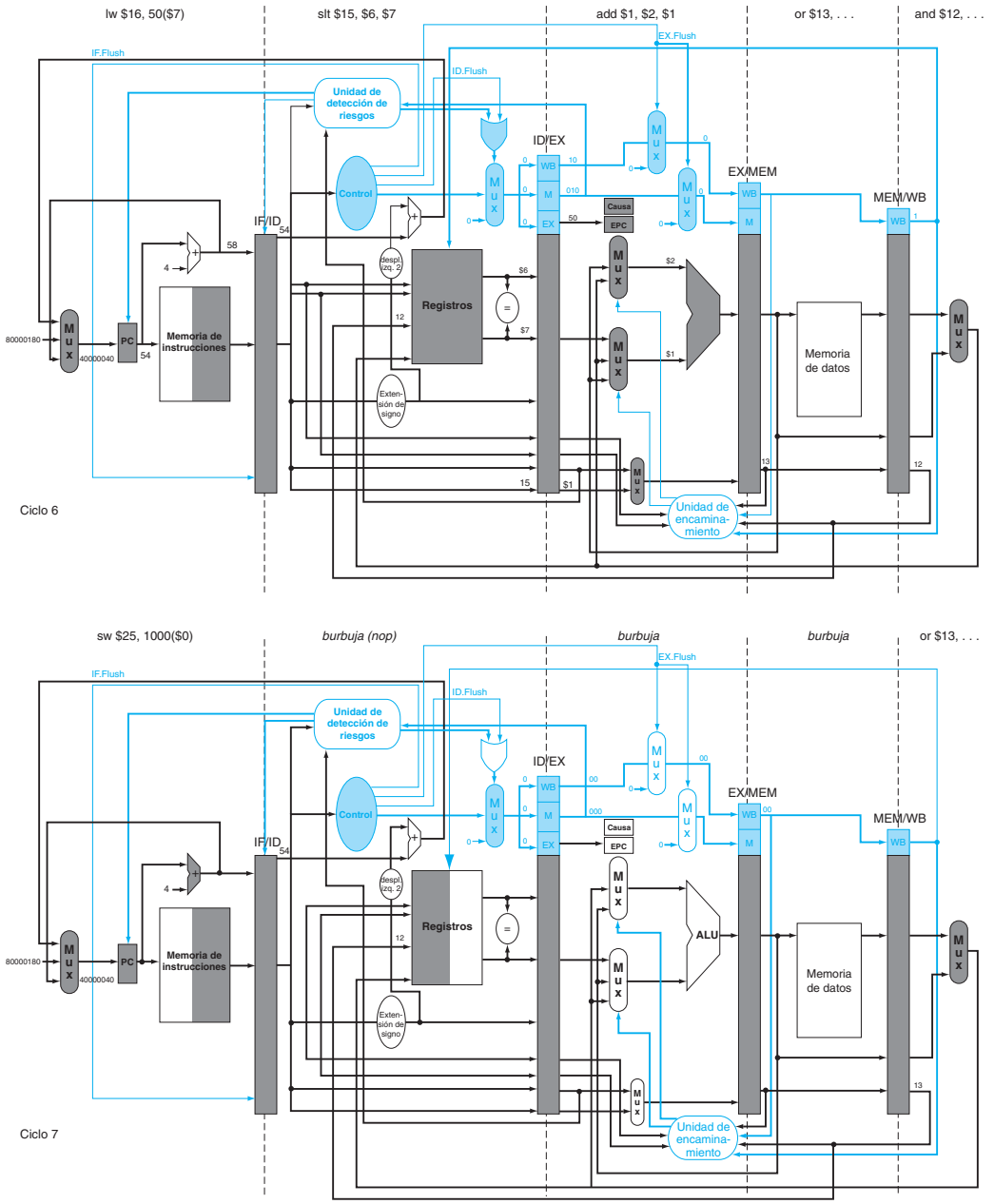


FIGURA 4.67 Resultado de una excepción causada por un desbordamiento aritmético en la instrucción de suma. El desbordamiento se detecta durante la etapa EX, en el ciclo 6, guardando la dirección de la instrucción que sigue a `add` en el registro EPC ($4C + 4 = 50_{hex}$). El desbordamiento hace que cerca del final de este ciclo de reloj se activen todas las señales de vaciado, desactivando los valores de las líneas control (poniéndolos a 0) para la instrucción `add`. El ciclo 7 muestra las instrucciones convertidas en burbujas dentro del pipeline y la búsqueda en la posición $8000\ 0180_{hex}$ de la primera instrucción de la rutina de excepción `sw $25 1000($0)`. Debe notarse que las instrucciones `and` y `or`, que estaban antes del `add`, se acaban completando. A pesar de que no se muestra, la señal de desbordamiento de la ALU es una entrada en la unidad de control.

El registro EPC captura la dirección de la instrucción interrumpida, y el registro de Causa de MIPS registra todas las posibles excepciones que se pueden dar en un ciclo de reloj, por lo que la rutina de gestión de la excepción debe asociar el tipo de la excepción producida a la instrucción que la provoca. Una pista importante para hacerlo es saber qué tipo de excepción puede ocurrir en cada una de las etapas de segmentación. Por ejemplo, el caso de una excepción provocada por una instrucción no definida se descubre en la etapa ID, mientras que una llamada al sistema operativo ocurre en la etapa de EX. Las excepciones se recogen en el registro de Causa de modo que una vez resuelta la gestión de la excepción de una instrucción anterior, el hardware se pueda interrumpir en una nueva excepción causada por una instrucción posterior.

Interfaz hardware software

La máquina y el sistema operativo deben trabajar conjuntamente para que las excepciones se comporten tal como se espera. Normalmente, el contrato por parte del hardware consiste en detener la instrucción que causa la excepción en medio del flujo de instrucciones, dejar que las instrucciones anteriores se completen, eliminar todas las instrucciones posteriores, modificar un registro para que muestre la causa de la excepción, guardar la dirección de la instrucción que la ha provocado y entonces saltar a una dirección predefinida. El contrato por parte del sistema operativo consiste en considerar la causa de la excepción y actuar en consecuencia. En el caso de una instrucción no definida, del mal funcionamiento del hardware o de una excepción producida por desbordamiento, el sistema operativo normalmente detiene la ejecución del programa y devuelve un indicador de la razón de esta detención. En el caso de una petición por parte de un dispositivo de E/S o de una llamada a una rutina de servicio del sistema operativo, éste guarda el estado del programa, realiza la tarea demandada y en algún punto del futuro restaura el programa para reanudar su ejecución. En el caso de pedir el uso de un dispositivo de E/S, frecuentemente se decide ejecutar otra tarea diferente antes de reanudar la tarea que realizó la petición de E/S, ya que generalmente la tarea no será capaz de continuar hasta que la tarea de E/S haya sido completada. Por esta razón es crítica la habilidad de guardar y restaurar el estado de una tarea. Uno de los usos más importantes y frecuentes de las excepciones es la gestión de fallos de página y de fallos de TLB. El capítulo 5 describe estas excepciones y su gestión de forma más detallada.

Interrupción (excepción) imprecisa: interrupción o excepción en un computador segmentado y que no está asociada a la instrucción exacta que ha causado la interrupción o excepción.

Interrupción (excepción) precisa: interrupción o excepción que en un computador segmentado siempre se asocia con la instrucción correcta.

Extensión: La dificultad en los computadores segmentados para asociar siempre de forma correcta la excepción con la instrucción que la ha causado ha llevado a algunos diseñadores de computadores a relajar los requerimientos en los casos que no son críticos. De estas máquinas se dice que tienen **interrupciones imprecisas o excepciones imprecisas**. En el caso del ejemplo anterior, el PC normalmente valdría 58_{hex} al comienzo del ciclo siguiente a la detección de la excepción, aún cuando la instrucción que causa la excepción se encuentra en la dirección $4C_{hex}$. Una máquina con excepciones imprecisas podría poner el valor 58_{hex} en EPC y dejar que el sistema operativo detecte cuál fue la instrucción que causó el problema. MIPS y la mayoría de los computadores actuales soportan **interrupciones precisas o excepciones precisas**. (Una razón es que han de poder soportar memoria virtual, lo cual se verá en el capítulo 5.)

Extensión: Aunque en MIPS la dirección a la que transfiere el control en casi todas las excepciones es la 8000 0180hex, utiliza también la dirección 8000 0000hex para mejorar las prestaciones del procesamiento de las excepciones de fallo de TLB (véase el capítulo 5).

Autoevaluación

¿Qué excepción debería reconocerse en primer lugar en esta secuencia?

```
1. add  $1, $2, $1    # desbordamiento aritmético
2. XXX  $1, $2, $1    # instrucción no definida
3. sub  $1, $2, $1    # error del hardware
```

4.10 Paralelismo y paralelismo a nivel de instrucciones avanzado

Se advierte con antelación que esta sección proporciona una visión general breve de temas fascinantes pero avanzados. Si se desea aprender más detalles, se debería consultar el libro más avanzado, *Arquitectura de Computadores: Un enfoque cuantitativo,* ¡donde el material cubierto por las siguientes 13 páginas se amplía a más de 200 páginas (incluyendo apéndices)!

La segmentación aprovecha el paralelismo potencial entre las instrucciones. Este paralelismo se denomina **paralelismo a nivel de instrucciones (*instruction-level Parallelism*, ILP)**. Existen dos estrategias básicas para incrementar la cantidad potencial del ILP. La primera consiste en aumentar la profundidad del pipeline para solapar la ejecución de más instrucciones. Empleando la analogía de la lavandería y suponiendo que el ciclo de lavado fuera más largo que los otros, se podría dividir nuestra lavadora en tres máquinas que realizaran los pasos de lavado, aclarado y centrifugado de la máquina tradicional. De este modo, convertiríamos el pipeline de cuatro etapas en un pipeline de seis etapas. Para conseguir la máxima ganancia en velocidad, tanto en el caso del procesador como en el caso de la lavandería, los pasos restantes se deberían volver a equilibrar para que tuvieran la misma longitud. La cantidad de paralelismo que se aprovecharía es mayor, puesto que se solapa la ejecución de más instrucciones. Las prestaciones también serían potencialmente mayores puesto que se puede reducir el ciclo de reloj.

Paralelismo a nivel de instrucciones: paralelismo entre instrucciones.

La segunda estrategia consiste en replicar los componentes internos del computador para poder ejecutar múltiples instrucciones dentro de cada etapa de segmentación. El nombre general para esta técnica es **ejecución múltiple (*multiple issue*)**. Una lavandería con ejecución múltiple reemplazaría muestra lavadora y secadora, por ejemplo, por tres lavadoras y tres secadoras. También se tendrían que reclutar más asistentes para doblar y guardar en el mismo tiempo una cantidad de ropa tres veces mayor. El inconveniente de la estrategia es que se requiere trabajo adicional para conseguir mantener todas las máquinas ocupadas y para transferir las cargas de ropa de una etapa de segmentación a la siguiente.

Ejecución múltiple: esquema que permite lanzar varias instrucciones para ejecutarse durante un ciclo de reloj.

Ejecutar varias instrucciones por etapa permite que la frecuencia de instrucciones ejecutadas sea mayor que la frecuencia del reloj, o, dicho de otra manera, que el CPI sea menor que 1. En ocasiones es beneficioso invertir la métrica del CPI y usar el IPC (*instructions per clock cycle*, instrucciones ejecutadas por ciclo de reloj). Por ejemplo, un microprocesador con ejecución múltiple de hasta cuatro instrucciones que funcione a 4 GHz podrá llegar a ejecutar instrucciones a una velocidad pico de 16 mil millones de instrucciones por segundo, y tener en el mejor de los casos un CPI de 0,25, o un IPC de 4. Suponiendo un pipeline de cinco etapas, este procesador tendría en todo momento hasta 20 instrucciones válidas en ejecución. Los microprocesadores de gama alta de hoy en día tratan de ejecutar entre tres y seis nuevas instrucciones (4 vías) en cada ciclo de

reloj. Sin embargo, es habitual encontrar muchas restricciones en el tipo de instrucciones que se pueden ejecutar de forma simultánea y diferencias en la forma de tratar las dependencias que se van encontrando entre las instrucciones.

Hay dos formas fundamentales de implementar un procesador de ejecución múltiple. La diferencia principal entre ellas es la forma de repartir el trabajo entre el compilador y el hardware. Puesto que esta división del trabajo determina si las decisiones se hacen estáticamente (esto es, en tiempo de compilación) o dinámicamente (durante la ejecución), a estas estrategias frecuentemente se las denomina **ejecución múltiple con planificación estática (*static multiple issue*)** y **ejecución múltiple con planificación dinámica (*dynamic multiple issue*)**. Veremos que ambas estrategias tienen otros nombres más conocidos, pero que pueden ser menos precisos o más restrictivos.

Ejecución múltiple con planificación estática: estrategia de implementación de un procesador con ejecución múltiple en la que el compilador toma muchas decisiones antes de la ejecución del programa.

Ejecución múltiple con planificación dinámica: estrategia de implementación de un procesador con ejecución múltiple en la que el propio procesador toma muchas decisiones durante la ejecución del programa.

Un pipeline de ejecución múltiple debe responsabilizarse de dos tareas principales muy distintas:

1. Empaquetar instrucciones en **ranuras de ejecución (*issue slots*)**: ¿Cómo determina el procesador cuántas y qué instrucciones pueden ser lanzadas a ejecutar en un determinado ciclo de reloj? En muchos procesadores que planifican el lanzamiento de instrucciones de forma estática, este proceso es gestionado, al menos parcialmente, por el compilador; en procesadores que planifican el lanzamiento de instrucciones de forma dinámica, esta tarea se realiza en tiempo de ejecución por parte del procesador, aunque es frecuente que el compilador haya tratado previamente de colocar las instrucciones en un orden beneficioso, que ayude a mejorar la velocidad de ejecución.
2. Gestionar los riesgos de datos y de control: En procesadores con ejecución de instrucciones planificado estáticamente algunas o todas las consecuencias de los riesgos de datos y de control son gestionadas por el compilador. Por el contrario, muchos procesadores con lanzamiento de instrucciones planificado dinámicamente intentan aliviar al menos algunas clases de riesgos usando técnicas hardware aplicadas en tiempo de ejecución.

Ranuras de ejecución: posiciones desde las que las instrucciones pueden ser enviadas a ejecutar en cada ciclo de reloj; usando una analogía, podrían corresponder a las posiciones de salida para una carrera.

Aunque se describan las dos estrategias como si fueran completamente distintas, en realidad algunas técnicas usadas por una de las estrategias son tomadas prestadas por la otra, y ninguna de las dos puede proclamarse como perfectamente pura.

El concepto de la especulación

Uno de los métodos más importantes para encontrar y aprovechar más el ILP es la **especulación** que es una estrategia que permite al compilador o al procesador "adivinar" las propiedades de una instrucción, y de este modo habilitar la ejecución inmediata de otras instrucciones que puedan depender de la instrucción sobre la cual se ha especulado. Por ejemplo, es posible especular con el resultado de un salto, de forma que las instrucciones que siguen al salto puedan ser ejecutadas antes. O se puede especular que una instrucción de almacenamiento que precede a una instrucción de carga no hace referencia a la misma dirección, lo cual permitiría que la carga se ejecute antes que el almacenamiento. La dificultad de la especulación es que puede ser errónea. Así, cualquier mecanismo de especulación debe incluir tanto un método para verificar si la predicción fue correcta, como un método para deshacer los efectos de las instrucciones que fueron ejecutadas especulativamente. La implementación de esta capacidad de vuelta atrás añade complejidad a los procesadores que soportan la especulación.

Especulación: estrategia utilizada por el compilador o por el procesador para predecir el resultado de una instrucción, y de ese modo poder eliminar la dependencia que esa instrucción genera en la ejecución de otras instrucciones.

La especulación la puede realizar tanto el compilador como el hardware. Por ejemplo, el compilador puede usar la especulación para reordenar las instrucciones, moviendo una instrucción antes o después de un salto, o una instrucción de carga antes o después de un almacenamiento. El procesador puede realizar la misma transformación por hardware y durante la ejecución usando técnicas que veremos un poco más adelante en esta sección.

Los mecanismos usados para recuperarse de una especulación incorrecta son bastante diferentes. En el caso de la especulación por software, el compilador generalmente inserta instrucciones adicionales para verificar la precisión de la especulación y proporciona rutinas que arreglan los desperfectos para que sean utilizadas en el caso de que la especulación sea incorrecta. En la especulación por hardware el procesador normalmente guarda los resultados especulativos hasta que se asegura de que ya no son especulativos. Si la especulación ha sido correcta, las instrucciones se completan permitiendo que el contenido de los búferes en los que se guardan temporalmente los resultados se escriban en los registros o en memoria. Si la especulación ha sido incorrecta, el hardware vacía el contenido de los búferes y se vuelve a ejecutar la secuencia de instrucciones correcta.

La especulación introduce otro posible problema: especular con ciertas instrucciones puede introducir excepciones que previamente no se producían. Por ejemplo, suponga que una instrucción de carga se mueve para ser ejecutada de forma especulativa, pero que la dirección que utiliza no es válida cuando la especulación es incorrecta. El resultado sería la generación de una excepción que nunca debería haber ocurrido. El problema se complica por el hecho de que si la instrucción de carga se está ejecutando de forma especulativa y la especulación es correcta, ¡entonces la excepción sí que debe ocurrir! En la especulación basada en el compilador estos problemas se evitan añadiendo un soporte especial a la especulación que permite que las excepciones sean ignoradas hasta que se pueda clarificar si éstas realmente deben ocurrir o no. En la especulación basada en hardware las excepciones son simplemente retenidas temporalmente en un búfer hasta que se verifica que la instrucción que causa la excepción ya no es especulativa y está lista para ser completada; llegados a este punto es cuando se permite que la excepción sea visible y se procede a su manejo siguiendo el mecanismo normal.

Puesto que la especulación puede mejorar las prestaciones cuando se hace de forma adecuada y puede reducirlas si se hace sin cuidado, se debe dedicar un esfuerzo importante a decidir cuándo es apropiado especular y cuándo no. Más adelante, en esta sección, veremos técnicas especulativas tanto estáticas como dinámicas.

Ejecución múltiple con planificación estática

Todos los procesadores con ejecución múltiple y planificación estática utilizan el compilador para que les asista en la tarea de empaquetar las instrucciones y en la gestión de los riesgos. En estos procesadores se puede interpretar que el conjunto de instrucciones que comienzan su ejecución en un determinado ciclo de reloj, lo que se denomina **paquete de ejecución (*issue packet*)**, es una instrucción larga con múltiples operaciones. Esta visión es mucho más que una analogía. Como este tipo de procesadores generalmente restringen las posibles combinaciones de instrucciones que pueden ser iniciadas en un determinado ciclo, es muy conveniente pensar en el paquete de ejecución como en una instrucción única que contiene ciertos campos predefinidos que le permiten especificar varias operaciones. Esta visión fue la que dio pie al nombre original de esta estrategia: **Very Long Instruction Word (VLIW, literalmente: palabra de instrucción de tamaño muy grande)**.

Paquete de ejecución: conjunto de instrucciones que se lanzan a ejecutar juntas durante un mismo ciclo de reloj; el paquete de instrucciones puede ser determinado de forma estática por parte del compilador o puede ser determinado de forma dinámica por parte del procesador.

Very Long Instruction Word, VLIW (palabra de instrucción de tamaño muy grande): estilo de arquitectura de repertorio de instrucciones que junta muchas operaciones independientes en una única instrucción más ancha, típicamente con muchos campos de código de operación separados.

La mayoría de los procesadores de planificación estática de la ejecución también confían al compilador cierta responsabilidad en la gestión de los riesgos de datos y de control. Entre estas responsabilidades se suele incluir la predicción estática de saltos y la reordenación de instrucciones para reducir o incluso evitar todos los riesgos de datos. Antes de describir el uso de las técnicas de planificación estática de ejecución con procesadores más agresivos, echaremos un vistazo a una versión simple del procesador MIPS.

Ejemplo: ejecución múltiple con planificación estática con MIPS

Para dar una idea de la planificación estática consideraremos un procesador MIPS capaz de ejecutar dos instrucciones por ciclo (dos vías, two issue), en el cual una de las instrucciones puede ser una operación entera en la ALU o un salto condicional, mientras que la otra puede ser una instrucción de memoria (carga o almacenamiento). Este tipo de diseño es similar al que se usa en algunos procesadores MIPS empotrados. Ejecutar dos instrucciones por ciclo requerirá la búsqueda y descodificación de un total de 64 bits de instrucciones. En muchos procesadores de planificación estática, y esencialmente en todos los procesadores VLIW, las posibilidades para seleccionar las instrucciones que se ejecutan simultáneamente están restringidas, para así simplificar la tarea de descodificar las instrucciones y de lanzarlas a ejecución. Así, suele ser necesario que la pareja de instrucciones esté alineada en una frontera de 64 bits, y que la operación ALU o de salto aparezca siempre en primer lugar. Si una de las instrucciones de la pareja no se puede utilizar, entonces será necesario reemplazarla por una no-operación (`nop`). Por tanto, las instrucciones se lanzarán a ejecutar siempre en parejas, aunque en ocasiones una de las ranuras de ejecución contenga un `nop`. La figura 4.68 muestra cómo avanzan las instrucciones a pares dentro del pipeline.

Los procesadores con planificación estática de la ejecución varían en la forma en que gestionan los riesgos potenciales de datos y de control. En algunos diseños es el compilador quien asume la total responsabilidad de eliminar *todos* los riesgos, planificando el código e insertando instrucciones `nop` para que se ejecute sin necesidad de la lógica de detección de riesgos y sin necesidad de que el hardware deba bloquear la ejecución. En otros diseños, el hardware detecta los riesgos de datos y genera bloqueos entre dos paquetes de ejecución consecutivos, aunque requiere que sea el compilador quien evite

Tipo de instrucción	Etapas del pipeline							
Instrucción ALU o salto condicional	IF	ID	EX	MEM	WB			
Instrucción carga o almacenamiento	IF	ID	EX	MEM	WB			
Instrucción ALU o salto condicional		IF	ID	EX	MEM	WB		
Instrucción carga o almacenamiento		IF	ID	EX	MEM	WB		
Instrucción ALU o salto condicional			IF	ID	EX	MEM	WB	
Instrucción carga o almacenamiento			IF	ID	EX	MEM	WB	
Instrucción ALU o salto condicional				IF	ID	EX	MEM	WB
Instrucción carga o almacenamiento				IF	ID	EX	MEM	WB

FIGURA 4.69 Funcionamiento de un pipeline de ejecución múltiple de dos instrucciones con planificación estática (*static two-issue*). Las instrucciones ALU y de transferencias de datos se lanzan al mismo tiempo. Se ha supuesto que se tiene la misma estructura de cinco etapas que la que se usa en un pipeline de ejecución simple (*single-issue*). Aunque esto no es estrictamente necesario, proporciona algunas ventajas. En particular, la gestión de las excepciones y el mantenimiento de un modelo de excepciones precisas, que se complica en los procesadores de ejecución múltiple, se ven simplificados por el hecho de mantener las escrituras en los registros en la etapa final del pipeline.

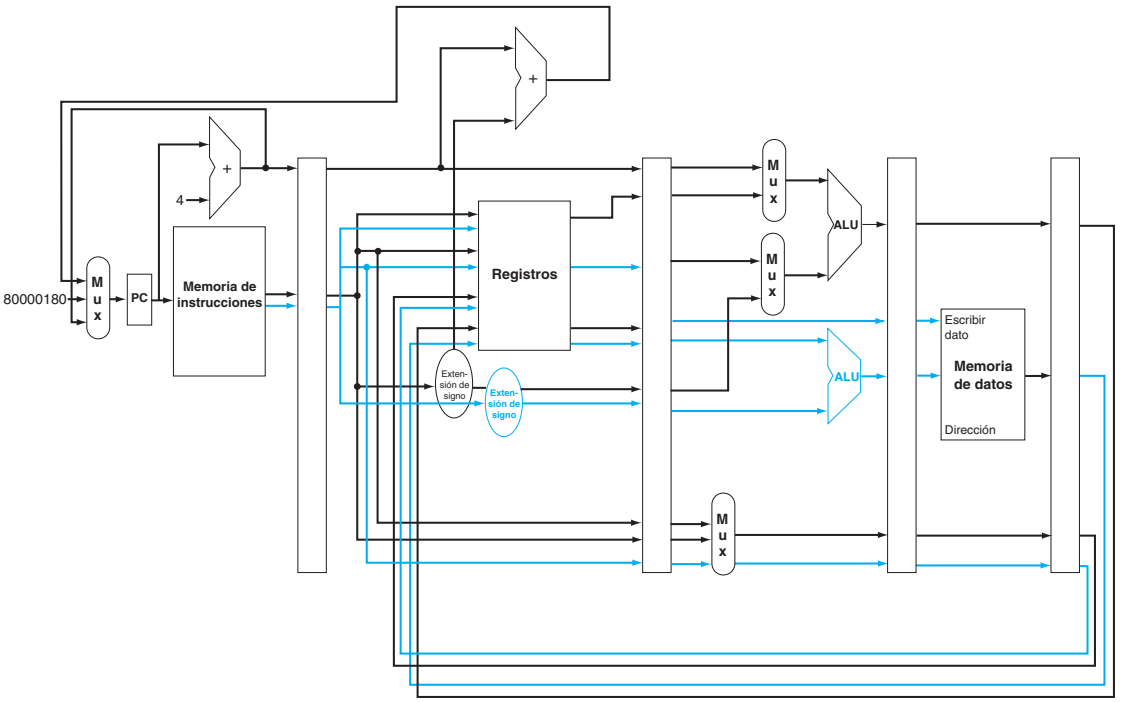


FIGURA 4.68 Camino de datos para la ejecución múltiple de dos instrucciones con planificación estática. Se resaltan los elementos adicionales para hacer que el camino de datos permita dos instrucciones simultáneas: 32 bits más desde la memoria de instrucciones, dos puertos de lectura más y uno más de escritura en el banco de registros, y otra ALU. Suponga que la ALU de la parte inferior calcula las direcciones de las transferencias de datos con memoria y que la ALU de la parte superior gestiona todas las demás operaciones enteras.

las dependencias entre las instrucciones internas de cada paquete de ejecución. Incluso así, un riesgo generalmente forzará a que se bloquee el paquete de ejecución entero que contiene la instrucción dependiente. Tanto si el software debe manejar todos los riesgos como si sólo debe tratar de reducir el porcentaje de riesgos entre diferentes paquetes de ejecución, se refuerza la apariencia de tener una única instrucción compuesta de múltiples operaciones. En el siguiente ejemplo se supondrá la segunda estrategia.

Para ejecutar en paralelo una operación de la ALU y una transferencia de datos a memoria, el primer hardware adicional que sería necesario, además de la habitual lógica de detección de riesgos, es disponer de más puertos de lectura y de escritura en el banco de registros (véase la figura 4.69). En un ciclo podríamos necesitar leer dos registros para la operación ALU y dos más para un almacenamiento, y también podría ser necesario un puerto de escritura para la operación ALU y uno más para una carga. Ya que la ALU está ocupada ejecutando su operación, también se necesitará un sumador adicional para calcular la dirección efectiva de las transferencias de datos a memoria. Sin estos recursos adicionales, el pipeline de dos instrucciones se vería limitado por riesgos estructurales.

Claramente, este procesador de ejecución doble puede llegar a mejorar las prestaciones en un factor 2. Pero para ello es necesario que puedan llegar a solapar su ejecución el doble de instrucciones, y este solapamiento adicional incrementa la pérdida relativa de rendimiento a causa de los riesgos de datos y de control. Por ejemplo, en nuestro pipeline simple de cinco etapas, las cargas tienen una **latencia de uso** de 1 ciclo de reloj, lo

Latencia de uso: número de ciclos de la señal de reloj entre una instrucción de carga y una instrucción que utiliza el resultado de la carga sin bloquear el pipeline.

cual impide que la instrucción siguiente pueda usar el dato inmediatamente y hace que deba bloquearse. En el pipeline de cinco etapas con doble capacidad de ejecución, el resultado de una instrucción de carga tampoco puede usarse en el siguiente ciclo de reloj, y esto significa ahora que las siguientes *dos* instrucciones no pueden usar el dato de la carga inmediatamente, y tendrían que bloquearse. Además, las instrucciones de la ALU que no tenían latencia de uso en el pipeline sencillo de cinco etapas tienen ahora una latencia de uso de 1 ciclo, ya que el resultado no puede utilizarse en la carga o el almacenamiento correpondiente. Para explotar el paralelismo disponible en un procesador de ejecución múltiple de forma eficiente, se necesitan técnicas más ambiciosas de planificación por parte del compilador o del hardware. En un esquema de planificación estática es el compilador el que debe asumir este rol.

Planificación simple de código para ejecución múltiple

EJEMPLO

¿Cómo se debería planificar este lazo de forma estática en un procesador MIPS con ejecución de dos instrucciones?

```
Loop: lw     $t0, 0($s1)     # $t0=elemento de un vector
      addu   $t0,$t0,$s2     # sumar valor escalar en $s2
      sw     $t0, 0($s1)     # almacenar resultado
      addi   $s1,$s1,-4      # decrementar puntero
      bne    $s1,$zero,Loop  # saltar si $s1!=0
```

Reordenar las instrucciones para evitar el máximo número de bloqueos que sea posible. Suponer que los saltos son predichos, de forma que los riesgos de control son gestionados por el hardware.

RESPUESTA

Las primeras tres instrucciones tienen dependencias de datos, así como las dos últimas. La figura 4.70 muestra la mejor planificación posible para estas instrucciones. Observe que sólo un par de instrucciones completan ambas ranuras del paquete de ejecución. Se tardan 4 ciclos por cada iteración del lazo; al ejecutar 5 instrucciones cada 4 ciclos se obtiene un decepcionante CPI de 0.8 frente un máximo valor alcanzable de 0.5, o un IPC de 1.25 frente a 2. Observe también que al calcular CPI o IPC los *nops* ejecutados no se cuentan como instrucciones útiles. Hacer eso podría mejorar el CPI, ¡pero no las prestaciones!

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

FIGURA 4.70 El codigo planificado para ser ejecutado en un pipeline MIPS de dos instrucciones por ciclo. Los huecos vacíos representan nops.

Una técnica importante aplicada por el compilador para conseguir un mayor rendimiento en los lazos es el **desenrollado de lazos (*loop unrolling*)**, que consiste en realizar múltiples copias del cuerpo del lazo. Después del desenrollado, se dispone de más ILP para poder solapar la ejecución de instrucciones que pertenecen a diferentes iteraciones.

Desenrollado de lazos: técnica para obtener mayor rendimiento en los lazos de acceso a vectores, en la que se realizan múltiples copias del cuerpo del lazo para poder planificar conjuntamente la ejecución de instrucciones de diferentes iteraciones.

EJEMPLO

Desenrollado de lazos para pipelines con ejecución múltiple

Investigue cómo se mejora las prestaciones del ejemplo anterior desenrollando el lazo y planificando de nuevo la ejecución. Por simplicidad, suponga que el índice del lazo es múltiplo de cuatro.

RESPUESTA

Se necesita hacer cuatro copias del cuerpo del lazo para poder planificar las instrucciones sin que haya retardos. Después de desenrollar el lazo y de eliminar las instrucciones innecesarias, el lazo contendrá cuatro copias de cada instrucción `lw`, `add` y `sw`, además de una instrucción `addi` y una `bne`. La figura 4.71 muestra el código desenrollado y planificado.

Durante el proceso de desenrollado el compilador introduce el uso de registros adicionales (`$t1`, `$t2`, `$t3`). El objetivo de este proceso, llamado **renombrado de registros (*register renaming*)**, es eliminar aquellas dependencias que no son dependencias de datos verdaderas, pero que pueden conducir a riesgos potenciales o limitar la flexibilidad del compilador para planificar el código. Consideremos cuál habría sido el resultado de desenrollar el código si sólo se hubiera usado `$t0`. Habría copias repetidas en las instrucciones `lw $t0,0($s1)`, `addu $t0,$t0,$s2` seguidas por la instrucción de `sw $t0,4($s1)`. Aunque únicamente se use el registro `$t0`, estas secuencias de instrucciones siguen siendo en realidad completamente independientes (entre un par de instrucciones y el siguiente par de instrucciones no se produce ninguna transferencia de datos). El orden entre las instrucciones está forzado única y exclusivamente por la reutilización de un nombre (el registro `$t0`), en lugar de estar forzado por una dependencia de datos verdadera, y a esto se le conoce como **antidependencia (*antidependence*) o dependencia de nombre (*name dependence*)**.

Renombrado de registros: volver a nombrar registros, mediante el compilador o el hardware, para eliminar las antidependencias.

Renombrar los registros durante el proceso de desenrollado permite que posteriormente el compilador pueda mover estas instrucciones independientes y así planificar mejor el código. El proceso de renombrado elimina las dependencias de nombre, mientras que preserva las dependencias de datos reales.

Antidependencia (dependencia de nombre): ordenación forzada por la reutilización de un nombre, generalmente un registro, en lugar de una dependencia de dato verdadera que acarrea un valor entre dos instrucciones.

Observe que ahora se logra que 12 de las 14 instrucciones del lazo se puedan ejecutar combinadas en una pareja. Cuatro iteraciones del lazo tardan 8 ciclos, que son 2 ciclos por iteración, lo cual supone un CPI de 8/14 = 0.57. El desenrollado de lazos y la subsiguiente planificación de instrucciones dan un factor de mejora de dos, en parte debido a la reducción en el número de instrucciones de control del lazo y en parte debido a la ejecución dual de instrucciones. El coste de esta mejora de las prestaciones es el uso de cuatro registros temporales en vez de uno, además de un incremento significativo del tamaño del código.

Procesadores con ejecución múltiple y planificación dinámica

Los procesadores con planificación dinámica de la ejecución múltiple de instrucciones se conocen también como procesadores **superescalares (*superscalar*)**, o simplemente superescalares. En los procesadores superescalares más simples, las instrucciones se

Superescalar: técnica avanzada de segmentación que permite al procesador ejecutar más de una instrucción por ciclo de reloj.

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

FIGURA 4.71 Código de la figura 4.70 desenrollado y planificado para un procesador MIPS de ejecución dual. Los huecos vacíos representan *nops*. Como la primera instrucción de lazo decrementa $s1 en 16 unidades, las direcciones de donde se cargan los datos son el valor original de $sl, y luego esta dirección menos 4, menos 8 y menos 12.

lanzan a ejecutar en orden, y en un determinado ciclo de reloj el procesador decide si se pueden ejecutar una o más instrucciones nuevas o ninguna. Obviamente, para alcanzar unas buenas prestaciones en esto procesadores, se requiere que el compilador planifique las instrucciones de forma que se eliminen dependencias y se mejore la frecuencia de ejecución. Incluso con esa planificación realizada por el compilador, existe una diferencia importante entre este superescalar sencillo y un procesador VLIW: el hardware garantiza que el código, tanto si está planificado como si no lo está, se ejecutará correctamente. Más aún, independientemente de la estructura del pipeline o de la capacidad del procesador para ejecutar instrucciones simultáneamente, el código compilado siempre funcionará de forma correcta. Esto no es así en algunos diseños VLIW, que requieren que se recompile el código cuando éste se quiere ejecutar en diferentes modelos del procesador. En otros procesadores de planificación estática el código sí que se ejecuta correctamente en las diferentes implementaciones de la arquitectura, pero frecuentemente el rendimiento acaba siendo tan pobre que la recompilación acaba siendo necesaria.

Planificación dinámica del pipeline: soporte hardware para reordenar la ejecución de las instrucciones de modo que se eviten bloqueos.

Muchos superescalares extienden el ámbito de las decisiones dinámicas para incluir la **planificación dinámica del pipeline (*dynamic pipeline scheduling*)**. La planificación dinámica del pipeline escoge las instrucciones a ejecutar en cada ciclo de reloj intentando en lo posible evitar los riesgos y los bloqueos. Comenzaremos con un ejemplo simple que evita un riesgo de datos. Considere la siguiente secuencia de código

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

Aunque la instrucción `sub` está preparada para ser ejecutada, debe esperar a que primero se completen las instrucciones `lw` y `addu`, lo cual puede suponer muchos ciclos si los accesos a memoria son lentos. (El capítulo 5 explica el funcionamiento de las cachés, la razón de que los accesos a memoria sean en ocasiones muy lentos). La planificación dinámica permite evitar estos riesgos parcial o completamente.

Planificación dinámica del pipeline

La planificación dinámica del pipeline escoge las siguientes instrucciones a ejecutar, posiblemente reordenándolas para así evitar bloqueos. En estos procesadores, el pipeline se divide en tres unidades fundamentales: una unidad de búsqueda instrucciones y de decisión de qué instrucciones ejecutar (*issue unit*), múltiples unidades funcionales (12 o incluso más en los diseños de gama alta en el 2008), y una **unidad de confirmación (*commit unit*).** La figura 4.72 muestra el modelo. La primera unidad busca instrucciones, las descodifica y envía cada una de ellas a la unidad funcional que corresponda para ser ejecutada. Cada unidad funcional dispone de búferes, llamados **estaciones de reserva (*reservation stations*)**, que almacenan la operación y los operandos. (En la siguiente sección veremos una alternativa a las estaciones de reserva que usan muchos procesadores recientes). Tan pronto como el búfer contiene todos los operandos de la operación y la unidad funcional está preparada, se ejecuta la instrucción y se calcula el resultado. Una vez obtenido el resultado, se envía a todas las estaciones de reserva que puedan estar esperando por él, además de enviarlo a la unidad de confirmación, que lo guardará temporalmente hasta que sea seguro escribirlo en el banco de registros o, si es un almacenamiento, escribirlo en memoria. Este búfer que se encuentra en la unidad de confirmación, llamado con frecuencia **búfer de reordenación (*reorder buffer*)**, se usa también para proporcionar operandos de una forma muy similar a como hace la lógica de anticipación de resultados en un procesador planificado de forma estática. Una vez que el resultado es confirmado en el banco de registros, puede ser accedido directamente desde allí, igual que se hace en un pipeline normal.

Unidad de confirmación: en un procesador con ejecución fuera de orden, es la unidad que decide cuando es seguro confirmar el resultado de una operación y hacerlo visible, bien a los registros, bien a la memoria.

Estación de reserva: búfer acoplado a las unidades funcionales que almacena temporalmente los operandos y las operaciones pendientes de ser ejecutadas.

Búfer de reordenación: búfer que almacena temporalmente los resultados producidos en un procesador planificado de forma dinámica hasta que es seguro almacenarlos en memoria o en un registro y hacerlos así visibles.

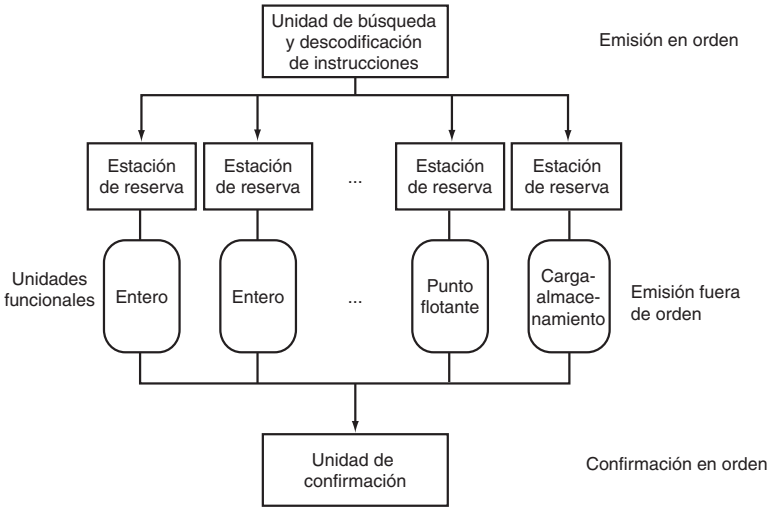


FIGURA 4.72 Las tres unidades principales de un pipeline planificado dinámicamente. El paso final de actualización del estado también se llama jubilación o graduación.

La combinación de almacenar los operandos temporalmente en las estaciones de reserva y de almacenar los resultados temporalmente en el búfer de reordenación supone una forma de renombrado, justamente como la usada por el compilador en el ejemplo anterior del desenrollado de lazos mostrado en la página 397. Para ver cómo funciona todo esto de forma conceptual es necesario considerar los siguientes pasos:

1. Cuando se lanza una instrucción a la unidad de ejecución, si alguno de sus operandos se encuentra en el banco de registros o en el búfer de reordenación, entonces se copia inmediatamente a la estación de reserva correspondiente, donde se mantiene temporalmente hasta que todos los operandos y la unidad funcional están disponibles. Para la instrucción que se ha lanzado, la copia del operando que hay en el registro ya no es necesaria, y se puede permitir realizar una escritura sobre ese registro y sobrescribir su valor actual.
2. Si un operando no está en el banco de registros ni en el búfer de reordenación, entonces debe esperarse a que sea producido por una unidad funcional. Se toma nota de la unidad funcional que producirá el resultado y, cuando finalmente esto ocurra, se copiará directamente desde la unidad funcional, mediante la red de anticipación, a la estación de reserva que está esperando.

Estos pasos usan de forma efectiva el búfer de reordenación y las estaciones de reserva para implementar el renombrado de registros.

Conceptualmente se puede pensar que un pipeline con planificación dinámica analiza la estructura del flujo de datos del programa. El procesador entonces ejecuta las instrucciones en un cierto orden que preserva el orden del flujo de datos del programa. Este estilo de ejecución se llama **ejecución fuera de orden**, ya que las instrucciones pueden ser ejecutadas en un orden diferente del orden en que fueron capturadas.

Ejecución fuera de orden: propiedad de la ejecución segmentada que permite que el bloqueo de la ejecución de una instrucción no provoque que las instrucciones que le sigan tengan que esperarse.

Para que los programas se comporten como si fueran ejecutados en un pipeline simple con ejecución ordenada, las etapas de búsqueda de instrucciones y de descodificación se deben realizar también en orden, lo cual permite tomar nota de las dependencias, y la unidad de confirmación de la ejecución también debe escribir los resultados en registros y en memoria en el orden original del programa. Este modo conservador de operación se denomina **confirmación en orden** (*in-order commit*). De este modo, si ocurre una excepción, el computador podrá apuntar a la última instrucción ejecutada, y los únicos registros que habrán sido actualizados serán aquellos escritos por las instrucciones anteriores a la que ha causado la excepción. Aunque las etapas denominadas *front-end* (búsqueda y lanzamiento de instrucciones) y la etapa denominada *back-end* (confirmación) del pipeline funcionen en orden, las unidades funcionales son libres de iniciar la ejecución de las instrucciones cuando los datos que se necesitan estén disponibles. Hoy en día, todos los pipelines planificados dinámicamente usan finalización en orden.

Confirmación en orden: confirmación de los resultados de la ejecución segmentada que se escriben en el estado visible para el programador en el mismo orden en que las instrucciones se traen de memoria.

La planificación dinámica frecuentemente se extiende para incluir especulación basada en hardware, especialmente en lo que se refiere a los resultados de los saltos. Prediciendo el sentido y la dirección destino de un salto condicional, un procesador con planificación dinámica puede seguir buscando y ejecutando instrucciones a lo largo del camino del flujo de control predicho. Puesto que las instrucciones se confirman en orden, antes de que alguna instrucción en el camino predicho sea confirmada se sabrá si el salto ha sido correctamente predicho o no. Es también fácil para un procesador segmentado con ejecución especulativa y planificación dinámica que dé soporte a la especulación de direcciones de las instrucciones de carga, permitiendo la reordenación entre cargas y almacenamientos, usando la unidad de confirmación para evitar los problemas de una especulación incorrecta. En la siguiente sección revisaremos el uso de la ejecución con planificación dinámica y especulativa en el diseño del AMD Opteron X4 (Barcelona).

Comprender las prestaciones de los programas

Puesto que los compiladores son capaces de planificar el código para sortear las dependencias de datos, uno puede preguntarse para qué los procesadores superescalares usan la planificación dinámica. Existen tres razones principales. En primer lugar, no todos los bloqueos en el pipeline son predecibles. En particular, los fallos de caché (véase capítulo 5) son la causa de muchos bloqueos impredecibles. La planificación dinámica permite que el procesador oculte el efecto de algunos de estos bloqueos al seguir ejecutando instrucciones mientras el bloqueo no finaliza.

En segundo lugar, si un procesador utiliza la predicción dinámica de saltos para especular con los resultados de los saltos, entonces no será posible conocer el orden exacto de las instrucciones en tiempo de compilación, puesto que el orden dependerá del propio comportamiento de los saltos. La incorporación de la especulación dinámica para aprovechar mejor el paralelismo de instrucciones ILP sin disponer a su vez de una planificación dinámica de la ejecución, restringiría de forma significativa el beneficio de la especulación.

En tercer lugar, como la latencia en el pipeline y la anchura de ejecución varían de una implementación a otra, la mejor forma de compilar una secuencia de código también varía. Por ejemplo, la mejor manera de planificar la ejecución de una secuencia de instrucciones dependientes se ve afectada tanto por la anchura de ejecución como por la latencia de las operaciones. La estructura del pipeline afecta al número de veces que un lazo debe ser desenrollado para evitar los bloqueos, así como al proceso de renombrado de registros realizado por el compilador. La planificación dinámica permite al hardware ocultar muchos de estos detalles. De este modo, los usuarios y los distribuidores de software no necesitan preocuparse de tener diferentes versiones de un programa para diferentes implementaciones del mismo repertorio de instrucciones. De forma similar, el código generado para versiones previas de un procesador podrá aprovechar mucha parte de las ventajas de una nueva implementación del procesador sin necesidad de tener que recompilarlo.

IDEA clave

Tanto la segmentación como la ejecución múltiple de instrucciones intentan aprovechar el paralelismo de instrucciones ILP para aumentar el ritmo máximo de ejecución de instrucciones. Pero las dependencias de datos y de control determinan un límite superior en las prestaciones que se puede conseguir de forma sostenida, ya que a veces el procesador debe esperar a que una dependencia se resuelva. Las estrategias para aprovechar el paralelismo de instrucciones ILP que están centradas en el software dependen de la habilidad del compilador para encontrar estas dependencias y reducir sus efectos, mientras que las estrategias centradas en el Hardware dependen de extensiones a los mecanismos de segmentación y de lanzamiento múltiple de instrucciones. La especulación, tanto si es realizada por el compilador como por el harware, incrementa la cantidad de paralelismo de instrucciones ILP que puede llegar a ser aprovechado, aunque se debe tener cuidado, ya que la especulación incorrecta puede llegar a reducir las prestaciones.

Interfaz hardware software

Los modernos microprocesadores de altas prestaciones son capaces de lanzar a ejecutar bastantes instrucciones por ciclo, pero desafortunadamente es muy difícil alcanzar esa capacidad de forma sostenida. Por ejemplo, a pesar de que existen procesadores que ejecutan 4 y 6 instrucciones por ciclo, muy pocas aplicaciones son capaces de ejecutar de forma sostenida más de dos instrucciones por ciclo. Esto se debe básicamente a dos razones.

Primero, dentro del pipeline, los cuellos de botella más importantes se deben a dependencias que no pueden evitarse o reducirse, y de este modo el paralelismo entre instrucciones y la velocidad sostenida de ejecución se reducen. Aunque poca cosa se puede hacer con las dependencias reales, frecuentemente ni el compilador ni el hardware son capaces de asegurar si existe o no una dependencia y, de una forma conservadora, deben suponer que la dependencia existe. Por ejemplo, un código que haga uso de punteros, particularmente si lo hace de forma tal que crea muchas formas de referenciar la misma variable (*aliasing*), dará lugar a muchas dependencias potenciales implícitas. En cambio, la mayor regularidad de los accesos a un vector frecuentemente permite que el compilador deduzca que no existen dependencias. De forma similar, las instrucciones de salto que no pueden ser predichas de forma precisa tanto en tiempo de ejecución como en tiempo de compilación limitarán las posibilidades de aprovechar el paralelismo de instrucciones ILP. En muchas ocasiones existe paralelismo ILP adicional disponible, pero al encontrarse muy alejado (a veces a una distancia de miles de instrucciones) la habilidad del compilador o del hardware para encontrar el paralelismo ILP es muy limitada.

En segundo lugar, las deficiencias del sistema de memoria (el tema del capítulo 7) también limitan la habilidad de mantener el pipeline lleno. Algunos bloqueos producidos por el sistema de memoria pueden ser ocultados, pero si se dispone de una cantidad limitada de paralelismo ILP, esto también limita la cantidad de estos bloqueos que se puede ocultar.

Eficiencia energética y segmentación avanzada

El inconveniente de aumentar la explotación del paralelismo a nivel de instrucciones vía la planificación dinámica con búsqueda de múltiples instrucciones y la especulación es la eficiencias energética. Cada innovación fue capaz de transformar los transitores adicionales en más prestaciones, pero a menudo se hizo de forma muy ineficiente. Ahora que tenemos que luchar contra el muro de la potencia, estamos buscando diseños con varios procesadores por chip, donde los procesadores no tienen pipelines tan profundos ni técnicas de especulación tan agresivas como sus predecesores.

La opinión general es que aunque los procesadores más sencillos no son tan rápidos como sus hermanos más sofisticados, tienen mejores prestaciones por vatio, de modo que pueden proporcionar mejores prestaciones por chip cuando los diseños están limitados más por la potencia disipada que por el número de transistores.

La figura 4.73 muesra el número de etapas, el ancho de emisión, el nivel de especulación, la frecuencia del reloj, el número de núcleos por chip y la potencia de varios microprocesadores pasados y recientes. Obsérvese el descenso del número de etapas y la potencia a medida que los fabricantes introducen diseños multinúcleo.

Microprocesador	Año	Frecuencia de reloj	Etapas	Ancho de ejecución	Fuera de orden/ especulación	Núcleos/ chip	Potencia
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Si	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Si	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Si	1	103 W
Intel Core	2006	2930 MHz	14	4	Si	2	75 W
Sun UltraSPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

FIGURA 4.73 Relación de microprocesadores Intel y Sun en términos de complejidad del pipeline, número de núcleos y potencia. Entre las etapas del pipeline del Pentium4 no se incluyen las etapas de confirmación (commit). Si estas etapas se incluyen, los pipelines del Pentium 4 serían todavía más profundos.

Extensión: Los controles de una unidad de confirmación actualizan el banco de registro y la memoria. Algunos procesadores con planificación dinámica actualizan el banco de registros de forma inmediata durante la ejecución, utilizando registros adicionales para implementar el renombrado y manteniendo una copia de los contenidos anteriores hasta que la instrucción que actualiza los registros ya no es especulativa. Otros procesadores guardan los resultados, típicamente en una estructura llamada búfer de reordenación, y la actualización real de los registros se realiza más tarde como parte de la finalización. Los almacenamientos en memoria deben mantenerse en un búfer hasta la finalización, bien en un búfer de almacenamientos (véase el capítulo 5) o bien en un búfer de reordenación. La unidad confirmación permite que la instrucción de almacenamiento escriba el contenido del búfer en memoria cuando el búfer tiene una dirección y un dato válidos y el almacenamiento ya no depende de saltos condicionales con predicción.

Extensión: Los accesos a memoria se benefician de las caches no bloqueantes (*nonblocking caches*), que continúan procesando accesos a la cache durante un fallo de cache (véase el capítulo 5). Los procesadores con ejecución fuera-de-orden necesitan que la cache esté diseñada de forma que permita la ejecución de instrucciones durante un fallo de cache.

Autoevaluación

Diga si las siguientes técnicas o componentes se asocian fundamentalmente con estrategias bien software o hardware de aprovechamiento del paralelismo de instrucciones ILP. En algunos casos, la respuesta puede ser ambas estrategias.

1. Predicción de saltos
2. Ejecución múltiple
3. VLIW
4. Superescalar
5. Planificación dinámica
6. Ejecución fuera-de-orden
7. Especulación
8. Búfer de reordenación
9. Renombrado de registros

4.11 Casos reales: El pipeline del AMD Opteron X4 (Barcelona)

Como la mayoría de los computadores modernos, los microprocesadores x86 incorporan técnicas de segmentación sofisticadas. Sin embargo, estos procesadores están afrontando todavía el reto de implementar el complejo repertorio de instrucciones x86, descrito en el capítulo 2. Tanto AMD como Intel capturan instrucciones x86 y las traducen internamente a instrucciones tipo MIPS, que AMD llama operaciones RICS (Rops) e Intel llama microoperaciones. Las operaciones RISC se ejecutan en un pipeline sofisticado con planificación dinámica y especulativo, capaz de mantener una velocidad de tres operaciones RISC por ciclo de reloj en el AMD Opteron X4 (Barcelona). En esta sección nos centramos en el pipeline de las operaciones RISC.

Cuando se considera el diseño de sofisticados procesadores con planificación dinámica, el diseño de las unidades funcionales, de la caché, del banco de registros, de la lógica para lanzar a ejecutar las instrucciones, y el control total del pipeline está muy interrelacionado, haciendo muy difícil separar lo que es sólo el camino de datos de la totalidad del pipeline. Por esta razón, muchos ingenieros e investigadores han adoptado el término **microarquitectura** para referirse a la arquitectura interna detallada de un procesador. La figura 4.74 muestra la microarquitectura del X4, centrándose en las estructuras que se utilizan para ejecutar las operaciones RISC.

Microarquitectura: organización del procesador, que incluye sus principales unidades funcionales, sus elementos de interconexión y el control.

Otra forma de considerar el X4 es mirando las etapas de segmentación que atraviesa una instrucción típica. La figura 4.75 muestra la estructura del pipeline y el número típico de ciclos de reloj que se gastan en él. Por supuesto, el número de ciclos variará debido a la naturaleza de la planificación dinámica, así como a los requerimientos individuales propios de cada operación RISC.

Extensión: El Pentium 4 usa un esquema para resolver las antidependencias y para arreglar las especulaciones incorrectas. Este esquema utiliza un búfer de reordenación junto al renombrado de registros. El renombrado de registros renombra de forma explícita los registros de la arquitectura (*architectural registers*) del procesador (16 en el caso de la versión de 64 bits de la arquitectura X86) a un conjunto mayor de registros físicos (72 en el X4). El X4 usa el renombrado de registros para eliminar las antidependencias. El renombrado de registros necesita que el procesador mantenga un asignación entre los registros de la arquitectura y los registros físicos, indicando qué registro físico es la copia más actual de cada registro de la arquitectura. Si se mantiene la información de los renombrados que se han producido, el renombrado de registros ofrece una estrategia alternativa para recuperarse cuando se produce una especulación incorrecta: simplemente se deshacen los mapeos que han ocurrido desde la primera instrucción producto de la especulación incorrecta. Esto provocará que se retorne al estado del procesador que se tenía después de la última instrucción correctamente ejecutada, logrando recuperar la asignación correcto entre registros de la arquitectura y físicos.

Registros de la arquitectura: registros visibles en el repertorio de instrucciones del procesador; por ejemplo, en MIPS, estos son 32 registros enteros y 16 registros de punto flotante.

Autoevaluación

Las siguientes afirmaciones, ¿son ciertas o falsas?

1. El pipeline de ejecución múltiple del Opteron X4 ejecuta directamente instrucciones x86.
2. El X4 usa planificación dinámica pero sin especulación.

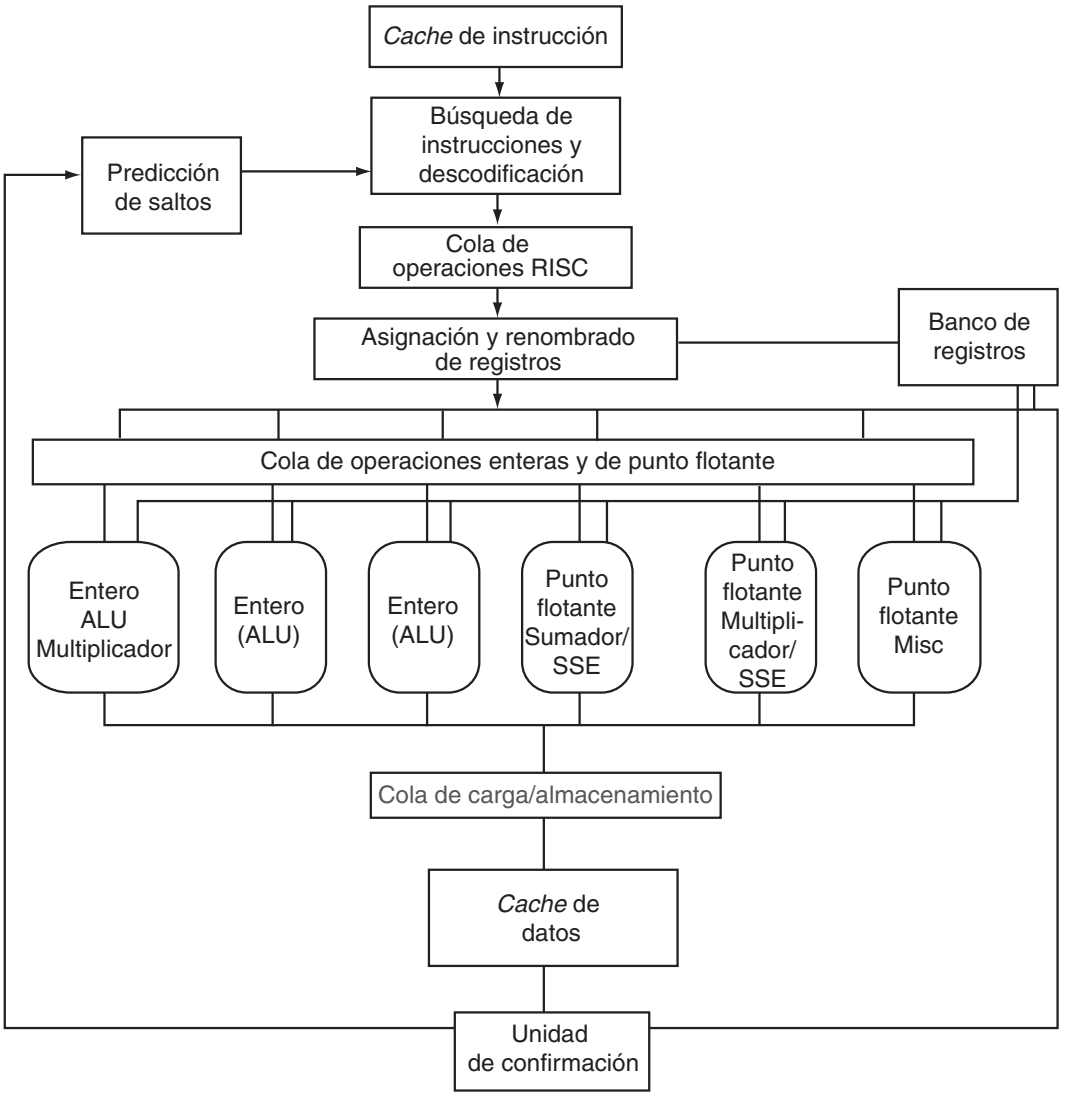


FIGURA 4.74 La microarquitectura del AMD Opteron X4. Las amplias colas disponibles permiten que hasta 106 operaciones RISC estén pendientes de entrar en ejecución, incluyendo 24 operaciones de enteros, 36 operaciones de punto flotante/SSE y 44 cargas y almacenamientos. Las unidades de carga y almacenamiento están realmente separadas en dos partes, la primera parte se encarga del cálculo de la dirección en unidad ALU de enteros y la segunda parte es responsable de la referencia a memoria. Hay un amplio circuito de anticipación entre unidades funcionales; puesto que el pipeline es dinámico en lugar de estático, la anticipación se hace etiquetando los resultados y rastreando los operandos fuente, para detectar cuando un resultado ha sido obtenido por una instrucción en una de las colas que están esperando por este resultado.

3. La microarquitectura del X4 tiene muchos más registros de los que requiere la arquitectura x86.
4. El pipeline del X4 tiene menos de la mitad de etapas que el Pentium 4 Prescott (véase la figura 4.73).

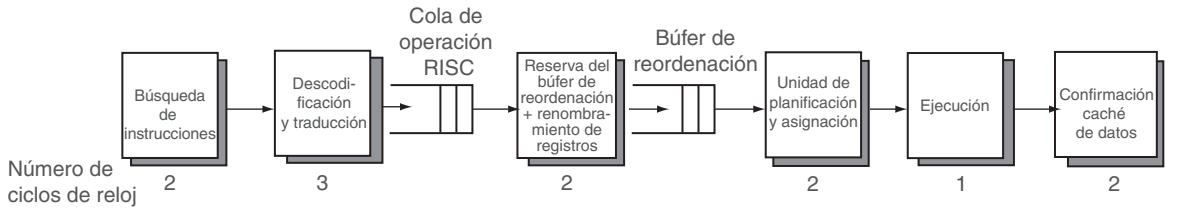


FIGURA 4.75 El pipeline del Opteron X4 mostrando el flujo de una instrucción típica y el número de ciclos de reloj de los principales pasos del pipeline de 12 etapas de las operaciones RISC de enteros. La cola de ejecución de punto flotante tiene una longitud de 17 etapas. Se muestran también los principales búferes donde esperan las operaciones RISC.

Comprender las prestaciones de los programas

El Opteron X4 combina un pipeline de 12 etapas con una ejecución múltiple agresiva para poder alcanzar altas prestaciones. El impacto de las dependencias de datos se reduce haciendo que las latencias entre operaciones consecutivas (*back-to-back*) sean reducidas. Para los programas que se ejecutan en este procesador, ¿cuáles son los cuellos de botella potenciales más serios para las prestaciones? La siguiente lista incluye algunos de estos problemas potenciales para las prestaciones; los tres últimos también se pueden aplicar de alguna manera a cualquier procesador segmentado de altas prestaciones.

- El uso de instrucciones x86 que no se traducen en pocas operaciones RISC simples.
- Las instrucciones de salto difíciles de predecir, que causan bloqueos debido a fallos de predicción y hacen que se deba reiniciar la ejecución por especulación errónea.
- Largas dependencias, generalmente causadas por instrucciones con latencias elevadas o por fallos en la caché de datos, que dan lugar a bloqueos durante la ejecución.
- Retrasos en los accesos a la memoria que provocan que el procesador se bloquee (véase capítulo 5).

4.12 Tema avanzado: una introducción al diseño digital utilizando un lenguaje de descripción hardware para describir y modelar un pipeline y más figuras sobre segmentación

El diseño digital moderno se hace utilizando lenguajes de descripción hardware y modernas herramientas de síntesis asistida por computador que son capaces de crear diseños hardware detallados a partir de una descripción con bibliotecas y síntesis lógica. Se han escrito libros enteros sobre estos lenguajes y su utilización en diseño digital. Esta sección, incluida en el CD, da una breve introducción y muestra como un lenguaje de descripción hardware, Verilog en este caso, puede utilizarse para des-

cribir el control del MIPS tanto a nivel de comportamiento como a un nivel adecuado para la síntesis del hardware. Así, proporciona una serie de modelos de comportamiento para el pipeline de cinco etapas de MIPS en Verilog. El modelo inicial ignora los riesgos y las variaciones posteriores resaltan los cambios introducidos para incorporar la anticipación, los riesgos de datos y los riesgos de los saltos.

Proporciona, también, alrededor de una docena de figuras con la representación de ciclo único del pipeline para aquellos lectores interesados en conocer con más detalle qué ocurre en el pipeline para unas cuantas secuencias de instrucciones MIPS.

4.13 Falacias y errores habituales

Falacia: La segmentación es sencilla.

Nuestros libros certifican lo delicado que es conseguir una ejecución segmentada correcta. La primera edición de nuestro libro avanzado tenía un error en el *pipeline*, a pesar de que fue revisado con anterioridad por más de 100 personas y fue probado en las aulas de 18 universidades. El error sólo fue descubierto cuando alguien intentó construir el computador explicado en el libro. El hecho de que el código en lenguaje Verilog que describe un pipeline como el del Opteron X4 será de miles de líneas es una indicación de su complejidad. ¡Tenga cuidado!

Falacia: Las ideas de la segmentación se pueden llevar a cabo independientemente de la tecnología.

Cuando el número de transistores por chip y la velocidad de los transistores hicieron que la segmentación de cinco etapas fuera la mejor solución, el salto retardado (véase la primera sección Extensión de la página 381) fue considerado una solución sencilla para los conflictos de control. Con pipelines más largos, ejecución superescalar y predicción dinámica de saltos, esa técnica es ahora redundante. A comienzos de los 90, la planificación dinámica del pipeline necesitaba demasiados transistores y no fue utilizada para aumentar las prestaciones, pero a medida que la cantidad de transistores disponibles se continuó doblando y la lógica se fue haciendo cada vez más rápida que la memoria, tuvo más sentido el uso de múltiples unidades funcionales y de la planificación dinámica. Hoy en día, la preocupación por la potencia está llevando a diseños menos agresivos.

Error habitual: Olvidar que el diseño del repertorio de instrucciones puede afectar negativamente a la segmentación.

Muchas de las dificultades de la segmentación se deben a complicaciones implícitas en el repertorio de instrucciones. Estos son algunos ejemplos:

- Si la longitud y los tiempos de ejecución de las instrucciones son muy variables, en un diseño segmentado a nivel del repertorio de instrucciones se puede crear un desequilibrio entre las etapas de segmentación que complique seriamente la detección de conflictos. Este problema fue resuelto por primera vez a finales de de la década de 1980 en el DEC VAX 8500, usando para ello el mismo esquema de microsegmentación que emplea actualmente el Opteron 4. Por supuesto, la sobrecarga de tener que transformar y mantener la correspondencia entre instrucciones reales y microoperaciones se sigue manteniendo.

- Los modos de direccionamiento muy sofisticados pueden conducir a diferentes tipos de problemas. Los modos de direccionamiento que actualizan registros complican la detección de conflictos. Otros modos de direccionamiento que requieren múltiples accesos a memoria complican sustancialmente el control de la segmentación y hacen difícil mantener las instrucciones avanzando por el pipeline de forma uniforme.

Quizás el mejor ejemplo es el DEC Alpha y el DEC NVAX. Con una tecnología comparable, la nueva arquitectura del repertorio de instrucciones del Alpha permitió una implementación cuyo rendimiento es más del doble de rápido que la del NVAX. En otro ejemplo, Bhandarkar y Clark [1991] compararon el MIPS M/2000 y el DEC VAX 8700 contando el número total de ciclos de los programas de evaluación SPEC; concluyeron que, a pesar de que el MIPS M/2000 ejecuta más instrucciones, el VAX ejecuta un promedio de 2.7 veces más ciclos, por lo que el MIPS es más rápido.

Un noventa por ciento de la sabiduría consiste en ser sabios a tiempo.

Proverbio americano

4.14 Conclusiones finales

Como hemos visto en este capítulo, tanto el camino de datos como el control de un procesador pueden diseñarse tomando como punto de partida la arquitectura del repertorio de instrucciones y una comprensión de las características básicas de la tecnología. En la sección 4.3 vimos cómo el camino de datos de un procesador MIPS podía construirse basándonos en la arquitectura y en la decisión de hacer una implementación de ciclo único. Desde luego, la tecnología subyacente afecta también a muchas decisiones de diseño, imponiendo los componentes que pueden utilizarse en el camino de datos, y determinando si una implementación de ciclo único tiene sentido.

Latencia de instrucciones: tiempo de ejecución inherente a una instrucción.

La segmentación mejora la productividad, pero no el tiempo de ejecución inherente, o **latencia de instrucciones**; para algunas instrucciones la latencia es similar en longitud a la de la estrategia de ciclo único. La ejecución múltiple de instrucciones añade un hardware adicional al camino de datos para permitir que varias instrucciones comiencen en cada ciclo de reloj, pero a costa de un aumento en la latencia efectiva. La segmentación se presentó como una técnica para reducir la duración del ciclo de reloj del camino de datos de ciclo único. En comparación, la ejecución múltiple de instrucciones se centra claramente en la reducción de los ciclos por instrucción (CPI).

Tanto la segmentación como la ejecución múltiple de instrucciones intentan aprovechar el paralelismo entre las instrucciones. La presencia de dependencias de datos y de control, que pueden convertirse en conflictos, representan las limitaciones más importantes para la utilización del paralelismo. La planificación de instrucciones y la especulación, tanto en hardware como en software, son las técnicas principales que se usan para reducir el impacto de las dependencias en las prestaciones.

El cambio realizado a mediados de la década de 1990 hacia pipelines más largos, ejecución múltiple de instrucciones y planificación dinámica ha ayudado a mantener el 60% de incremento anual en el rendimiento de los procesadores con el que nos hemos visto beneficiados desde principios de los años 1980. Como se ha mencionado en el capítulo 1, estos microprocesadores mantenían el modelo de programación secuencial,

pero chocaron con el muro de la potencia. Así, los fabricantes se vieron forzados a introducir los multiprocesadores, que explotan el paralelismo en niveles mucho más gruesos (el objetivo del capítulo 7). Esta tendencia ha causado también que los diseñadores reconsideren las implicaciones del binomio potencia-prestaciones de algunas de las innovaciones introducidas desde mediados de la década de 1990, resultando en una simplificación de los pipelines en las microarquitecturas más recientes.

Para mantener los avances en las prestaciones mediante la utilización de procesadores paralelos, la ley de Amdahl sugiere que será otra parte del sistema la que se convertirá en cuello de botella. Ese cuello de botella es el tema del siguiente capítulo: el sistema de memoria.

4.15 Perspectiva histórica y lecturas recomendadas

Esta sección, que aparece en el CD, discute la historia de los primeros procesadores segmentados, los primeros superescalares, el desarrollo de las técnicas de ejecución fuera de orden y especulativa, así como el importante desarrollo de la tecnología de compiladores que les acompaña.

4.16 Ejercicios

Contribución de Milos Prvulovic, Georgia Tech.

Ejercicio 4.1

En la implementación básica de ciclo único, instrucciones diferentes utilizan bloques hardware diferentes. Los tres problemas siguientes en este ejercicio se refieren a las siguientes instrucciones:

	Instrucción	Interpretación
a.	`add Rd, Rs, Rt`	`Reg[Rd]=Reg[Rs]+Reg[Rt]`
b.	`lw Rt,Offs(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+Offs]`

4.1.1 [5]<4.1> ¿Cuáles son los valores de las señales de control generadas por el control de la figura 4.2 para esta instrucción?

4.1.2 [5]<4.1> ¿Qué recursos (bloques) hacen algo útil para esta instrucción?

4.1.3 [10]<4.1> ¿Qué recursos (bloques) producen salidas que no son útiles para esta instrucción? ¿Qué recursos no generan ninguna salida para esta instrucción?

Unidades de ejecución y bloques de lógica digital diferentes tienen latencias (tiempo necesario para hacer su trabajo) diferentes. En la figura 4.2 hay siete clases de bloques principales. La latencia mínima de una instrucción está determinada por la latencia de los bloques a lo largo de su camino crítico (el de mayor latencia). En los tres problemas siguientes en este ejercicio se suponen las siguientes latencias:

	I-mem	Add	Mux	ALU	Regs	D-Mem	Control
a.	400ps	100ps	30ps	120ps	200ps	350ps	100p
b.	500ps	150ps	100ps	180ps	220ps	1000ps	65ps

4.1.4 [5]<4.1> ¿Cuál es el camino crítico de una instrucción AND?

4.1.5 [5]<4.1> ¿Cuál es el camino crítico de una instrucción de carga (LD)?

4.1.6 [5]<4.1> ¿Cuál es el camino crítico de una instrucción BEQ?

Ejercicio 4.2

La implementación de ciclo único básica del MIPS mostrada en la figura 4.2 solamente puede ejecutar algunas instrucciones. Se pueden añadir nuevas instrucciones a la ISA, pero la decisión de hacerlo depende de, entre otras cosas, el coste y la complejidad adicional que se introduce en el camino de datos y el control del procesador. Los tres primeros problemas de este ejercicio se refieren a esta nueva instrucción:

	Instrucción	Interpretación
a.	`add3 Rd, Rs, Rt, Rx`	`Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]`
b.	`sll Rt, Rd, Shift`	`Reg[Rd]=Reg[Rt] << Shift` (desplazamiento a la izquierda de Shift bits)

4.2.1 [10]<4.1> ¿Cuáles de los bloques existentes pueden utilizarse, en caso de haya alguno, para esta instrucción?

4.2.2 [10]<4.1> ¿Qué bloques funcionales nuevos, en caso de haya alguno, se necesitan para esta instrucción?

4.2.3 [10]<4.1> ¿Qué nuevas señales, en caso de haya alguna, es necesario añadir a la unidad de control para poder ejecutar esta instrucción?

Cuando los diseñadores de procesadores consideran la posibilidad de añadir una mejora al camino de datos del procesador, la decisión sobre incorporarla o no depende de una solución de compromiso entre el coste y las prestaciones. En los tres problemas siguientes supondremos que partimos de un camino de datos como la de la figura 4.2, donde la latencias de los bloques I-Mem, Add, Mux, ALU, Regs, D-Mem y Control son 400ps, 100ps, 30ps, 120ps, 200ps, 350ps y 100ps, respectivamente, y el coste de cada bloque es 1000, 30, 10, 100, 200, 2000 y

500, respectivamente. Los tres siguientes problemas de este ejercicio se refieren a esta mejora del procesador:

	Mejora	Latencia	Coste	Beneficio
a.	Sumador más rápido	–20ps en cada unidad de suma	+20 para cada unidad de suma	Reemplaza las unidades de suma por unidades más rápidas.
b.	Más registros	+100ps en acceso a registros	+200 para los registros	Se necesitan menos cargas y almacenamientos para salvar y recuperar los valores de los registros. Se reduce el número de instrucciones en un 5%.

4.2.4 [10]<4.1> ¿Cuál es la duración del ciclo de reloj con y sin la mejora?

4.2.5 [10]<4.1> ¿Cuál es la aceleración que se alcanza con esta mejora?

4.2.6 [10]<4.1> Compare la relación coste/prestaciones con y sin la mejora.

Ejercicio 4.3

Los problemas de este ejercicio se refieren al siguiente bloque lógico:

	Bloque lógico
a.	Pequeña memoria de instrucciones (I-Mem) con cuatro palabras de 8 bits
b.	Pequeño banco de registros con dos registros de 8 bits

4.3.1 [5]<4.1, 4.2> Este bloque, ¿está formado sólo de lógica, sólo de biestables o de ambos?

4.3.2 [20]<4.1, 4.2> Muester la implementación de este bloque, utilizando únicamente puertas AND, OR, inversores y biestables tipo D.

4.3.3 [10]<4.1, 4.2> Repita el problema 4.3.2, pero con puertas AND y OR de dos entradas, inversores y biestables tipo D.

El coste y la latencia de una lógica digital depende de los elementos lógicos básicos (puertas) disponibles y de sus propiedades. Los tres problemas siguientes de este ejercicio se refieren a estas puertas, latencias y costes:

	Inversor		AND, OR de 2 entradas		Cada entrada adicional de una AND, OR		Biestables D	
	Latencia	Coste	Latencia	Coste	Latencia	Coste	Latencia	Coste
a.	20ps	1	30ps	2	+0ps	+1	40ps	6
b.	50ps	1	100ps	2	+40ps	+1	160ps	2

4.3.4 [5]<4.1, 4.2> ¿Cuál es la latencia de la implementación del problema 4.3.2?

4.3.5 [5]<4.1, 4.2> ¿Cuál es el coste de la implementación del problema 4.3.2?

4.3.6 [20]<4.1, 4.2> Cambie el diseño para primero minimizar la latencia y después para minimizar el coste. Compare el coste y la latencia de estos dos diseños optimizados.

Ejercicio 4.4

Cuando se implementa una expresión lógica con componentes digitales, si un cierto operador no está disponible como un componente digital, se deben utilizar las puertas lógicas disponibles para implementarlo. Los problemas de este ejercicio se refieren a las siguientes expresiones lógicas:

	Señal de control 1	Señal de control 2
a.	(((A OR B) OR C) OR (A AND C)) OR (A AND B)	(A OR B) OR C
b.	(((A OR B) XOR B) OR (A OR C)) OR (A AND B)	A AND B

4.4.1 [5]<4.2> Implemente la lógica de la señal de control 1. No reorganice la expresión para optimizarla e implementarla con inversores y puertas AND, OR y XOR de 2 entradas.

4.4.2 [10]<4.2> Suponiendo que todas las puertas tiene la misma latencia, ¿cuál es la longitud (en puertas) del camino crítico de la implementación del problema 4.4.1?

4.4.3 [10]<4.1, 4.2> Cuando se implementan varias expresiones lógicas es posible reducir el coste compartiendo varias señales. Repita el problema 4.4.1 pero implemente las dos expresiones, señal de control 1 y señal de control 2, y comparta componentes entre las dos expresiones cuando sea posible.

Para los tres siguientes problemas de este ejercicio se supondrá que se dispone de los siguientes elementos lógicos, con la latencia y el coste indicados:

	Inversor		AND de 2 entradas		OR de 2 entradas		XOR de 2 entradas	
	Latencia	Coste	Latencia	Coste	Latencia	Coste	Latencia	Coste
a.	20ps	1	30ps	2	34ps	3	40ps	6
b.	50ps	1	100ps	2	120ps	2	150ps	2

4.4.4 [10]<4.2> ¿Cuál es la longitud del camino crítico de la implementación del problema 4.4.3?

4.4.5 [10]<4.2> ¿Cuál es el coste de la implementación del problema 4.4.3?

4.4.6 [10]<4.2> ¿Qué fracción del coste se ha ahorrado al implementar las dos señales de control juntas, en lugar de por separado, en el problema 4.4.3?

Ejercicio 4.5

El objetivo de este ejercicio es ayudarle a que se familiarice con el diseño y el funcionamiento de circuitos lógicos secuenciales. Los problemas que se plantean se refieren al funcionamiento de la siguiente ALU:

	Función de la ALU
a.	Suma uno (X+1)
b.	Desplazamiento a la izquierda de 2 bits (X<<2)

4.5.1 [20]<4.2> Diseñe un circuito con entrada de datos de 1 bit y salida de datos de 1 bit que realice esta operación en serie empezando por el bit menos significativo. En una implementación serie se procesa el operando de entrada bit a bit y se generan los bits de salida de uno en uno. Por ejemplo, un circuito AND serie es sencillamente una puerta AND en la que en el ciclo N recibe como entrada el bit N de cada uno de los operandos y genera el bit N del resultado. Además de los datos de entrada, se dispone de una señal de reloj y una entrada de "comienzo" que se pone a 1 solamente al comienzo del primer ciclo. En este diseño, utilice biestables D, inversores y puertas AND, OR y XOR.

4.5.2 [20]<4.2> Repita el problema 4.5.1, pero ahora diseñe un circuito que realice la función indicada procesando de 2 en 2 bits.

Para el resto de este ejercicio, suponga que dispone de los siguientes elementos lógicos, con la latencia y el coste indicados:

	Inversor		AND		OR		XOR		Biestable D	
	Latencia	Coste	Latencia	Coste	Latencia	Coste	Latencia	Coste	Latencia	Coste
a.	20ps	1	30ps	2	20ps	2	30ps	4	40ps	6
b.	40ps	1	50ps	2	60ps	2	80ps	3	80ps	12

El tiempo dado para el biestable D es el tiempo de preestabilización. La entrada de datos debe tener un valor correcto un tiempo igual al tiempo de preestabilización antes del flanco de la señal de reloj (fin del ciclo de reloj) que almacena el valor en el biestable.

4.5.3 [10]<4.2> ¿Cuál es la duración del ciclo de reloj del circuito implementado en el problema 4.5.1? ¿Cuánto tarda en hacer una operación de 32 bits?

4.5.4 [10]<4.2> ¿Cuál es la duración del ciclo de reloj del circuito implementado en el problema 4.5.2? ¿Cuál es la aceleración que se obtiene al utilizar esta implementación en lugar de la implementación del problema 4.5.1 en una operación de 32 bits?

4.5.5 [10]<4.2> Calcule el coste de las implementación de los problemas 4.5.1. y 4.5.2.

4.5.6 [5]<4.2> Compare las relaciones coste/prestaciones de las implementación de los problemas 4.5.1. y 4.5.2. Para este problema las prestaciones se definen como el inverso del tiempo necesario para una operación de 32 bits.

Ejercicio 4.6

En los problemas de este ejercicio se supone que los bloques lógicos necesarios para implementar el camino de datos de un procesador tienen las siguientes latencias:

	I-mem	Add	Mux	ALU	Regs	D-Mem	Ext. signo	Desp-izq-2
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

4.6.1 [10]<4.3> Si lo único que se quiere que haga el procesador es capturar instrucciones consecutivas (figura 4.6), ¿cuál será el tiempo de ciclo?

4.6.2 [10]<4.3> Considere un camino de datos similar a la de la figura 4.11, pero para un procesador que tiene únicamente un tipo de instrucción: salto incondicional relativo al PC. ¿Cuál será el tiempo de ciclo para este camino de datos?

4.6.3 [10]<4.3> Repita el problema 4.6.2 pero ahora el único tipo de instrucción es el salto condicional relativo al PC.

Los restantes problemas de este ejercicio se refieren al siguiente bloque lógico (recurso) en el camino de datos:

	Recurso
a.	Suma 4 (al PC)
b.	Memoria de datos

4.6.4 [10]<4.3> ¿Qué clases de instrucciones necesitan este recurso?

4.6.5 [20]<4.3> ¿Qué clases de instrucciones tienen este recurso en el camino crítico?

4.6.6 [10]<4.3> Suponiendo que solamente se admiten instrucciones `beq` y `add`, discuta cómo afectan al tiempo de ciclo del procesador los cambios en la latencia del recurso. Suponga que las latencias de los otros recursos no cambian.

Ejercicio 4.7

En este ejercicio se examina cómo afectan las latencias de los componentes del camino de datos al tiempo de ciclo del reloj y cómo utilizan las instrucciones esos componentes. En los problemas de este ejercicio se supone que los bloques lógicos del camino de datos tienen las siguientes latencias:

	I-mem	Add	Mux	ALU	Regs	D-Mem	Ext. signo	Desp-izq-2
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	0ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

4.7.1 [10]<4.3> ¿Cuál es la duración del ciclo de reloj si el único tipo de instrucción admitidas son las instrucciones de la ALU (`add`, `and`, etc.)?

4.7.2 [10]<4.3> ¿Cuál es la duración del ciclo de reloj si sólo se admiten instrucciones de carga (`lw`)?

4.7.3 [10]<4.3> ¿Cuál es la duración del ciclo de reloj si se admiten las instrucciones `add`, `beq`, `lw` y `sw`?

Para los restantes problemas de este ejercicio se supone que no hay bloqueos en el pipeline y que el porcentaje de instrucciones ejecutadas es el siguiente:

	add	addi	not	beq	lw	sw
a.	30%	15%	5%	20%	20%	10%
b.	25%	5%	5%	15%	35%	15%

4.7.4 [10]<4.3> ¿Cuál es la fracción del total de ciclos en los que se utiliza la memoria de datos?

4.7.5 [10]<4.3> ¿Cuál es la fracción del total de ciclos en los que se necesita la entrada del circuito de extensión de signo? ¿Qué función realiza este elemento en los ciclos en los que no se necesita esta entrada?

4.7.6 [10]<4.3> Si se puede mejorar la latencia de uno de los componentes del camino de datos en un 10%, ¿qué componente debería mejorarse? ¿Qué aceleración se conseguiría con esta mejora?

Ejercicio 4.8

En la fabricación de chips de silicio, los defectos del material (por ejemplo, del silicio) y los errores del proceso de fabricación pueden hacer que haya circuitos defectuosos. Un defecto muy común es que una conexión afecte a una señal en otra

conexión próxima. Este tipo de defecto se llama fallo de interferencias (*cross-talk fault*). Un tipo de fallo de interferencias especial se produce cuando una señal queda conectada a una conexión que tiene un valor lógico constante (por ejemplo, la tensión de alimentación); en este caso tendremos un fallo con-0 (*stuck-at-0*) o con-1 (*stuck-at-1*) y la señal afectada tendrá siempre el valor lógico 0 ó 1, respectivamente.

Los problemas siguientes se refieren a la siguiente señal de la figura 4.24:

	Señal
a.	Memoria de instrucciones, instrucción accedida, bit 7
b.	Unidad de control, salida MemtoReg

4.8.1 [10]<4.3, 4.4> Suponga que el chequeo del procesador se hace llenando el PC, los registros y las memorias de datos e instrucciones con algunos valores (estos valores se pueden elegir), dejando que se ejecute una instrucción y finalmente leyendo el PC, los registros y las memorias. Entonces, los valores leídos se examinan para averiguar si se ha producido un determinado fallo. Diseñe un test (PC, memorias y registros) que pueda determinar si hay un fallo con-0 en esta señal.

4.8.2 [10]<4.3, 4.4> Repita el problema 4.8.1 para un fallo con-1. ¿Se podría utilizar un único test para los dos tipos de fallos, con-0 y con-1? En caso afirmativo, explique cómo; en caso negativo, explique por qué no.

4.8.3 [60]<4.3, 4.4> Si se sabe que el procesador tiene un fallo con-1 en esta señal, ¿se podría utilizar el procesador? Para poder utilizar el procesador, se debe ser capaz de convertir cualquier programa que se ejecute en un procesador MIPS normal en un programa que funcione correctamente en este procesador. Puede suponer que se dispone del suficiente espacio libre en las memoria de instrucciones y de datos para almacenar un programa con más instrucciones y datos adicionales. Una pista: el procesador se puede utilizar si cada instrucción "inutilizada" por este fallo se puede reemplazar por una secuencia de instrucciones no inutilizadas que tenga el mismo efecto.

Los problemas siguientes se refieren al siguiente fallo:

	Fallo
a.	Con-1 (stuck-at-1)
b.	Se pone a 0 si los bits 31 a 26 de la instrucción son todos 0, en otro caso no hay ningún fallo

4.8.4 [10]<4.3, 4.4> Repita el problema 4.8.1, pero ahora se quiere comprobar si la señal de control "MemRead" tiene este fallo.

4.8.5 [10]<4.3, 4.4> Repita el problema 4.8.1 pero ahora compruebe si la señal de control "Jump" tiene este fallo.

4.8.6 [40]<4.3, 4.4> Utilizando el test descrito en el problema 4.8.1, es posible comprobar la presencia de fallos en varias señales diferentes, pero no en todas las señales. Describa una serie de test que permitan buscar este fallo en todas las salidas de los multiplexores (cada bit de salida de cada uno de los cinco multiplexores). Intente que los test tengan una única instrucción.

Ejercicio 4.9

En este ejercicio se examina el funcionamiento de un camino de datos de ciclo único con una instrucción particular. Los problemas de este ejercicio se refieren a la siguiente instrucción MIPS:

	Instrucción
a.	`lw $1,40($6)`
b.	`Label: bne $1,$2,Label`

4.9.1 [10]<4.4> ¿Cuál es el contenido de la palabra de la instrucción?

4.9.2 [10]<4.4> ¿Cuál es el identificador de registro que aparece en la entrada "Read register 1" del banco de registros?

4.9.3 [10]<4.4> ¿Cuál es el identificador de registro que aparece en la entrada "Write register" del banco de registros? ¿Realmente se escribe en este registro?

Instrucciones diferentes necesitan que se activen diferentes señales de control en el camino de datos. Los restantes problemas de este ejercicio se refieren a las siguientes señales control de la figura 4.24:

	Señal de control 1	Señal de control 2
a.	RegDst	MemRead
b.	RegWrite	MemRead

4.9.4 [20]<4.4> ¿Cuál es el valor de estas dos señales para esta instrucción?

4.9.5 [20]<4.4> Dibuje el diagrama lógico para la parte del control del camino de datos de la figura 4.24 que implementa la primera de las dos señales. Suponga que sólo se admiten las instrucciones `lw`, `sw`, `beq`, `add` y `j` (salto incondicional).

4.9.6 [20]<4.4> Repita el problema 4.9.5, pero implementando las dos señales.

Ejercicio 4.10

En este ejercicio se examina cómo la duración del ciclo de reloj del procesador afecta al diseño de la unidad de control y viceversa. En los problemas de este ejercicio se supone que los bloques lógicos del camino de datos tienen las siguientes latencias:

	I-mem	Add	Mux	ALU	Regs	D-Mem	Ext. signo	Desp-izq-2	ALU ctrl
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	0ps	50ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

4.10.1 [10]<4.2, 4.4> Para evitar un alargamiento del camino crítico del camino de datos de la figura 4.24, ¿cuánto tiempo puede tardar la unidad de control en generar la señal MemWrite?

4.10.2 [20]<4.2, 4.4> ¿Qué señal de control de la figura 4.24 necesita menos tiempo para obtenerse, y cuánto tiempo tiene la unidad de control para obtener esta señal sin que esté en el camino crítico?

4.10.3 [20]<4.2, 4.4> ¿Qué señal de control de la figura 4.24 es la más crítica en cuanto a que es necesario obtenerla cuanto antes, y cuánto tiempo tiene la unidad de control para obtener esta señal sin que esté en el camino crítico?

En los restantes problemas de este ejercicio se supone que el tiempo que la unidad de control necesita para obtener cada una de las señales de control es el siguiente:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUop	MemWrite	ALUsrc	RegWrite
a.	720ps	730ps	600ps	400ps	700ps	200ps	710ps	200ps	800ps
b.	1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

4.10.4 [20]<4.4> ¿Cuál es la duración del ciclo de reloj del procesador?

4.10.5 [20]<4.4> Si se pudiese acelerar la generación de las señales de control, pero el coste del procesador completo aumentase $1 por cada 5ps de mejora de una señal de control individual, ¿qué señales de control y cuánto se podrían acelerar para maximizar las prestaciones? ¿Cuál sería el coste (por procesador) de esta mejora de las prestaciones?

4.10.6 [20]<4.4> Si el procesador es ya demasiado caro, en lugar de añadir coste para la aceleración de las señales de control como se ha hecho en el problema 4.10.5, se podría minimizar su coste sin hacerlo más lento. Si se pudiese emplear lógica más lenta en la implementación de las señales de control, ahorrando $1 en el coste del procesador por cada 5ps que se añaden a la latencia de una señal de control individual, ¿qué señales de control podrían hacerse más lentas y cuánto se reduciría el coste del procesador sin hacerlo más lento?

Ejercicio 4.11

En este ejercicio se examina en detalle cómo se ejecuta una instrucción en un camino de datos de ciclo único. Los problemas de este ejercicio se refieren a un ciclo de reloj en el cual el procesador captura la siguiente palabra de instrucción:

	Palabra de instrucción
a.	10001100010000110000000000010000
b.	00010000001000110000000000001100

4.11.1 [5]<4.4> ¿Cuál es la salida de la extensión de signo y el salto de la unidad "Despl. Izq. 2" (en la parte superior de la figura 4.24) para esta instrucción?

4.11.2 [10]<4.4> ¿Cuáles son los valores de las entradas de la unidad de control de la ALU para esta instrucción?

4.11.3 [10]<4.4> ¿Cuál es la nueva dirección en el PC después de ejecutar esta instrucción? Resalte el camino que determina este valor.

Los problemas siguientes de este ejercicio suponen que el contenido de la memoria de datos son todos 0 y que los registros del procesador tienen los siguientes valores al comienzo del ciclo en el que se captura la instrucción:

	$0	$1	$2	$3	$4	$5	$6	$8	$12	$31
a.	0	1	2	3	–4	5	6	8	1	–32
b.	0	–16	–2	–3	4	–10	–6	–1	8	–4

4.11.4 [10]<4.4> Muestre los valores de las salidas de cada multiplexor durante la ejecución de esta instrucción con estos valores en los registros.

4.11.5 [10]<4.4> ¿Cuáles son los valores de las entradas de la ALU y las dos unidades de suma?

4.11.6 [10]<4.4> ¿Cuáles son lo valores de todas las entradas de la unidad de "Registros"?

Ejercicio 4.12

En este ejercicio se examina cómo afecta la segmentación a la duración del ciclo de reloj del procesador. Los problemas de este ejercicio suponen que las etapas del camino de datos tienen las siguientes latencias:

	IF	ID	EX	MEM	WB
a.	300ps	400ps	350ps	500ps	100ps
b.	200ps	150ps	120ps	190ps	140ps

4.12.1 [5]<4.5> ¿Cuál es el ciclo de reloj en un procesador segmentado y en uno no segmentado?

4.12.2 [10]<4.5> ¿Cuál es la latencia de una instrucción `lw` en un procesador segmentado y en uno no segmentado?

4.12.3 [10]<4.5> Si se divide una etapa del camino de datos segmentado en dos nuevas etapas, cada una con una latencia mitad de la de la etapa original, ¿qué etapa se debería dividir y cuál sería el nuevo ciclo de reloj del procesador?

Los restantes problemas de este ejercicio suponen que el porcentaje de instrucciones ejecutadas por el procesador es el siguiente:

	ALU	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	25%	30%	15%

4.12.4 [10]<4.5> Suponiendo que no hay ni bloqueos ni riesgos, ¿cuál es la utilización (% de ciclos utilizada) de la memoria de datos?

4.12.5 [10]<4.5> Suponiendo que no hay ni bloqueos ni riesgos, ¿cuál es la utilización del puerto de escritura de la unidad "Registros"?

4.12.6 [30]<4.5> En lugar de una organización de ciclo único, se puede utilizar una organización multiciclo en la que cada instrucción necesita varios ciclos, pero una instrucción termina antes de que la siguiente sea capturada. En esta organización, una instrucción únicamente pasa por la etapas que realmente necesita (por ejemplo, ST sólo dura cuatro ciclo, porque no necesita la etapa WB). Compare el ciclo de reloj y el tiempo de ejecución de las organizaciones de ciclo único, multiciclo y segmentada.

Ejercicio 4.13

En este ejercicio se examina cómo afectan las dependencias de datos a la ejecución en el pipeline básico de 5 etapas descrito en la sección 4.5. Los problemas de este ejercicio se refieren a la siguiente secuencia de instrucciones:

	Secuencia de instrucciones

a.	`lw  $1, 40($6)`
	`add $6, $2, $2`
	`sw  $6, 50($1)`

b.	`lw  $5, -16($5)`
	`sw  $5,-16 ($5)`
	`add $5, $5, $5`

4.13.1 [10]<4.5> Indique las dependencias y su tipo.

4.13.2 [10]<4.5> Suponiendo que en este procesador segmentado no hay anticipación, indique los riesgos y añadir instrucciones `nop` para eliminarlos.

4.13.3 [10]<4.5> Suponga que hay anticipación completa. Indique los riesgos y añada instrucciones `nop` para eliminarlos. Los restantes problemas de este ejercicio suponen los siguientes ciclos de reloj:

	Sin anticipación	Con anticipación completa	Sólo con anticipación ALU-ALU
a.	300ps	400ps	360ps
b.	200ps	250ps	220ps

4.13.4 [10]<4.5> ¿Cuál es el tiempo total de ejecución de esta secuencia de instrucciones sin anticipación y con anticipación completa? ¿Cuál es la aceleración que se obtiene al añadir anticipación completa a un pipeline que no tiene anticipación?

4.13.5 [10]<4.5> Añada instrucciones `nop` a este código para eliminar los riesgos cuando solamente hay anticipación ALU-ALU (no hay anticipación desde la etapa MEM a la etapa EX).

4.13.6 [10]<4.5> ¿Cuál es el tiempo total de ejecución de esta secuencia de instrucciones con anticipación sólo ALU-ALU? ¿Cuál es la aceleración respecto a un pipeline que no tiene anticipación?

Ejercicio 4.14

En este ejercicio se examina cómo afectan los riesgos de recursos y de control y el diseño de la ISA a la ejecución segmentada. Los problemas de este ejercicio se refieren a la siguiente secuencia de instrucciones MIPS:

	Secuencia de instrucciones

a.	`lw $1,40($6)`
	`beq $2,$0 Label ; suponga $2 == $0`
	`sw $6,50($2)`
	`Label: add $2,$3, $4`
	`sw $3,50($4)`

b.	`lw $5,-16($5)`
	`sw $4,-16 ($4)`
	`lw $3,-20($4)`
	`beq $2,$0 Label ; suponga $2!=$0`
	`add $5,$1,$4`

4.14.1 [10]<4.5> Suponga para este problema que todos los saltos se predicen correctamente (lo que elimina todos los riesgos de control) y que no se utilizan los huecos del salto. Si solamente se dispone de una memoria (para datos e ins-

trucciones), se produce un riesgo estructural cada vez que se necesita capturar una instrucción en el mismo ciclo en el que otra instrucción accede a un dato en memoria. Para garantizar el avance de las instrucciones, este riesgo debe resolverse siempre a favor de la instrucción que accede al dato en memoria. ¿Cuál es el tiempo de ejecución total de esta secuencia de instrucciones en un pipeline de cinco etapas que tiene solamente una memoria? Se ha visto que los riesgos de datos pueden eliminarse con la introducción de instrucciones `nop` en el código. ¿Se puede hacer lo mismo con los riesgos estructurales? ¿Por qué?

4.14.2 [20]<4.5> Suponga para este problema que todos los saltos se predicen correctamente (lo que elimina todos los riesgos de control) y que no se utilizan los huecos del salto. Si se cambian las instrucciones de carga/almacenamiento para que usen el contenido de un registro como dirección de memoria (sin desplazamiento), no necesitan utilizar la ALU. En consecuencia, las etapas MEM y EX pueden solaparse y el pipeline tendría sólo cuatro etapas. Cambie este código para incorporar este cambio en la ISA. Suponiendo que este cambio no afecta al ciclo de reloj, ¿cuál es la aceleración que se obtiene para esta secuencia de instrucciones?

4.14.3 [10]<4.5> Suponiendo que cada salto produce un bloqueo y que no se utilizan los huecos del salto, ¿cuál es la aceleración en este código si las salidas del salto se determinan en la etapa ID con respecto a una ejecución en la que las salidas del salto se determinan en la etapa EX?

Los problemas siguientes de este ejercicio suponen que las etapas del camino de datos tienen las siguientes latencias:

	IF	ID	EX	MEM	WB
a.	100ps	120ps	90ps	130ps	60ps
b.	180ps	100ps	170ps	220ps	60ps

4.14.4 [10]<4.5> Dadas las latencias de la tabla, repita el cálculo de la aceleración del problema 4.14.2 pero teniendo en cuenta que el ciclo de la señal de reloj podría haber cambiado. Si EX y MEM se combinan en una sola etapa, la mayor parte de su trabajo puede hacerse en paralelo; en consecuencia, la latencia de la etapa EX/MEM resultante es igual a la latencia de la más lenta de las dos etapas originales más 20ps, necesarios para el trabajo que no puede hacerse en paralelo.

4.14.5 [10]<4.5> Dadas las latencias de la tabla, repita el cálculo de la aceleración del problema 4.14.3 pero teniendo en cuenta que el ciclo de la señal de reloj podría haber cambiado. Suponga que al mover la resolución de la salida del salto a la etapa ID, la latencia de la etapa ID aumenta un 50% y la de la etapa EX disminuye en 10ps.

4.14.6 [10]<4.5> Suponiendo que cada salto produce un bloqueo y que no se utilizan los huecos del salto, ¿cuál es nuevo ciclo de reloj y el tiempo de ejecución de esta secuencia de instrucciones si el cálculo de la dirección destino de la instrucción

`beq` se mueve a la etapa MEM? ¿Cuál es la aceleración que se obtiene con este cambio? Suponga que al mover la la latencia de la etapa EX se reduce en 20ps y la de la etapa MEM no cambia.

Ejercicio 4.15

En este ejercicio se examina cómo afecta el diseño de la ISA a la ejecución segmentada. Los problemas de este ejercicio se refieren a las nuevas instrucciones siguientes:

a.	`bezi (Rs),Label`	**If Mem[Rs] = 0 then PC=PC+desplz.**
b.	`swi Rd,Rs(Rt)`	**Mem[Rs+Rt]=Rd**

4.15.1 [20]<4.5> ¿Qué cambios hay que hacer en el camino de datos para añadir estas instrucciones a la ISA del MIPS?

4.15.2 [10]<4.5> ¿Qué nuevas señales de control deben añadirse la diseño del problema 4.15.1?

4.15.3 [20]<4.5, 4.13> ¿Estas modificaciones han introducido riesgos adicionales? ¿Se han empeorado los bloqueos debido a los riesgos ya existentes anteriormente?

4.15.4 [10]<4.5, 4.13> Dé un ejemplo de dónde podría ser útil esta instrucción y una secuencia de instrucciones MIPS que son reemplazadas por esta instrucción.

4.15.5 [10]<4.5, 4.11, 4.13> Si esta instrucción ya existiese en una ISA anterior, explique cómo se ejecutaría en un procesador moderno como el AMD Barcelona.

El último problema de este ejercicio supone que cada uso de la nueva instrucción reemplaza al número dado de instrucciones originales, que la sustitución puede hacerse una vez en el número de instrucciones originales dado y que cada vez que se ejecuta la nueva instrucción se añaden el número de ciclos de bloqueo dado al tiempo de ejecución del programa:

	Reemplazamientos	Una vez cada	Ciclos de bloqueo adicionales
a.	2	20	1
b.	3	60	0

4.15.6 [10]<4.5> ¿Qué aceleración se obtiene con esta nueva instrucción? Para calcularla, suponga que el CPI del programa original, sin la nueva instrucción, es 1.

Ejercicio 4.16

Los tres primeros problemas de este ejercicio se refieren a la siguiente instrucción MIPS:

	Instrucción
a.	`lw $1,40($6)`
b.	`add $5,$5,$5`

4.16.1 [5]<4.6> Al ejecutar esta instrucción, ¿qué se almacena en cada uno de los registros situados entre dos etapas del pipeline?

4.16.2 [5]<4.6> ¿Qué registros necesitan leerse y cuáles se leen realmente?

4.16.3 [5]<4.6> ¿Qué hace la instrucción en las etapas EX y MEM?

Los tres problemas restantes se refieren al siguiente lazo. Suponga que se utiliza una predicción de saltos perfecta (sin bloqueos debido a riesgos de control), que no hay huecos de salto y que el pipeline implementa una anticipación completa. Suponga también que se ejecutan muchas iteraciones del lazo antes de salir del mismo.

	Lazo

a.	`Lazo: lw $1,40($6)`
	`add $5,$5,$8`
	`add $6,$6,$8`
	`sw $1,20($5)`
	`beq $1,$0, Lazo`

b.	`Lazo: add $1,$2,$3`
	`sw $0,0($1)`
	`sw $0,4($1)`
	`add $2,$2,$4`
	`beq $2,$0, Lazo`

4.16.4 [10]<4.6> Muestre el diagrama de ejecución del pipeline para la tercera iteración del lazo, desde el ciclo de captura de la primera instrucción de la iteración hasta, pero no incluido, el ciclo en el que se captura la primera instrucción de la siguiente iteración. Muestre todas las instrucciones que están en el pipeline durante esos ciclos (no sólo las de la tercera iteración).

4.16.5 [10]<4.6> ¿Cuántas veces (expresado como un porcentaje del total de ciclos) se tiene un ciclo con las cinco etapas del pipeline haciendo un trabajo útil?

4.16.6 [10]<4.6> ¿Qué se almacena en el registro IF/ID al comienzo del ciclo en el que se captura la primera instrucción de la tercera iteración?

Ejercicio 4.17

Los problemas de este ejercicio suponen que el porcentaje de instrucciones ejecutadas por el procesador es el siguiente:

	ALU	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	15%	35%	20%

4.17.1 [5]<4.6> Suponiendo que no hay bloqueos y que el 60% de los saltos condicionales son tomados, ¿en qué porcentaje de los ciclos el sumador del salto en la etapa EX genera un valor que realmente utiliza el procesador?

4.17.2 [5]<4.6> Suponiendo que no hay bloqueos, ¿cuántas veces (expresado como un porcentaje del total de ciclos) es necesario utilizar los tres puertos del banco de registros en el mismo ciclo?

4.17.3 [5]<4.6> Suponiendo que no hay bloqueos, ¿cuántas veces (expresado como un porcentaje del total de ciclos) se utiliza la memoria de datos?

Cada etapa del pipeline de la figura 4.33 tiene una cierta latencia. Además, la segmentación introduce registros entre las etapas (véase la figura 4.35) y cada uno de estos añade una latencia adicional. Los restantes problemas de este ejercicio suponen las siguientes latencias para la lógica dentro de cada etapa del pipeline y para los registros entre dos etapas:

	IF	ID	EX	MEM	WB	Registros del pipeline
a.	100ps	120ps	90ps	130ps	60ps	10ps
b.	180ps	100ps	170ps	220ps	60ps	10ps

4.17.4 [5]<4.6> Suponiendo que no hay bloqueos, ¿qué aceleración se obtiene al segmentar un camino de datos de ciclo único?

4.17.5 [10]<4.6> Es posible convertir todas las instrucciones de carga/almacenamiento en instrucciones basadas en registros, sin desplazamiento, y hacer los accesos a memoria en paralelo con la ALU. ¿Cuál el ciclo de reloj si se hace esto en un camino de datos de ciclo único y en un camino de datos segmentado? Suponga que la latencia de la nueva etapa EX/MEM es igual a la mayor de sus latencias.

4.17.6 [10]<4.6> Para implementar el cambio del problema 4.17.5, todas las instrucciones `lw`/`sw` existentes tienen que convertirse en secuencias de dos instrucciones. Suponiendo que esto es necesario para el 50% de estas instrucciones, ¿qué aceleración se obtiene al pasar de un pipeline de cinco etapas a uno de cuatro etapas con EX y MEM operando en paralelo?

Ejercicio 4.18

Loas tres primeros problemas de este ejercicio abordan la ejecución de las siguientes instrucciones en el camino de datos segmentado de la figura 4.51, y supone el siguiente ciclo de reloj, latencia de la ALU y latencia de los multiplexores:

	Instrucción	Ciclo de reloj	Latencia de la ALU	Latencia del mux
a.	`add $1,$2,$3`	100ps	80ps	10ps
b.	`slt $2,$1,$3`	80ps	50ps	20ps

4.18.1 [10]<4.6> ¿Cuáles son los valores de las señales de control activas en cada etapa del pipeline?

4.18.2 [10]<4.6, 4.7> ¿De cuánto tiempo dispone la unidad de control para generar la señal de control ALUSrc? Compare con una organización de ciclo único.

4.18.3 [10]<4.6> ¿Qué valor tiene la señal PCSrc para esta instrucción? Esta señal se genera al principio de la etapa MEM (sólo una puerta AND). ¿Por qué motivo podría interesar generar esta señal en la etapa EX? ¿Cuál es la razón para no hacerlo en la etapa EX?

Los siguientes problemas de este ejercicio se refieren a las siguientes señales de la figura 4.48:

	Señal 1	Señal 2
a.	RegDst	RegWrite
b.	MemRead	RegWrite

4.18.4 [5]<4.6> Identifique la etapa del pipeline en la que se generan estas dos señales y la etapa en la se utilizan.

4.18.5 [5]<4.6> ¿Para qué instrucciones MIPS estás señales toman el valor 1?

4.18.6 [10]<4.6> ¿Cuál de estas dos señales retrocede en el pipeline? ¿Constituye esto una paradoja de viaje en el tiempo? Explique la respuesta.

Ejercicio 4.19

Este ejercicio le ayudará a entender la relación coste/complejidad/prestaciones de la anticipación en un procesador segmentado. Los problemas de este ejercicio hacen uso del camino de datos segmentado de la figura 4.45. En estos problemas se supone que, de todas las instrucciones que se ejecutan en un procesador, los

siguientes porcentajes tienen un tipo particular de dependencia de datos RAW. Este tipo de dependencia particular se identifica por la etapa que produce el resultado (EX o MEM) y la etapa que lo utiliza (primera instrucción que sigue a la que produce el resultado, segunda instrucción que sigue a la que produce el resultado, o ambas). Se supone que la escritura en registro se hace en la primera mitad del ciclo de reloj, y la lectura en la segunda mitad, por lo tanto, no se tiene en cuenta las dependencias "Etapa EX a tercera instrucción" ni "Etapa MEM a segunda instrucción" porque no causan riesgos de datos. Suponga también que el CPI es 1 y que no hay riesgos de datos.

	EX a 1ª instr.	EX a 1ª y 2ª instrs.	EX a 2ª instr.	MEM a 1ª instr.
a.	10%	10%	5%	25%
b.	15%	5%	10%	20%

4.19.1 [10]<4.7> Si no se utiliza anticipación, ¿qué porcentaje de los ciclos están bloqueados debido a riesgos de datos?

4.19.2 [5]<4.7> Si se utiliza anticipación completa (anticipación de todos los resultados que pueden ser anticipados), ¿qué porcentaje de los ciclos están bloqueados debido a riesgos de datos?

4.19.3 [10]<4.7> Suponga que no es posible tener los multiplexores de tres entradas que son necesarios para la anticipación. Hay que decidir si es mejor anticipar sólo desde el registro de segmentación EX/MEM (anticipación al siguiente ciclo) o sólo desde el registro de segmentación MEM/WB (anticipación de dos ciclos). ¿Cuál de estas dos opciones tendrá menos bloqueos, expresado en número de ciclos?

Los tres problemas restantes de este ejercicio hacen uso de las siguientes latencias de la etapas del pipeline. Para la etapa EX, las latencias de dan separadas para un procesador sin anticipación y para un procesador con diferentes tipos de anticipación.

	IF	ID	EX (sin anticip.)	EX (anticip. completa)	EX (Anticip. sólo desde EX/MEM)	EX (Anticip. sólo desde EX/WB	MEM	WB
a.	100ps	50ps	75ps	110ps	100ps	100ps	100ps	60ps
b.	250ps	300ps	200ps	350ps	320ps	310ps	300ps	200ps

4.19.4 [10]<4.7> Para los porcentajes de riesgos y las latencias de las etapas del pipeline dadas, ¿qué aceleración se obtiene al incorporar anticipación completa a un pipeline sin anticipación?

4.19.5 [10]<4.7> ¿Qué aceleración (relativa a un procesador sin anticipación) adicional se obtiene si se incorpora una anticipación que elimine todos los riesgos

de datos? Suponga que este circuito, todavía no inventado, añade una latencia de 100ps a la latencia de la etapa EX de un procesador con anticipación completa.

4.19.6 [20]<4.7> Repita el problema 4.19.3, pero en esta ocasión determine cuál de las dos opciones tiene un tiempo más corto por instrucción.

Ejercicio 4.20

Los problemas de este ejercicio utilizan la siguiente secuencia de instrucciones:

	Secuencia de instrucciones

a.	`lw $1,40($2)`
	`add $2,$3,$3`
	`add $1,$1,$2`
	`sw $1,20($2)`

b.	`add $1,$2,$3`
	`sw $2,0($1)`
	`lw $1,4($2)`
	`add $2,$2,$1`

4.20.1 [5]<4.7> Encuentre todas las dependencias de datos en esta secuencia de instrucciones.

4.20.2 [10]<4.7> Encuentre todos los riesgos en esta secuencia de instrucciones para un pipeline de cinco etapas con y sin anticipación.

4.20.3 [10]<4.7> Para reducir el ciclo del reloj, se considera la posibilidad de dividir la etapa MEM en dos etapas. Repita el problema 4.20.2 para este pipeline de seis etapas.

En los tres problemas restantes de este ejercicio se supone que, antes de ejecutar la secuencia de instrucciones anterior, todos los valores en la memoria de datos valen 0 y los valores en los registros 0 a 3 son los siguientes:

	$0	$1	$2	$3
a.	0	1	31	1000
b.	0	-2	63	2500

4.20.4 [5]<4.7> ¿Cuál es el primer valor que se anticipa y a qué valor reemplaza?

4.20.5 [10]<4.7> Si se supone que la anticipación se implementa cuando se diseña la unidad de detección de riesgos, pero realmente hemos olvidado implementar el adelantamiento, ¿qué valores tendrán los registros al final de esta secuencia de instrucciones?

4.20.6 [10]<4.7> Para el diseño del problema 4.20.5, añada `nop` a la secuencia de instrucciones para asegurar una ejecución correcta a pesar de no disponer de anticipación.

Ejercicio 4.21

Este ejercicio le ayudará a entender la relación entre la anticipación, la detección de riesgos y el diseño de la ISA. Los problemas de este ejercicio hacen uso del camino de la siguiente secuencia de instrucciones, que se ejecutan en un camino de datos segmentado:

	Secuencia de instrucciones

a.	`lw $1, 40($6)`
	`add $2, $3, $1`
	`add $1, $6, $4`
	`sw $2, 20($4)`
	`and $1, $1, $4`

b.	`add $1, $5, $3`
	`sw $1,0 ($2)`
	`lw $1, 4($2)`
	`add $5, $5, $1`
	`sw $1,0 ($2)`

4.21.1 [5]<4.7> Considerando que no hay anticipación ni detección de riesgos, inserte `nop` para asegurar una ejecución correcta.

4.21.2 [10]<4.7> Repita el problema 4.21.1, pero usando `nop` sólo cuando un riesgo no puede evitarse cambiando o reorganizando las instrucciones. Suponga que el registro R7 puede usarse en el código modificado para almacenar valores temporalmente.

4.21.3 [10]<4.7> Si el procesador tiene anticipación pero se ha olvidado de implementar la unidad de detección de riesgos, ¿qué ocurre al ejecutar esta secuencia de instrucciones?

4.21.4 [20]<4.7> Si hay anticipación, especifique qué señales activan las unidades de detección de riesgos y anticipación de la figura 4.60 en cada ciclo, durante los cinco primeros ciclos de la ejecución de la secuencia de instrucciones.

4.21.5 [10]<4.7> Si no hay anticipación, ¿qué nuevas señales de entrada y salida son necesarias en la unidad de detección de riesgos de la figura 4.60? Usando esta secuencia de instrucciones como ejemplo, explique por qué es necesaria cada una de estas señales.

4.21.6 [20]<4.7> Especifique qué señales se activan en cada uno de los cinco primeros ciclos de ejecución de la secuencia de instrucciones en la nueva unidad de detección de riesgos del problema 4.21.5.

Ejercicio 4.22

Este ejercicio le ayudará a entender la relación entre los huecos de salto, los riesgos control y la ejecución de saltos en un procesador segmentado. En este ejercicio se supone que el siguiente código MIPS se ejecuta en un procesador segmentado de cinco etapas, con anticipación completa y predicción de salto tomado:

a.	`Label1: lw $1,40($6)`
	`beq $2,$3,Label2 ; Tomado`
	`add $1,$6, $4`
	`Label2: beq $1,$2,Label1 ; No tomado`
	`sw $2,20($4)`
	`and $1,$1, $4`

b.	`add $1,$5,$3`
	`Label1: sw $1,0 ($2)`
	`add $2,$2,$3`
	`beq $2,$4,Label1 ; No tomado`
	`add $5,$5,$1`
	`sw $1,0($2)`

4.22.1 [10]<4.8> Dibuje el diagrama de ejecución del pipeline para este código, suponiendo que no se utilizan los huecos de saltos y que los saltos se ejecutan en la etapa EX.

4.22.2 [10]<4.8> Repita el problema 4.22.1, pero suponiendo que se utilizan los huecos de saltos. En el código de la tabla, la instrucción que está después del salto ocupará el hueco de salto retardado.

4.22.3 [20]<4.8> Una manera de mover la resolución del salto a una etapa anterior es no hacer uso de una operación de la ALU en las instrucciones de salto condicional. Las instrucciones de salto condicional podrían ser "`bez Rd, Label`" y "`bnez Rd, Label`", y se tomaría el salto si ese registro tiene el valor 0 o un valor distinto de 0, respectivamente. Cambie el código para emplear estas instrucciones de salto en lugar de la instrucción `beq`. Asuma que se puede utilizar el registro `$8` como registro temporal, y que se puede utilizar la instrucción de tipo R `seq` (poner a `1` si igual).

La sección 4.8 describe cómo se puede reducir el impacto de los riesgos de control moviendo la ejecución del salto a la etapa ID. Esta alternativa implica que se debe disponer de un comparador en la etapa ID dedicado a las instrucciones de salto, tal como se muestra en la figura 4.62. Sin embargo, es posible que aumente la latencia de la etapa ID y se requiere lógica de anticipación y detección de riesgos adicional.

4.22.4 [10]<4.8> Utilizando la primera instrucción de salto del código de la tabla como ejemplo, describa la lógica de detección de riesgos necesaria para ejecutar el salto en la etapa ID, tal como se indica en la figura 4.62. ¿Qué tipo de riesgos detectará esta nueva lógica?

4.22.5 [10]<4.8> Para el código dado en la tabla, ¿qué aceleración se obtiene al mover la ejecución de los saltos a la etapa ID? Explique la respuesta. Para el cálculo de la aceleración, suponga que la comparación adicional en la etapa ID no afecta al ciclo del reloj.

4.22.6 [10]<4.8> Utilizando la primera instrucción de salto del código de la tabla como ejemplo, describa la lógica de anticipación que debe añadirse para ejecutar el salto en la etapa ID. Compare la complejidad de esta nueva etapa de anticipación con la de la unidad de anticipación de la figura 4.62.

Ejercicio 4.23

La importancia de disponer de un buen predictor de saltos depende lo frecuentes que sean las instrucciones de salto. Esto, junto con la precisión del predictor, determinará cuánto tiempo se bloquea el procesador debido a una predicción fallida. En este ejercicio se supondrá que el porcentaje de instrucciones dinámicas de varias categorías es el siguiente:

	Tipo R	beq	jmp	lw	sw
a.	50%	15%	10%	15%	10%
b.	30%	10%	5%	35%	20%

Suponga además las siguientes precisiones en el predictor de saltos

	Siempre tomado	Siempre no tomado	2 bits
a.	40%	60%	80%
b.	60%	40%	95%

4.23.1 [10]<4.8> Los ciclos de bloqueo debidos a fallos en la predicción de los saltos aumentan el CPI. ¿Cuál es el aumento de CPI debido a fallos del predictor "siempre tomado"? Suponga que las salida del salto se determina en la etapa EX, que no hay dependencias de datos y que no se utilizan los huecos del salto.

4.23.2 [10]<4.8> Repita el problema 4.23.1 para el predictor "siempre no tomado".

4.23.3 [10]<4.8> Repita el problema 4.23.1 para el predictor de 2 bits.

4.23.4 [10]<4.8> Con el predictor de 2 bits, ¿qué aceleración se obtendría si se pudiese convertir la mitad de las instrucciones de salto de forma que se reemplaza una instrucción de salto por una instrucción de la ALU? Suponga que la probabilidad de reemplazar una instrucción correctamente o incorrectamente predicha es la misma.

4.23.5 [10]<4.8> Con el predictor de 2 bits, ¿qué aceleración se obtendría si se pudiese convertir la mitad de las instrucciones de salto de forma que se reemplaza una instrucción de salto por dos instrucciones de la ALU? Suponga que la probabilidad de reemplazar una instrucción correctamente o incorrectamente predicha es la misma.

4.23.6 [10]<4.8> Algunas instrucciones de salto son mucho más fáciles de predecir que otras. Sabiendo que el 80% de las instrucciones de salto ejecutadas son saltos de final de lazo fáciles de predecir, que siempre se predicen correctamente, ¿cuál es la precisión del predictor de 2 bits con el 20% restante de instrucciones de salto?

Ejercicio 4.24

Este ejercicio examina la precisión de varios predictores de saltos para el siguiente patrón repetitivo de salidas de saltos.

	Salida de los saltos
a.	T, T, NT, T
b.	T, T, T, NT, NT

4.24.1 [5]<4.8> ¿Cuál es la precisión de los predictores "siempre tomado" y siempre no tomado" para esta secuencia de salidas de saltos?

4.24.2 [5]<4.8> ¿Cuál es la precisión del predictor de 2 bits para los primeros cuatro saltos de esta secuencia, suponiendo que el predictor arranca en el estado en la parte inferior a la izquierda de la figura 4.63 (predicción no tomado)?

4.24.3 [10]<4.8> ¿Cuál es la precisión del predictor de 2 bits si este patrón se repite continuamente?

4.24.4 [30]<4.8> Diseñe un predictor pueda alcanzar una predicción perfecta si este patrón se repite constantemente. El predictor debe ser un circuito secuencial con una única salida que indica la predicción (1 para tomado, 0 para no tomado) y con dos entradas, la señal de reloj y una señal de control que le indica que la instrucción es un salto condicional.

4.24.5 [10]<4.8> ¿Cuál es la precisión del predictor del problema 4.24.4 si el patrón que se repite constantemente es justamente el opuesto al de la tabla?

4.24.6 [20]<4.8> Repita el problema 4.24.4, pero ahora el predictor debe ser capaz de predecir correctamente este patrón y el opuesto (después de un tiempo transitorio durante el cual podría hacer predicciones equivocadas). El predictor debería tener una entrada que le indicase cuál fue la salida real del salto. Una pista: esta entrada le indica cual de los dos patrones se está produciendo.

Ejercicio 4.25

Este ejercicio explora cómo afecta el procesamiento de las excepciones al diseño de pipeline. Los primeros tres problemas utilizan las siguientes dos instrucciones:

	Instrucción 1	Instrucción 2
a.	`add $0, $1, $2`	`bne $1, $2, label`
b.	`lw $2, 40($3)`	`nand $1, $2, $3`

4.25.1 [5]<4.9> ¿Qué excepción puede producirse en cada instrucción? Para cada una de estas excepciones, especifique en qué etapa del pipeline se detectan.

4.25.2 [10]<4.9> Si hay una dirección diferente para procesar cada tipo de excepción, muestre cómo debe modificarse la organización del pipeline para ser capaz de procesar esta excepción. Suponga que estas direcciones se conocen en el momento de diseñar el procesador.

4.25.3 [10]<4.9> Si la segunda instrucción se captura justo después de la primera, describa qué ocurre en el pipeline cuando la primera instrucción produce la excepción que se indicó en el problema 4.25.1.

Muestre el diagrama de ejecución desde el ciclo en que se captura la primera instrucción hasta que se completa la primera instrucción del procesamiento de la excepción.

Los tres problemas siguientes de este ejercicio suponen que las rutina de las excepción están en las siguientes direcciones:

	Desbordamiento	Dirección de dato no válida	Instrucción no definida	Dirección de instrucción no válida	Funcionamiento erróneo del hardware
a.	0xFFFFF000	0xFFFFF100	0xFFFFF200	0xFFFFF300	0xFFFFF400
b.	0x00000008	0x000000010	0x00000018	0x00000020	0x00000028

4.25.4 [5]<4.9> ¿Cuál es la dirección de comienzo de la rutina de la excepción del problema 4.25.3? ¿Qué ocurre si hay una una instrucciones no válida en esta dirección de la memoria de instrucciones?

4.25.5 [20]<4.9> En rutinas de excepción vectorizadas, la tabla de las direcciones de las rutinas de las excepciones está en la memoria de datos en direcciones conocidas. Cambie el pipeline para implementar este mecanismo de procesamiento de instrucciones. Repita el problema 4.25.3 utilizando este pipeline modificado y el procesamiento vectorizado de las excepciones.

4.25.6 [15]<4.9> Se quiere emular el procesamiento vectorizado de las excepciones (descrito en el problema 4.25.5) en un procesador que tiene una sola dirección de entrada a las rutinas de excepción. Escriba el código que debería encontrarse en esta

dirección. Una pista: este código debe identificar la excepción, proporcionar la dirección apropiada de la tabla de vectores de excepción y transferir la ejecución a la rutina.

Ejercicio 4.26

Este ejercicio explora cómo afecta el procesamiento de las excepciones al diseño de la unidad de control y a la duración del ciclo de la señal de reloj. Los primeros tres problemas utilizan las dos instrucciones MIPS siguientes, que producen una excepción:

	Instrucción	Excepción
a.	`add $0,$1, $2`	Desbordamiento aritmético
b.	`lw $2,40($3)`	Dirección de memoria de datos inválida

4.26.1 [10]<4.9> Determine los valores de la señales de control relacionadas con la excepción, figura 4.66, en cada etapa del pipeline a medida que la instrucción pasa de una etapa a otra.

4.26.2 [5]<4.9> Algunas de las señales de control generadas en la etapa ID se almacenan en el registro de segmentación ID/EX, y otras van directamente a la etapa EX. Utilizando esta instrucción como ejemplo, explique por qué razón.

4.26.3 [10]<4.9> Se puede hacer que la etapa EX sea más rápida si las excepciones no se comprueban en la etapa en la que ocurre la condición excepcional, sino en la etapa siguiente. Utilizando esta instrucción como ejemplo, indique las desventajas de este método.

Los problemas siguientes de este ejercicio suponen que las etapas del camino de datos tienen las siguientes latencias:

	IF	ID	EX	MEM	WB
a.	300ps	320ps	350ps	350ps	100ps
b.	200ps	170ps	210ps	210ps	150ps

4.26.4 [10]<4.9> Si la excepción por desbordamiento ocurre una vez cada 100.000 instrucciones ejecutadas, ¿cuál es la aceleración que se obtiene al mover la comprobación del desbordamiento a la etapa MEM? Suponga que este cambio reduce la latencia de la etapa EX en 30ns y que el IPC del procesador es 1 cuando no hay excepciones.

4.26.5 [20]<4.9> ¿Es posible generar las señales de control de una excepción en la etapa EX en lugar de en la etapa ID? Explique cómo funcionaría o por qué no funcionaría este cambio usando la instrucción `bne $4, $5, label` y las latencias de la tabla como ejemplo.

4.26.6 [10]<4.9> Suponiendo que la latencia de cada multiplexor es 40ps, determine cuánto tarda la unidad de control en obtener las señales que permiten descartar instrucciones. ¿Cuál es la señal más crítica?

Ejercicio 4.27

Este ejercicio explora cómo interactúan el procesamiento de las excepciones con las instrucciones de salto y de carga/almacenamiento. Los problemas de este ejercicio utilizan las siguientes instrucciones de salto y los huecos de salto retardado correspondientes:

	Instrucción de salto y hueco de salto retardado
a.	`beq $1,$0,Label` `sw $6, 50($1)`
b.	`beq  $5,$0,Label` `nor  $5,$4,$3`

4.27.1 [20]<4.9> Suponga que la instrucción de salto se predice correctamente como tomada, pero que la instrucción en "label" es una instrucción no definida. Describa qué ocurre en cada etapa del pipeline y en cada ciclo desde el ciclo en el que se descodifica el salto hasta el ciclo en el que se captura la primera instrucción de la rutina de la excepción.

4.27.2 [10]<4.9> Repita el problema 4.27.1, pero suponiendo que la instrucción en el hueco de salto retardado causa una excepción de error de hardware en el etapa MEM.

4.27.3 [10]<4.9> ¿Cuál es el valor del EPC si el salto es tomado pero la instrucción en el hueco de salto retardado causa una excepción? ¿Qué ocurre una vez que finaliza la rutina de la excepción?

Los tres problemas restantes de este ejercicio se refieren a las siguientes instrucciones:

	Instrucción de almacenamiento
a.	`sw $6, 50($1)`
b.	`sw $5, 60($3)`

4.27.4 [10]<4.9> ¿Qué ocurre si el salto es tomado, la instrucción en "label" es una instrucción no válida, la primera instrucción de la rutina de la excepción es la instrucción sw de la tabla y este almacenamiento accede a una dirección de datos no válida?

4.27.5 [10]<4.9> Si se puede producir un desbordamiento en el cálculo de la dirección de una carga/almacenamiento, ¿se puede retrasar la detección del desbordamiento a etapa MEM? Explique lo que ocurre utilizando la instrucción de almacenamiento de la tabla.

4.27.6 [10]<4.9> Para depurar un programa, es útil poder detectar si un determinado valor se escribe en una dirección de memoria determinada. Se quieren añadir dos nuevos registros WADDR y WVAL, de forma que se produce una excepción cuando un valor igual a WVAL se escribe en la dirección WADDR. ¿Qué cambios hay que hacer en el pipeline? ¿Cómo se procesaría la instrucción sw en este camino de datos modificado?

Ejercicio 4.28

En este ejercicio se comparan las prestaciones de los procesadores de 1 y 2 vías, teniendo en cuenta las transformaciones que se pueden aplicar a los programas para optimizar su ejecución en un procesador de 2 vías. Para ello se utilizará el siguiente programa (escrito en C):

	Código C
a.	`for (i=0; !=j, i++)` `  b[i]=a[i];`
b.	`for (i=0; a[i]!=a[i+1], i++)` `  a[i]=0;`

Suponga que al escribir un código MIPS las variables se mantienen en registros como se muestra en la tabla, y que todos los registros excepto aquellos etiquetados como Libre se utilizan para almacenar varias variables, de modo que no pueden ser utilizados para ninguna otra función.

	i	j	a	b	c	Libre
a.	$1	$2	$3	$4	$5	$6,$7,$8
b.	$4	$5	$6	$7	$8	$1,$2,$3

4.28.1 [10]<4.10> Traduzca este código C a instrucciones MIPS. La traducción debe ser directa, sin reorganizar el código para obtener mejores prestaciones.

4.28.2 [10]<4.10> Dibuje el diagrama del pipeline para el código MIPS del problema 4.28.1 ejecutándose en el procesador de 2 vías de la figura 4.69 suponiendo que se sale del lazo después de dos iteraciones. Suponga que el procesador tiene una predicción perfecta de saltos y puede capturar dos instrucciones cualquiera (no necesariamente consecutivas) en el mismo ciclo.

4.28.3 [10]<4.10> Reorganice el código del problema 4.28.1 para mejorar las prestaciones de la ejecución en el procesador de 2 vías con planificación estática de la figura 4.69.

4.28.4 [10]<4.10> Repita el problema 4.28.2 utilizando el código MIPS del problema 4.28.3.

4.28.5 [10]<4.10> ¿Qué aceleración se obtiene al utilizar el procesador de 2 vías de la figura 4.69, respecto a un procesador de 1 vía? Utilice el código del problema 4.28.1 tanto para el procesador de 1 vía como para el de 2 vías y suponga se ejecutan 1.000.000 de iteraciones del lazo. Al igual que en el problema 4.28.2, suponga que el procesador tiene una predicción perfecta de saltos y puede capturar dos instrucciones cualquiera (no necesariamente consecutivas) en el mismo ciclo.

4.28.6 [10]<4.10> Repita el problema 4.28.5 suponiendo que en el procesador de 2 vías en un ciclo puede ejecutar, simultáneamente, una instrucción de cualquier tipo y una instrucción que no sea de acceso a memoria.

Ejercicio 4.29

En este ejercicio se analiza la ejecución de un lazo planificado de forma estática en un procesador superescalar. Para simplificar, se supondrá que en el mismo ciclo pueden ejecutarse instrucciones de cualquier tipo, por ejemplo, en un superescalar de 3 vías, las tres instrucciones pueden ser tres instrucciones de la ALU, tres instrucciones de salto, tres instrucciones de carga/almacenamiento o cualquier combinación de estos tipos de instrucciones. Observe que así se eliminan las dependencias de recursos, pero todavía hay que contemplar las dependencias de datos y de control. Se utiliza el siguiente lazo:

	Lazo

a.	`Lazo: lw $1,40($6)`
	`add $5,$5,$1`
	`sw $1,20($5)`
	`addi $6,$6,4`
	`addi $5,$5,-4`
	`beq $5,0$,Lazo`

b.	`Lazo: add $1,$2,$3`
	`sw $0,0($1)`
	`addi $2,$2,4`
	`beq $2,0$,Lazo`

4.29.1 [10]<4.10> Determine el porcentaje de las lecturas de registros que son realmente útiles en un procesador superescalar estático de 2 vías. Suponga que se ejecutan muchas iteraciones del lazo, por ejemplo, 1.000.000.

4.29.2 [10]<4.10> Determine el porcentaje de las lecturas de registros que son realmente útiles en un procesador superescalar estático de 3 vías. Suponga que se ejecutan muchas iteraciones del lazo, por ejemplo, 1.000.000. Compare con el procesador de 2 vías del problema 4.29.1

4.29.3 [10]<4.10> Determine el porcentaje de ciclos en los que se realizan escrituras en dos o tres registros en un procesador superescalar estático de 3 vías. Suponga que se ejecutan muchas iteraciones del lazo, por ejemplo, 1.000.000.

4.29.4 [20]<4.10> Desenrolle el lazo una vez y planifíquelo en un procesador superescalar estático de 2 vías. Suponga que siempre se ejecuta un número par de iteraciones. Utilice los registros $10 a $20 para cambiar el código y eliminar dependencias.

4.29.5 [20]<4.10> ¿Qué aceleración se obtiene en un procesador superescalar estático de 2 vías al utilizar el código del problema 4.29.4 en lugar del código original? Suponga que se ejecutan muchas iteraciones del lazo, por ejemplo, 1.000.000.

4.29.6 [10]<4.10> ¿Qué aceleración se obtiene en un procesador segmentado (1 vía) al utilizar el código del problema 4.29.4 en lugar del código original? Suponga que se ejecutan muchas iteraciones del lazo, por ejemplo, 1.000.000.

Ejercicio 4.30

En este ejercicio se hacen varias suposiciones. Primero, se supone que un procesador superescalar de N vías ejecuta N instrucciones de cualquier tipo en el mismo ciclo, independientemente del tipo de instrucción. Segundo, se supone que las instrucciones se escogen de forma independiente, sin considerar las instrucciones que le preceden o le siguen. Tercero, se supone que no hay bloqueos debido a dependencias de datos, que no se utilizan los huecos de salto retardado y que los saltos se ejecutan en la etapa EX. Finalmente, el porcentaje de instrucciones que se ejecutan es el siguiente:

	ALU	Instr. beq con predicción correcta	Instr. beq con predicción errónea	lw	sw
a.	50%	18%	2%	20%	10%
b.	40%	10%	5%	35%	10%

4.30.1 [5]<4.10> Determine el CPI de un procesador superescalar estático de 2 vías que ejecute este programa.

4.30.2 [10]<4.10> En un procesador superescalar estático de 2 vías cuyo predictor sólo puede predecir un salto por ciclo, ¿qué aceleración se alcanza añadiendo la capacidad de predecir dos saltos por ciclo? Suponga que se implementa la política de bloqueo en salto si el predictor no puede hacer la predicción de un salto.

4.30.3 [10]<4.10> En un procesador superescalar estático de 2 vías con un solo puerto de escritura en registro, ¿qué aceleración se alcanza al añadir un segundo puerto de escritura?

4.30.4 [5]<4.10> En un procesador superescalar estático de 2 vías con la segmentación clásica en cinco etapas, ¿qué aceleración se obtiene al implementar un predictor de saltos perfecto?

4.30.5 [5]<4.10> Repita el problema 4.30.4 para un procesador de 4 vías. ¿Qué conclusión se obtiene sobre la importancia de un buen predictor de saltos cuando se aumenta el número de vías?

4.30.6 [5]<4.10> Repita el problema 4.30.5 suponiendo que el procesador de 4 vías está segmentado en 50 etapas. Suponga que cada una de las cinco etapas originales se ha dividido en 10 etapas y que los saltos se ejecutan en la primera de las 10 nuevas etapas EX. ¿Qué conclusión se obtiene sobre la importancia de un buen predictor de saltos cuando se aumenta la profundidad del procesador?

Ejercicio 4.31

Los problemas de este ejercicio utilizan el siguiente lazo, del que se dispone el código x86 y la traducción a código MIPS. Se supondrá que se ejecutan muchas iteraciones antes de salir del lazo. Esto significa que, para determinar las prestaciones, sólo se necesita analizar el estado estático, sin considerar las iteraciones para el llenado y el vaciado del pipeline. Además se supondrá que se implementa un anticipación completa y una predicción de salto perfecta, sin huecos de salto; de modo que sólo tendremos que preocuparnos de los riesgos de de recursos y de datos. Observe que la mayoría de las instrucciones x86 de este ejercicio tiene dos operandos. El último operando (normalmente el segundo) indica tanto la fuente del primer dato como el destino del resultado. Si es necesario una segunda fuente de datos, se utiliza el otro operando de la instrucción. Por ejemplo, "sub (edx), eax" accede a la posición de memoria señalada por el registro edx, resta este valor del registro eax y deja el resultado en el registro eax.

	Instrucciones x86	Traducción a MIPS

a.	`Label: mov  -4(esp), eax`
	`add  (edx), eax`
	`mov  eax, -4(esp)`
	`add  1, ecx`
	`add  4, edx`
	`cmp  esi, ecx`
	`jl   Label`
		`Label: lw   $2, -4($sp)`
		`lw   $3, 0($4)`
		`add  $2, $2, $3`
		`sw   $2, -4($sp)`
		`addi $6, $6,1`
		`addi $4, $4,4`
		`slt  $1, $6,$5`
		`bne  $1, $0,Label`

b.	`Label: add  eax, (edx)`
	
	`mov  eax, edx`
	`add  1, eax`
	`jl   Label`
		`Label: lw   $2, 0($4)`
		`add  $2, $2, $5`
		`sw   $2, 0($4)`
		`add  $4, $5,$0`
		`addi $5, $5,1`
		`slt  $1, $5,$0`
		`bne  $1, $0,Label`

4.31.1 [20]<4.11> Determine el CPI de la versión MIPS en procesador de 1 vía con planificación estática y segmentación en cinco etapas.

4.31.2 [20]<4.11> Determine el CPI de la versión x86 en procesador de 1 vía con planificación estática y segmentación en siete etapas. Las etapas son IF, ID, ARD, MRD, EXE y WB. Las etapas IF, ID son similares las del MIPS de cinco etapas, ARD calcula la dirección de memoria que se va a leer, MRD realiza la lectura de memoria, EXE ejecuta la operación y WB escribe el resultado en un registro o en

memoria. La memoria de datos tiene puertos de lectura y escritura separados, para las etapas MRD y WB, respectivamente.

4.31.3 [20]<4.11> Determine el CPI de la versión x86 en procesador que traduce internamente las instrucciones a microoperaciones tipo MIPS y ejecuta esta microoperaciones en un procesador de 1 vía con planificación estática y segmentación en cinco etapas. Observe que el número de instrucciones para obtener el CPI es el número de instrucciones x86.

4.31.4 [20]<4.11> Determine el CPI de la versión MIPS en procesador de 1 vía con planificación dinámica. Suponga que no se implementa el renombrado de registros, de modo que sólo se pueden reorganizar las instrucciones que no tienen dependencias de datos.

4.31.5 [30]<4.10, 4.11> Haga el renombrado de la versión MIPS para eliminar el mayor número posible de dependencias de datos entre instrucciones de la misma iteración del lazo; suponga que hay muchos registros libres disponibles. Repita el problema 4.31.4 utilizando el código renombrado.

4.31.6 [20]<4.10, 4.11> Repita el problema 4.31.4 suponiendo que en la descodificación de una instrucción el procesador asigna un nombre nuevo al resultado y renombra los registros utilizados por las instrucciones siguientes para emplear los valores de registro correctos.

Ejercicio 4.32

En los problemas de este ejercicio se supone que los saltos representan los siguientes porcentajes del total de instrucciones ejecutadas y que los predictores de salto tiene la precisión que se indica. Se supone que el procesador nunca se bloquea debido a dependencias de datos o de recursos; es decir, el procesador en ausencia de riesgos de control siempre captura y ejecuta el número máximo de instrucciones por ciclo. Para las dependencias de control, el procesador utiliza predicción de saltos y captura de instrucciones en el camino predicho. Si la predicción es errónea, una vez resuelto el salto, las instrucciones capturadas se descartan y en el siguiente ciclo comienza la captura de instrucciones en el camino correcto.

	% de saltos sobre el total de instrucciones ejecutadas	Precisión del predictor de saltos
a.	20	90%
b.	20	99.5%

4.32.1 [5]<4.11> ¿Cuántas instrucciones se habrán ejecutado desde el instante en que se detecta que la predicción de un salto ha sido errónea hasta que se detecta otra predicción de salto errónea?

En los siguientes problemas de este ejercicio se supone que la profundidad del pipeline es la que se indica en la tabla y que la salida del salto se determina en la etapa siguiente (contando desde la etapa 1):

	Profundidad del pipeline	Salida del salto conocida en la etapa
a.	12	10
b.	25	18

4.32.2 [5]<4.11> ¿Cuántas instrucciones estarán "en progreso" (ya capturadas pero todavía sin finalizar) en cualquier instante en un procesador de 4 vías?

4.32.3 [5]<4.11> ¿Cuántas instrucciones del camino erróneo se buscarán en cada fallo de la predicción de saltos en un procesador de 4 vías?

4.32.4 [10]<4.11> ¿Qué aceleración se consigue al pasar de un procesador de 4 vías a uno de 8 vías? Suponga que el procesador de 4 vías y el de 8 vías sólo se diferencian en el número de instrucciones por ciclo y que en todo lo demás son idénticos (profundidad del pipeline, etapa de resolución del salto, etc.).

4.32.5 [10]<4.11> ¿Qué aceleración se consigue en un procesador de 4 vías al ejecutar los saltos una etapa antes?

4.32.6 [10]<4.11> ¿Qué aceleración se consigue en un procesador de 8 vías al ejecutar los saltos una etapa antes? Discuta las diferencias respecto al resultado del problema 4.32.5.

Ejercicio 4.33

En este ejercicio se explora cómo afecta la predicción de saltos a las prestaciones de un procesador profundamente segmentado de ejecución múltiple. Se utilizarán el número de etapas y el número de instrucciones ejecutadas por ciclo indicados en la tabla:

	Profundidad del pipeline	Número de instrucciones ejecutadas por ciclo
a.	10	4
b.	25	2

4.33.1 [10]<4.11> ¿Cuántos puertos de lectura de registros debería tener el procesador para evitar las dependencias de recursos en la lectura de registros?

4.33.2 [10]<4.11> En ausencia de predicciones de salto erróneas y de dependencias de datos, ¿cuál es la mejora de la prestaciones respecto a un procesador de 1 vía con la segmentación clásica en cinco etapas? Suponga que la duración del ciclo de reloj disminuye proporcionalmente al número de etapas.

4.33.3 [10]<4.11> Repita el problema 4.33.2 suponiendo, ahora, que todas las instrucciones tienen una dependencia RAW con la instrucción inmediatamente posterior. Suponga que no se producen bloqueos; es decir, la anticipación permite que las instrucciones se ejecuten en ciclos consecutivos.

Para los problemas restantes utilizaremos las siguientes estadísticas de saltos, precisión del predictor y pérdida de prestaciones debido a predicciones erróneas, a no ser que se indique lo contrario:

	% de saltos sobre el total de instrucciones ejecutadas	Los saltos se ejecutan en la etapa	Precisión del predictor de saltos	Pérdida de prestaciones
a.	30%	7	95%	10%
b.	15%	8	97%	2%

4.33.4 [10]<4.11> ¿Qué porcentaje del total de ciclos se gasta en la captura de instrucciones en el camino erróneo? Ignore los datos de pérdida de prestaciones.

4.33.5 [20]<4.11> ¿Cuál debería ser la precisión del predictor si se quiere limitar los bloqueos debido a predicciones erróneas al porcentaje dado del tiempo de ejecución ideal (sin bloqueos)? Ignore los datos de precisión de la predicción.

4.33.6 [20]<4.11> ¿Cuál sería la precisión de la predicción de saltos si se está dispuesto a tener una aceleración de 0.5 (mitad) respecto al mismo procesador con predicción de altos ideal?

Ejercicio 4.34

Este ejercicio le ayudará a entender la discusión de la falacia "la segmentación es fácil" de la sección 4.13. Se utilizarán las siguientes instrucciones MIPS en los tres primeros problemas:

	Instrucción	Interpretación
a.	`add Rd,Rs,Rt`	`Reg[Rd]=Reg[Rs]+Reg[Rt]`
b.	`lw Rt,desplaz(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+desplaz]`

4.34.1 [10]<4.13> Describa el camino de datos segmentado necesario para la ejecución de únicamente esta instrucción. El diseño debe hacerse suponiendo que éste es el único tipo de instrucción que se va a ejecutar.

4.34.2 [10]<4.13> Describa las necesidades de anticipación y detección de riesgos del camino de datos del problema 4.34.1.

4.34.3 [10]<4.13> ¿Qué hay que modificar o añadir en el camino de datos del problema 4.34.1 para aceptar excepciones debido a instrucciones no definidas? La excepción debe producirse cuando el procesador encuentra cualquier otro tipo de instrucción.

Los dos problemas siguientes utilizan la instrucción MIPS:

	Instrucción	Interpretación
a.	`beq Rs, Rt, label`	`If Reg[Rs] == Reg[Rt]  PC=PC+desplaz`
b.	`and Rd, Rs, Rt`	`Reg[Rd]=Reg[Rs]&Reg[Rt]`

4.34.4 [10]<4.13> Describa cómo hay que extender el camino de datos del problema 4.34.1 para aceptar esta instrucción. El diseño debe hacerse suponiendo que éstos son los únicos tipos de instrucciones que se van a ejecutar.

4.34.5 [10]<4.13> Repita el problema 4.34.2 para el camino de datos extendido del problema 4.34.4.

Ejercicios 4.35

Este ejercicio le ayudará a entender el diseño de la ISA y la segmentación. En los problemas de este ejercicio se supone que se dispone de un procesador segmentado con ejecución múltiple de instrucciones, con el número de etapas, instrucciones ejecutadas por ciclo, resolución del salto y precisión del predictor indicados en la tabla:

	Profundidad del pipeline	Instrs. por ciclo	Saltos ejecutados en etapa	Precisión del predictor de saltos	% de instrucciones de salto
a.	10	4	7	80%	20%
b.	25	2	17	92%	25%

4.35.1 [5]<4.8, 4.13> Los riesgos de control pueden eliminarse añadiendo huecos de salto retardado. ¿Cuántos huecos se deberán añadir después de cada salto para eliminar todos los riesgos de control en este procesadores?

4.35.2 [10]<4.8, 4.13> ¿ Qué aceleración se obtendría con cuatro huecos de salto retardado? Suponga que no hay dependencias de datos entre instrucciones y que los cuatro huecos de salto retardado se pueden ocupar con instrucciones útiles sin aumentar el número de instrucciones ejecutadas. Para facilitar los cálculos, suponga que la instrucción de salto erróneamente predicha es siempre la última instrucción que se captura en un ciclo; es decir, ninguna de las instrucciones que se encuentran en la misma etapa que el salto pertenece al camino erróneo.

4.35.3 [10]<4.8, 4.13> Repita el problema 4.35.2 suponiendo que el 10% de los saltos ejecutados han llenado los huecos de salto retardado con instrucciones útiles, el 20% tiene sólo tres instrucciones útiles en los huecos de salto (la cuarta es un `nop`), el 30% sólo dos instrucciones útiles y el 40% no tiene ninguna instrucción útil es los huecos de salto.

Los tres problemas restantes utilizan el siguiente lazo escrito en C:

a.	`for (i=0; i!=j;i++){`
	`  b[i]=a[i];`
	`}`

b.	`for (i=0; a[i]!=a[i+1];i++){`
	`  c++;`
	`}`

4.35.4 [10]<4.8, 4.13> Traduzca este código C a instrucciones MIPS suponiendo que nuestra ISA tiene un sólo huecos de salto retardado para cada salto. Intente llenar los huecos de salto retardado con instrucciones que no son `nop` cuando sea posible. Suponga que las variables `a`, `b`, `c`, `i`, `j` están en los registros `$1`, `$2`, `$3`, `$4` y `$5` respectivamente.

4.35.5 [10]<4.7, 4.13> Repita el problema 4.35.4 para un procesador con dos huecos de salto retardado en cada salto.

4.35.6 [10]<4.10, 4.13> ¿Cuántas iteraciones del lazo del problema 4.35.4 pueden estar en ejecución en el pipeline del procesador? Una iteración está en ejecución cuando al menos una de sus instrucciones y ha sido capturada y todavía no ha finalizado.

Ejercicio 4.36

Este ejercicio le ayudará a entender el último "error habitual" de la sección 4.13, errores al considerar la segmentación en el diseño del repertorio de instrucciones. Los primeros cuatro ejercicios se refieren a las siguientes nuevas instrucciones MIPS:

	Instrucción	Interpretación
a.	`lwinc Rt,desplaz(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+desplaz]` `Reg[Rs]=Reg[Rs]+4`
b.	`addr Rt,desplaz(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+desplaz]+Reg[Rt]`

4.36.1 [10]<4.11, 4.13> Traduzca este código a microoperaciones MIPS.

4.36.2 [10]<4.11, 4.13> ¿Qué cambios hay que hacer en el pipeline de cinco etapas del MIPS para poder añadir el soporte hardware necesario para las microoperaciones resultantes?

4.36.3 [20]<4.13> Si se quiere añadir esta instrucción a la ISA del MIPS, discuta los cambios que hay que introducir en el pipeline (en qué etapas, qué estructuras en cada etapa) para implementar directamente, sin microoperaciones, esta instrucción.

4.36.4 [10]<4.13> ¿Se utilizará muy a menudo esta instrucción? ¿Estaría justificado añadir esta instrucción a la ISA?

Los dos problemas restantes de este ejercicio abordan la incorporación de una nueva instrucción, `addm`, a la ISA. En el procesador en que se ha añadido esta instrucción se supondrán los siguientes porcentajes de ciclos de reloj según la instrucción que se completa en ese ciclo (o el bloqueo que está evitando que una instrucción finalice):

	add	beq	lw	sw	addm	Bloqueos de control	Bloqueos de datos
a.	35%	20%	20%	10%	5%	5%	5%
b.	25%	10%	25%	10%	10%	10%	10%

4.36.5 [10]<4.13> ¿Qué aceleración se obtiene reemplazando esta instrucción por una secuencia de tres instrucciones (`lw`, `add` y `sw`)? Suponga que la instrucción `addm` se ha incorporado (de forma mágica) al pipeline clásico de cinco etapas sin añadir recursos hardware.

4.36.6 [10]<4.13> Repita el problema 4.36.5 suponiendo que para incorporar `addm` se ha añadido una etapa adicional al pipeline. Al traducir `addm`, esta etapa adicional puede eliminarse y, como resultado, la mitad de los bloqueos de datos se eliminan. Observe que la eliminación de los bloqueos de datos se aplica sólo a los bloqueos que ya existían antes de la traducción de `addm`, no a los que aparecieron debido a la traducción de la instrucción `addm`.

Ejercicio 4.37

En este ejercicios se analizan algunas de las decisiones de diseño de la segmentación, tales como la duración del ciclo de reloj y la utilización de algunos recursos hardware. Los primeros tres problemas se refieren al siguiente código MIPS. El código se ha escrito suponiendo que el procesador no utiliza los huecos de salto retardado.

a.

```
        lw  $1,40($6)
        beq $1,$0, Label ; suponga  $1 == $0
        sw  $6,50($1)
Label:  add $2,$3, $1
        sw  $2,50($1)
```

b.

```
        lw  $5,-16($5)
        sw  $5,-16($5)
        lw  $5,-20($5)
        beq $5,$0,Label ; suponga  $5 != $0
        add $5,$5,$5
```

4.37.1 [5]<4.3, 4.14> ¿Qué partes del camino de datos básico de ciclo único se usan para todas estas instrucciones? ¿Qué partes son las menos utilizadas?

4.37.2 [10]<4.6, 4.14> ¿Cuál es la utilización del puerto de lectura y del puerto de escritura de la memoria de datos?

4.37.3 [10]<4.6, 4.14> Suponga que ya se dispone del diseño de ciclo único. ¿Cuántos bits se necesitan en total en los registros de segmentación para implementar el diseño segmentado?

En los tres problemas restantes de este ejercicio se suponen las siguientes latencias para los componentes del camino de datos:

	I-mem	Add	Mux	ALU	Regs	D-Mem	Ext. signo	Desp-izq-2
a.	**400ps**	100ps	30ps	120ps	200ps	350ps	20ps	0ps
b.	**500ps**	**150ps**	**100ps**	**180ps**	**220ps**	**1000ps**	**90ps**	20ps

4.37.4 [10]<4.3, 4.5, 4.14> Compare el ciclo de reloj del camino de datos de ciclo único y el camino de datos segmentado en cinco etapas.

4.37.5 [10]<4.3, 4.5, 4.14> Repita el problema 4.37.4, pero suponiendo que sólo se implementan instrucciones ADD.

4.37.6 [20]<4.3, 4.5, 4.14> Suponiendo que reducir en 1 ps la latencia de un componente individual del camino de datos cuesta $1, ¿cuánto costaría reducir el ciclo de reloj del camino de datos de ciclo único y del segmentado en un 20%?

Ejercicio 4.38

En este ejercicio se analiza la eficiencia energética y su relación con las prestaciones. Se supone el siguiente consumo de energía por actividad en la memoria de instrucciones, en los registros y en la memoria de datos. Además, se supone que el consumo en otros componentes del camino de datos es despreciable:

	I-Mem	Una lectura de un registro	Escritura de un registro	Lectura D-Mem	Escritura D-Mem
a.	**100pJ**	60pJ	70pJ	120pJ	100pJ
b.	**200pJ**	**90pJ**	**80pJ**	**300pJ**	**280pJ**

4.38.1 [10]<4.3, 4.6, 4.14> ¿Cuánta energía se consume en la ejecución de una instrucción `add` en el diseño de ciclo único y en el diseño segmentado en cinco etapas?

4.38.2 [10]<4.6, 4.14> ¿Qué instrucción MIPS representa el peor-caso en términos de consumo de energía y cuánta energía se consume en su ejecución?

4.38.3 [10]<4.6, 4.14> Si el consumo de energía es primordial, ¿en qué se podría cambiar el diseño segmentado? ¿En qué porcentaje se reduce el consumo de energía de la ejecución de una instrucción `lw` después de este cambio?

En los siguientes tres problemas de este ejercicio supondremos que las latencias de los componentes del camino de datos son las indicadas en la tabla. Además, supondremos que las latencias de los restantes componentes del camino de datos son despreciables:

	I-Mem	Control	Lectura o escritura de un registro	ALU	Lectura o escritura de la D-mem
a.	400ps	300ps	200ps	120ps	350ps
b.	500ps	400ps	220ps	180ps	1000ps

4.38.4 [10]<4.6, 4.14> ¿Qué impacto tienen los cambios del problema 4.38.3 en las prestaciones?

4.38.5 [10]<4.6, 4.14> Es posible eliminar la señal de control MemRead haciendo que la memoria de datos se lea en todos los ciclos; es decir, se puede tener MemRead=1 permanentemente. Explique por qué después de este cambio el procesador todavía funcionaría correctamente. ¿Qué efecto tiene este cambio en la frecuencia del reloj y en el consumo de energía?

4.38.6 [10]<4.6, 4.14> Considerando que una unidad inactiva consume el 10% de la energía que consume cuando está activa, ¿cuánta energía consume la memoria de instrucciones en cada ciclo? ¿Qué porcentaje de la energía total consumida en la memoria de instrucciones representa la energía consumida cuando está inactiva?

Ejercicio 4.39

En los problemas de este ejercicio se supondrá que, durante la ejecución de un programa, la utilización de los ciclos de un procesador se distribuyen de la forma que se indica en la siguiente tabla. Se "gasta" un ciclo en una instrucción si el procesador finaliza este tipo de instrucción en ese ciclo; se "gasta" un ciclo en un bloqueo si el procesador no puede finalizar una instrucción en ese ciclo debido a los bloqueos.

	add	beq	lw	sw	Bloqueos de control	Bloqueos de datos
a.	35%	20%	20%	10%	10%	5%
b.	25%	10%	25%	10%	20%	10%

Se supondrá también que la latencia y el consumo de energía de las etapas del pipeline son las de la tabla. La energía de cada etapa se consume para hacer su función con la latencia indicada. Observe que la etapa MEM no consume energía en un ciclo en el que no hay accesos a memoria. De forma similar, no se consume energía en la WB en aquellos ciclos en los que no se realiza la escritura de un registro. En varios de los problemas siguientes se hacen suposiciones sobre cómo cambia el consumo de energía si una etapa hace su trabajo más lento o más rápido que con la latencia especificada.

	IF	ID	EX	MEM	WB
a.	300ps/120pJ	400ps/60pJ	350ps/75pJ	500ps/130pJ	100ps/20pJ
b.	200ps/150pJ	150ps/60pJ	120ps/50pJ	190ps/150pJ	140ps/20pJ

4.39.1 [10]<4.14> Determine las prestaciones (en instrucciones por segundo).

4.39.2 [10]<4.14> ¿Cuál es la potencia disipada en vatios (julios por segundo)?

4.39.3 [10]<4.6, 4.14> ¿Qué etapas pueden hacerse más lentas, y cuánto más, sin afectar al ciclo de la señal de reloj?

4.39.4 [20]<4.6, 4.14> A menudo es posible sacrificar velocidad de un circuito para reducir el consumo de energía. Suponga que se puede reducir el consumo de energía en un factor X (el nuevo consumo de energía es 1/X el consumo de energía anterior) incrementando la latencia en un factor X (la nueva latencia es X veces la latencia anterior). Así, se puede ajustar la latencia de las etapas del pipeline para minimizar el consumo de energía sin sacrificar las prestaciones. Repita el problema 4.39.2 para un procesador con estos ajustes.

4.39.5 [10]<4.6, 4.14> Repita el problema 4.39.4, pero minimizando la energía consumida por instrucción y aumentando el ciclo de la señal de reloj en un 10% como máximo.

4.39.6 [10]<4.6, 4.14> Repita el problema 4.39.5 suponiendo que el consumo de energía se reduce en un factor X^2 haciendo la latencia X veces mayor. Determine el ahorro en el consumo de potencia respecto al problema 4.39.2.

Respuestas a las autoevaluaciones

§4.1, página 303: 3 de 5: Control, camino de datos, memoria. Faltan entrada y salida.
§4.2, página 307: falso. Los elementos de estado activos en las transiciones pueden hacer lecturas y escrituras simultáneas de forma no ambigua.
§4.3, página 315: I. A. II. C.
§4.4, página 330: Sí, Salto y ALUOp0 son identicas. Además, MemtoReg y RegDst son complementarias (una inversa de la otra). No es necesario un inversor, !sencillamente se puede usar la otra señal y cambiar el orden de las entradas en el multiplexor!

§4.5, página 343: 1. Bloqueo en el resultado de LW. 2. Anticipar el primer resultado de la ADD escrito en $t1. 3. No es necesario ni bloqueo ni anticipación.
§4.6, página 358: Las sentencias 2 y 4 son correctas; el resto son incorrectas.
§4.8, página 383: 1. Predicción no tomado. 2. Predicción tomado. 3. Predicción dinámica.
§4.9, página 391: La primera instrucción, porque se ejecuta lógicamente antes que las demás instrucciones.
§4.10, página 403: 1. Ambos. 2. Ambos. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Ambos. 8. Hardware. 9. Ambos.
§4.11, página 404: Los dos primeros son falsos y los dos últimos verdaderos.
§4.12,  página 6.7-3: Las sentencias 1 y 3 son las dos verdaderas.
§4.12,  página 6.7-7: Sólo la sentencia 3 es completamente exacta.

5

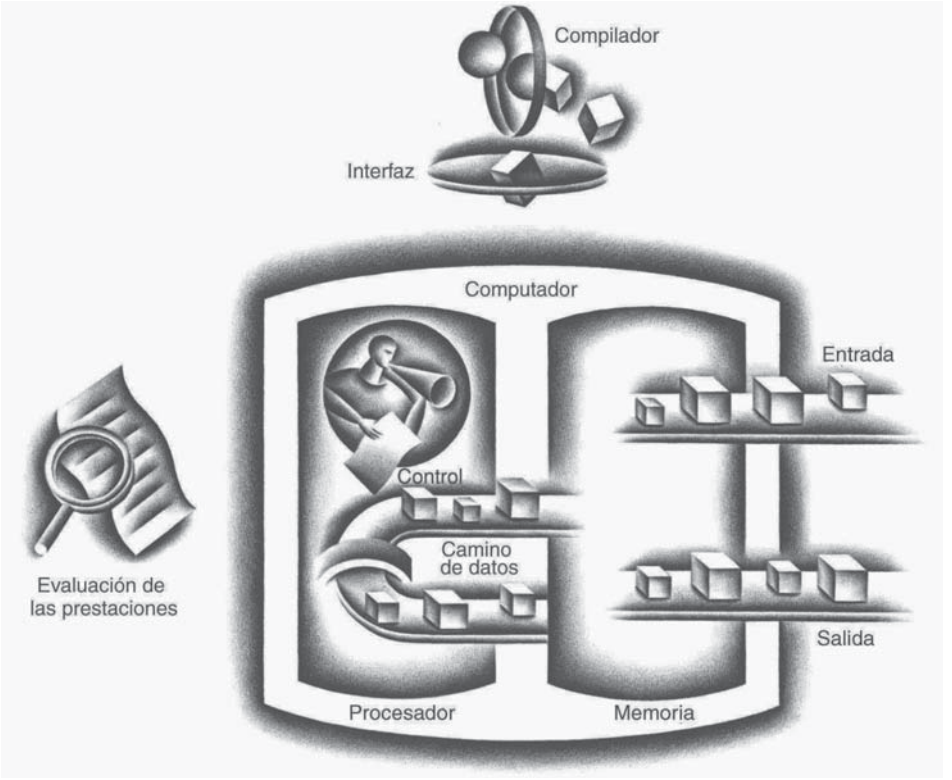
Idealmente, uno desearía una capacidad de memoria muy grande, de forma tal que cualquier… palabra estuviera disponible inmediatamente… Nos vemos…forzados a admitir la posibilidad de construir una jerarquía de niveles de memorias, cada uno de los cuales tiene mayor capacidad que el precedente, pero al que se accede menos rápidamente.

A. W. Burks, H. H. Goldstine y J. von Neumann
Discusión preliminar del diseño lógico de un instrumento de computación electrónico, 1946

Grande y rápida: aprovechamiento de la jerarquía de memoria

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco componentes clásicos del computador



5.1 Introducción

Desde los primeros días de la Informática, los programadores han deseado cantidades ilimitadas de memoria rápida. Los temas que abordaremos en este capítulo ayudan a los programadores a tener la impresión de que disponen de una memoria rápida e ilimitada. Antes de ver cómo se genera realmente esta impresión, consideremos una simple analogía que ilustra los principios y mecanismos básicos que utilizaremos.

Suponga que usted está escribiendo un informe sobre los acontecimientos históricos más importantes en la evolución del hardware del computador. Usted se encuentra sentado en la mesa de una biblioteca junto a varios libros que ha sacado de las estanterías y los está hojeando. Encuentra que varios de los computadores más importantes sobre los que es necesario escribir están en los libros que dispone, pero no encuentra nada sobre el EDSAC. Por lo tanto, usted vuelve a las estanterías a buscar otro libro. Por fin encuentra uno sobre los primeros computadores británicos que describe el EDSAC. Una vez que dispone de una buena selección de libros sobre su mesa, existe una alta probabilidad de que en ellos encuentre muchos de los temas que necesita cubrir, y por ello usted puede estar la mayor parte del tiempo utilizando los libros de su mesa sin tener que volver a las estanterías. Tener varios libros en su mesa permite ahorrar tiempo, en comparación con la situación en la que se consulta un solo libro, se vuelve a las estanterías para colocarlo en su sitio y se escoge otro libro de consulta.

El mismo principio permite crear la impresión de que se dispone de una memoria grande a la que se puede acceder tan rápidamente como a una memoria muy pequeña. De la misma forma que usted no necesitaba acceder al mismo tiempo a todos los libros de la biblioteca con la misma probabilidad, un programa no accede a la vez a todo su código o datos con la misma probabilidad. Si no fuera así, sería imposible realizar en un computador de forma rápida la mayoría de los accesos a una memoria que fuera grande, de la misma forma que sería imposible para usted poner todos los libros de la biblioteca sobre su mesa y además encontrar rápidamente lo que busca.

Localidad temporal: principio que establece que si se accede a la dirección de memoria de un dato, pronto se accederá de nuevo a esa dirección.

Localidad espacial: principio de localidad que establece que si se accede a la dirección de memoria de un dato es accedida, pronto se accederá a las direcciones de memoria que se encuentran próximas a ella.

Este *principio de localidad* subyace tanto la manera en que usted hace una búsqueda en la biblioteca como en la forma en que funcionan los programas. El principio de localidad establece que los programas acceden a una parte relativamente pequeña del espacio de direcciones en un determinado instante, de la misma forma que usted accede a una cantidad muy pequeña de libros del total de libros de la biblioteca. Existen dos tipos distintos de localidad:

- **Localidad temporal** (localidad en el tiempo): Si se accede a una dirección de memoria, esta dirección será utilizada de nuevo en un corto intervalo de tiempo. Si usted ha llevado recientemente un libro a su mesa para consultarlo, probablemente pronto necesitará consultarlo de nuevo.
- **Localidad espacial** (localidad en el espacio): Si se accede a una dirección de memoria, las direcciones de memoria que se encuentran próximas a ella serán utilizadas en un corto intervalo de tiempo. Por ejemplo, cuando usted sacó de

la estantería el libro sobre los primeros computadores ingleses para encontrar información sobre EDSAC, quizás vio cerca de él otro libro que trataba sobre los primeros computadores mecánicos; así que también se lo llevó, y posteriormente encontró en ese libro alguna información que le fue útil. Los libros que tratan un mismo tema se colocan juntos en las estanterías de las bibliotecas para incrementar la localidad espacial. Más adelante, en este capítulo, veremos cómo se utiliza la localidad espacial en las jerarquías de memoria.

Así como los accesos a los libros que están encima de la mesa tienen localidad de forma natural, la localidad en los programas surge de sus estructuras simples y naturales. Por ejemplo, la mayoría de los programa contienen lazos, por lo que probablemente se accederá repetitivamente a instrucciones y datos, mostrando una gran cantidad de localidad temporal. Puesto que el acceso a las instrucciones es secuencial, los programas muestran también una alta localidad espacial. Los accesos a datos también tienen una localidad espacial natural. Por ejemplo, los accesos a elementos de una matriz o una tupla experimentarán por naturaleza un alto grado de localidad espacial.

Nosotros aprovechamos el principio de localidad implementando la memoria de un computador como una **jerarquía de memoria**, que está formada por varios niveles de memoria con diferentes tiempos de accesos y capacidades. Las memorias más rápidas son de mayor coste económico por bit que las memorias más lentas, y por ello, son más pequeñas.

Jerarquía de memoria: estructura que utiliza varios niveles de memoria; a medida que aumenta la distancia desde la CPU, el tamaño de las memorias y el tiempo aumentan.

Hoy en día, en la construcción de jerarquías de memorias se usan principalmente tres tecnologías. La memoria principal es implementada con DRAM (memoria dinámica de acceso aleatorio), mientras que los niveles más cercanos al procesador (caches) utilizan SRAM (memoria estática de acceso aleatorio). La DRAM es menos costosa por bit que la SRAM, aunque es considerablemente más lenta. La diferencia en precio se debe a que la DRAM utiliza bastante menos área de chip por bit de memoria, y por ello la DRAM dispone de mayor capacidad de memoria para la misma cantidad de silicio; la diferencia en tiempo de acceso se debe a distintos factores que se describen en la sección C.9 del apéndice C. La tercera tecnología que se usa para implementar el mayor y más lento nivel de la jerarquía, es el disco magnético. (Las memorias Flash se utilizan en muchos dispositivos empotrados en lugar de los discos; véase sección 6.4). El tiempo de acceso y precio por bit de estas tecnologías varían extensamente, como se puede ver en la siguiente tabla, en la que se han utilizado valores típicos de 2008.

Tecnología de memoria	Tiempo de acceso típico	Dólares por GB en 2004
SRAM	0.5–2.5 ns	$2000–$5000
DRAM	50–70 ns	$20–$75
Disco magnético	5 000 000–20 000 000 ns	$0.20–$2

Debido a estas diferencias en coste económico y tiempo de acceso, es beneficioso construir memorias como una jerarquía de niveles. La figura 5.1 muestra que la memoria más rápida está cerca del procesador y la más lenta y menos cara está debajo de ella. El objetivo es presentar al usuario tanta memoria como esté disponible en la tecnología más barata, a la vez que se pueda acceder a la velocidad de la memoria más rápida.

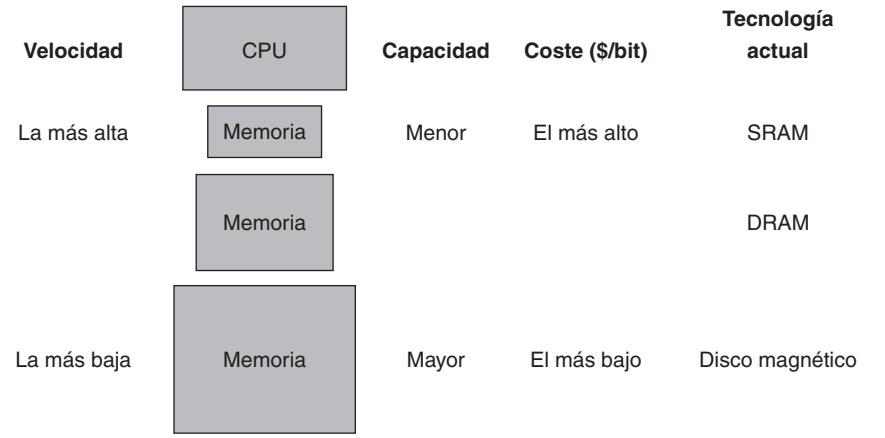


FIGURA 5.1 Estructura básica de una jerarquía de memoria. Al implementar el sistema de memoria como una jerarquía, el usuario tiene la impresión de que existe una memoria que es tan grande como la del nivel más grande de la jerarquía, pero que puede ser accedida como si estuviera construida con la memoria más rápida. Las memorias Flash han reemplazado a los discos en muchos dispositivos empotrados, y pueden llevar a un nuevo nivel en la jerarquía de almacenamiento en servidores y computadores de sobremesa; véase sección 6.4.

El sistema de memoria está organizado como una jerarquía: un nivel más cercano al procesador corresponde generalmente a un subconjunto de cualquiera de los niveles más alejados, y todos los datos están almacenados en el nivel más bajo. Siguiendo nuestra analogía, los libros con los que está trabajando y que están encima de la mesa forman un subconjunto de los libros de la biblioteca, que a su vez forman un subconjunto de todas las bibliotecas del campus. Además, a medida que nos alejamos del procesador, los niveles requieren progresivamente mayor tiempo de acceso, al igual que lo que nos podríamos encontrar en una jerarquía de bibliotecas del campus.

Bloque: unidad mínima de información que puede estar o no estar presente en una jerarquía de dos niveles.

Frecuencia de aciertos: fracción de accesos a memoria que se encontraron en un nivel determinado de la jerarquía de memoria.

Frecuencia de fallos: fracción de accesos a memoria que no se encontraron en un determinado nivel de la jerarquía de memoria.

Una jerarquía de memoria puede constar de varios niveles, pero en un instante los datos sólo se transfieren entre dos niveles adyacentes, de forma tal que podemos centrar nuestra atención en solamente dos niveles. El nivel superior —el más cercano al procesador— es más pequeño y rápido que el nivel inferior ya que utiliza la tecnología más cara. La figura 5.2 muestra que la unidad mínima de información que puede estar o no estar presente en una jerarquía de dos niveles se llama **bloque** o *línea*; en nuestra analogía de la biblioteca, un bloque de información es un libro.

Si los datos que requiere el procesador se encuentran en algún bloque del nivel superior, se dice que esto es un *acierto* (análogo a encontrar la información en uno de los libros de la mesa). Si los datos no se encuentran en el nivel superior, la petición se denomina *fallo*. En este caso, se accede al nivel inferior de la jerarquía para recuperar el bloque que contiene los datos requeridos. (Continuando con nuestra analogía, usted se dirige desde su mesa a las estanterías para encontrar el libro deseado). La **frecuencia de aciertos** o *tasa de aciertos* es la fracción de accesos a memoria cuyos datos se encuentran en el nivel superior; a menudo se usa como medida de prestaciones de la jerarquía de memoria. La **frecuencia de fallos** (1 – frecuencia de aciertos) es la fracción de los accesos a memoria no encontrados en el nivel superior.

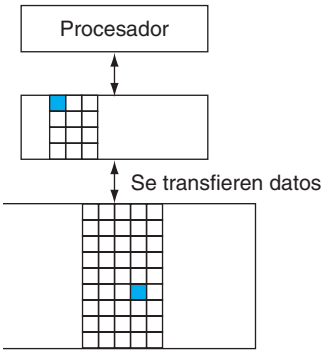


FIGURA 5.2 Se puede considerar que cada par de niveles de la jerarquía de memoria tiene un nivel superior y un nivel inferior. Dentro de cada nivel, la unidad de información que está o no presente se denomina *bloque*. Cuando algo se transmite entre niveles, normalmente se transfiere un bloque completo.

Como las prestaciones son el motivo más importante para establecer una jerarquía de memoria, es importante considerar el tiempo dedicado a aciertos y fallos. El **tiempo de acierto** es el tiempo necesario para acceder al nivel superior de la jerarquía de memoria, que incluye el tiempo requerido para determinar si el acceso corresponde a un acierto o a un fallo (esto es, el tiempo necesario para ver si un libro está encima de la mesa). La **penalización por fallo** es el tiempo necesario para reemplazar un bloque en el nivel superior por el correspondiente bloque del nivel inferior, al que hay que añadir el tiempo que se tarda en suministrar este bloque al procesador (o, el tiempo para escoger otro libro de las estanterías y depositarlo sobre la mesa). Como el nivel superior es más pequeño y está construido con componentes de memoria más rápidos, el tiempo de acierto será mucho menor que el tiempo de acceso al siguiente nivel de la jerarquía, el cual es la componente temporal más importante de la penalización por fallo. (Se tarda mucho menos en inspeccionar los libros de la mesa que en levantarse de la silla y escoger un nuevo libro de las estanterías).

Tiempo de acierto: tiempo necesario para acceder a un determinado nivel de la jerarquía de memoria, incluido el tiempo necesario para determinar si el acceso corresponde a un acierto o a un fallo.

Penalización por fallo: tiempo necesario para ir a buscar un bloque a un determinado nivel de la jerarquía de memoria partiendo desde otro nivel inferior, incluido el tiempo necesario para acceder al bloque, transmitirlo de un nivel al otro y guardarlo en el nivel que experimentó el fallo.

Como veremos en este capítulo, los conceptos utilizados para construir sistemas de memoria afectan a muchos otros aspectos de un computador, entre ellos la gestión de la memoria y la E/S por parte del sistema operativo, la generación de código por parte de los compiladores, e incluso la forma en que las aplicaciones utilizan los recursos del computador. Por supuesto, como todos los programas dedican mucho tiempo a los accesos a memoria, el sistema de memoria es necesariamente un factor de gran importancia en la determinación de las prestaciones. La dependencia de las jerarquías de memoria para conseguir buenas prestaciones ha supuesto que los programadores, quienes solían pensar que la memoria es un dispositivo de acceso aleatorio de un solo nivel, ahora necesiten entender cómo funcionan las jerarquías de memoria para obtener buenas prestaciones. Posteriormente, utilizaremos dos ejemplos, figura 5.18 en la página 490, para mostrar la importancia de esta compresión.

Puesto que los sistemas de memoria son tan importantes para obtener buenas prestaciones, los diseñadores de computadores dedicaron muchos esfuerzos a estos sistemas y desarrollaron sofisticados mecanismos para mejorar las prestaciones del sistema de memoria. En este capítulo veremos las ideas conceptuales más impor-

tantes, aunque hemos utilizado muchas simplificaciones y abstracciones para que la materia sea manejable tanto en longitud como complejidad.

IDEA clave

Los programas muestran tanto localidad temporal, que es la tendencia a reutilizar los datos que previamente han sido accedidos, como localidad espacial, que es la tendencia a acceder datos que se localizan en posiciones de memoria cercanas a otras que han sido referenciadas recientemente. Las jerarquías de memoria aprovechan la localidad temporal guardando los datos recientemente accedidos en un lugar próximo al procesador. Las jerarquías de memoria aprovechan la localidad espacial transmitiendo hacia niveles superiores de la jerarquía bloques de varias palabras que se almacenan en posiciones contiguas de memoria.

La figura 5.3 muestra que una jerarquía de memoria utiliza cerca del procesador tecnologías de memoria más pequeñas y más rápidas. De este modo, los datos que se encuentran en el nivel más elevado de la jerarquía pueden ser procesados rápidamente. Los datos que no se encuentran en el nivel más alto, deben ser buscados en niveles inferiores de la jerarquía, los cuales son más grandes y más lentos. Si la frecuencia de aciertos en el nivel más alto es suficientemente elevada, la jerarquía de memoria se caracteriza por un tiempo de acceso que es próximo al tiempo de acceso del nivel más alto (y más rápido), y tiene una capacidad igual a la del nivel más bajo (y más grande).

En la mayoría de los sistemas, la memoria es una verdadera jerarquía, lo que significa que los datos no pueden estar presentes en el nivel *i* a menos que se encuentren también presentes en el nivel *i*+1.

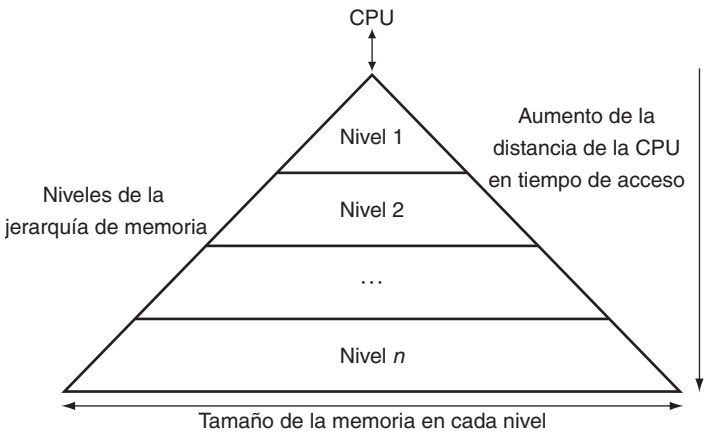


FIGURA 5.3 Este diagrama muestra la estructura de una jerarquía de memoria: a medida que aumenta la distancia desde el procesador, también aumenta la capacidad de almacenamiento. Esta organización, junto con sus condiciones de funcionamiento, permite al procesador tener un tiempo de acceso determinado principalmente por el nivel 1 de la jerarquía y además disponer de una memoria tan grande como la del nivel *n*. Inculcar esta forma de ver la jerarquía de memoria es el objetivo de este capítulo. Aunque el disco local es generalmente el nivel más bajo de la jerarquía, algunos sistemas utilizan periféricos de cinta o sistemas de ficheros en una red de área local como los siguientes niveles de la jerarquía.

Autoevaluación

¿Cuáles de las siguientes afirmaciones son generalmente ciertas?

1. Las memorias caches aprovechan la localidad temporal.
2. En una lectura a memoria, el valor devuelto depende de qué bloques se encuentran en la cache.
3. La mayor parte del coste de la jerarquía de memoria se corresponde con el del nivel más alto.
4. La mayor parte de la capacidad de la jerarquía de memoria se encuentra en el nivel más bajo.

5.2 Principios básicos de las caches

Cache: un lugar seguro para esconder o guardar cosas.

Nuevo Diccionario Mundial del Lenguaje Americano de Webster, Tercera Edición Académica (1988)

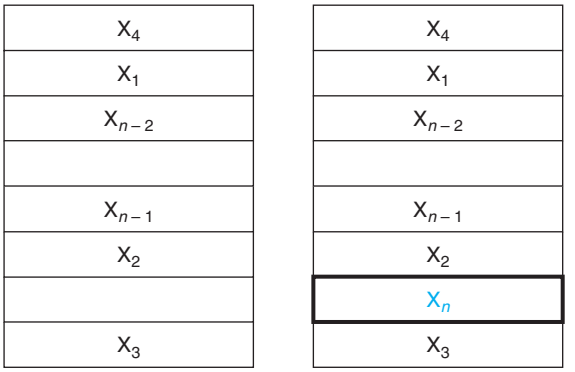
En nuestro ejemplo de la biblioteca, la mesa equivalía a la cache: un lugar seguro para almacenar cosas (libros) que necesitábamos consultar. *Cache* fue el nombre escogido para representar el nivel de la jerarquía que se sitúa entre el procesador y la memoria principal en el primer computador comercial que tuvo este nivel extra de memoria. Las memorias en el camino de datos del capítulo 4 sencillamente se reemplazan por caches. Hoy en día, aunque éste siga siendo el ámbito dominante donde se usa la palabra cache, el término se utiliza también para referirse a cualquier tipo de almacenamiento que aproveche la localidad de los accesos. Las cache aparecieron a principios de la década de 1960 en máquinas de investigación, y posteriormente, durante la misma década, en máquinas comerciales; cada computador de propósito general que se construye hoy en día, desde servidores hasta procesadores empotrados de bajo consumo, incluyen caches.

Empezaremos esta sección considerando una cache muy simple, en la cual tanto las solicitudes del procesador como los bloques tienen el tamaño de una palabra. (Los lectores que ya estén familiarizados con los fundamentos de las caches podrían saltarse esta sección e ir directamente a la sección 5.3). La figura 5.4 muestra esta sencilla cache, antes y después de solicitar un dato que inicialmente no se encuentra en la cache. Antes de la solicitud, la cache contiene un conjunto de datos recientemente solicitados X_1, X_2, ..., X_{n-1}, y el procesador solicita una palabra X_n que no se encuentra en la cache. Esta solicitud genera un fallo, y la palabra X_n es llevada desde la memoria principal a la cache.

Observando la situación de la figura 5.4, surgen dos preguntas a responder: ¿cómo sabemos si un dato está en la cache? Y además, si está, ¿cómo lo encontramos? Las respuestas a estas dos preguntas están relacionadas entre sí. Si cada palabra puede estar alojada en un único lugar de la cache, entonces es muy sencillo encontrar la palabra si ésta se encuentra en la cache. La forma más sencilla de asignar una posición de la cache a cada palabra de la memoria principal consiste en asignar una posición de la cache basándose en la *dirección* de la palabra en memoria principal. Esta estructura de la cache se denomina de **correspondencia directa**, puesto que cada posición de memoria principal se corresponde directamente con una única posición de la cache. La correspondencia típica entre direcciones y posiciones de la cache para caches de este tipo es normalmente muy sencilla. Por ejemplo, casi todas las caches de correspondencia directa utilizan la siguiente asignación

(Dirección de bloque) módulo (Número de bloques de la cache)

Cache de correspondencia directa: organización de cache en la que cada posición de la memoria principal se corresponde con una única posición de la cache.



a. Antes de acceder a X_n b. Después de acceder a X_n

FIGURA 5.4 Estado de la cache antes y después de que la palabra X_n haya sido solicitada, la cual no se encuentra inicialmente en la cache. Esta solicitud ocasiona un fallo que fuerza a la cache a traer X_n desde la memoria principal e insertarla dentro de la cache.

Esta asignación es atractiva ya que si el número de entradas de la cache es una potencia de dos, entonces la operación módulo puede ser realizada simplemente utilizando los $\log_2$ (capacidad de la cache en bloques) bits menos significativos de la dirección de memoria principal. De este modo, en una cache de 8 bloques se utilizan los tres bits menos significativos de la dirección del bloque ($8 = 2^3$). Por ejemplo, la figura 5.5 muestra cómo las direcciones de memoria principal entre 1_{diez} (00001_{dos}) y 29_{diez} (11101_{dos}) se corresponde con las localizaciones 1_{diez} (001_{dos}) y 5_{diez} (101_{dos}) en una cache de correspondencia directa de ocho palabras.

Ya que cada posición de la cache puede alojar el contenido de varias posiciones diferentes de memoria principal, ¿cómo sabemos si los datos de la cache se corresponden con la palabra solicitada? Es decir, ¿cómo sabemos si la palabra solicitada se encuentra o no en la cache? Respondemos a esta pregunta añadiendo un conjunto de **etiquetas** a la cache. Las etiquetas contienen la información de la dirección que se necesita para identificar si una palabra de la cache se corresponde con la palabra solicitada. La etiqueta sólo necesita incluir la parte más significativa de la dirección, lo cual se corresponde con los bits que no son utilizados por la cache como índice. Por ejemplo, en la figura 5.5 es necesario que la etiqueta sólo incluya los 2 bits más significativos de los 5 bits que forman la dirección, ya que el campo índice de la dirección, formado por sus 3 bits menos significativos, selecciona el bloque. Excluimos de la etiqueta los bits del índice debido a que son redundantes, ya que por definición, el campo índice de cualquier dirección de un bloque de cache es siempre el número de bloque.

Etiqueta: campo de cada una de las entradas de la tabla utilizada por la jerarquía de memoria que contiene información de la dirección que es necesaria para identificar si el bloque de información asociado se corresponde con la palabra solicitada.

Bit de validez: campo de cada una de las entradas de la tabla utilizada por la jerarquía de memoria que indica que el bloque asociado en la jerarquía contiene datos válidos.

Necesitamos también una forma de reconocer que la información contenida en un bloque de cache no tiene validez. Por ejemplo, cuando un procesador empieza a ejecutar un programa, la cache puede contener información que no tiene significado para el programa y por eso las etiquetas no tienen sentido. Incluso después de ejecutar muchas instrucciones, algunas de las entradas de la cache pueden estar vacías, como se observa en la figura 5.4. Por ello, se necesita saber que las correspondientes etiquetas deben ser ignoradas. El método más común consiste en añadir un **bit de validez** que indique si

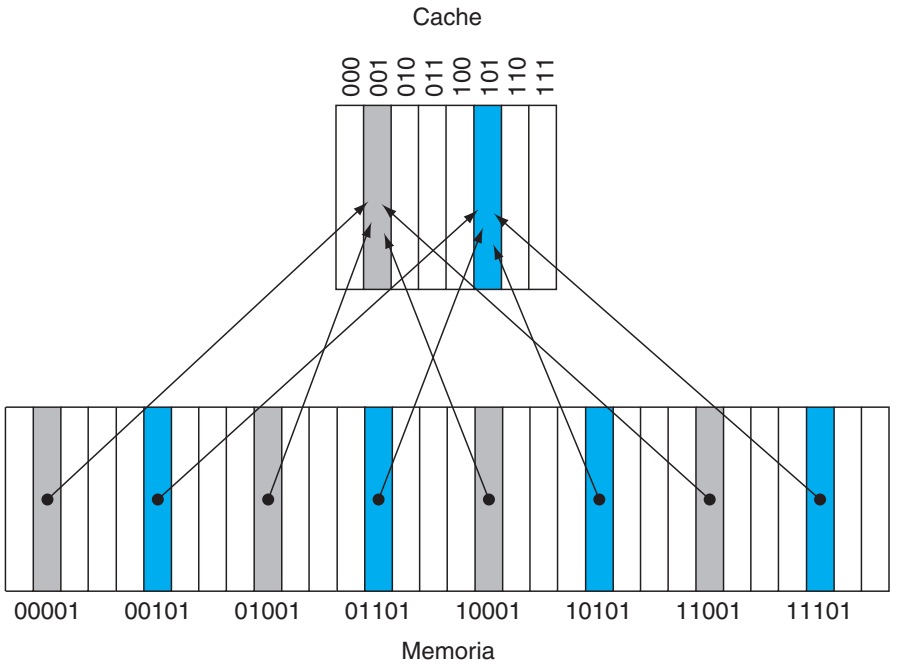


FIGURA 5.5 Una cache de correspondencia directa con ocho entradas donde se muestran las direcciones de palabra de memoria principal que van desde la 0 a la 31 y que se corresponden con las mismas entradas de la cache. Como existen ocho palabras en la cache, una dirección X se corresponde con la palabra de la cache X módulo 8. Es decir, los 3 bits menos significativos ($\log_2(8)$) se utilizan como índice la cache. De este modo, las direcciones 00001_{dos}, 01001_{dos}, 10001_{dos} y 11001_{dos} se corresponden todas con la entrada 001_{dos} de la cache, mientras que las direcciones 00101_{dos}, 01101_{dos}, 10101_{dos} y 11101_{dos} se corresponden todas con la entrada 101_{dos} de la cache.

una entrada contiene una dirección con información válida o no. Si el bit no está activado, el correspondiente bloque de datos no puede ser utilizado.

En el resto de esta sección nos centraremos en explicar cómo funcionan las lecturas a la cache. En general, el manejo de las operaciones de lectura a la cache es una tarea un poco más sencilla que el manejo de las operaciones de escritura, ya que las lecturas no necesitan cambiar el contenido de la cache. Después de describir cómo funcionan las lecturas y cómo se manejan los fallos de cache, examinaremos diseños de caches en computadores reales y detallaremos cómo estas caches manejan las escrituras.

Acceso a la cache

A continuación se muestra una secuencia de nueve referencias a memoria a una cache de 8 bloques inicialmente vacía, incluyendo la acción para cada referencia. La figura 5.6 muestra el estado de la cache y los cambios en cada fallo. Ya que la cache dispone de ocho bloques, los 3 bits menos significativos de la dirección proporcionan el número de bloque:

Dirección decimal del acceso	Dirección binaria del acceso	Acierto o fallo en la cache	Bloque de la cache asignado (donde se encuentra o se almacena)
22	10110_{dos}	fallo (7.6b)	$(10110_{dos} \bmod 8) = 110_{dos}$
26	11010_{dos}	fallo (7.6c)	$(11010_{dos} \bmod 8) = 010_{dos}$
22	10110_{dos}	acierto	$(10110_{dos} \bmod 8) = 110_{dos}$
26	11010_{dos}	acierto	$(11010_{dos} \bmod 8) = 010_{dos}$
16	10000_{dos}	fallo (7.6d)	$(10000_{dos} \bmod 8) = 000_{dos}$
3	00011_{dos}	fallo (7.6e)	$(00011_{dos} \bmod 8) = 011_{dos}$
16	10000_{dos}	acierto	$(10000_{dos} \bmod 8) = 000_{dos}$
18	10010_{dos}	fallo (7.6f)	$(10010_{dos} \bmod 8) = 010_{dos}$

Puesto que la cache está vacía, varías de las primeras referencias son fallos; el pie de la figura 5.6 describe las acciones para cada referencia a memoria. En la octava referencia se produce un conflicto en el acceso a un bloque. La palabra almacenada en la dirección 18 (10010_{dos}) de la memoria principal se lleva al bloque 2 de la cache (010_{dos}), reemplazando a la palabra almacenada en la dirección 26 (11010_{dos}) que ya estaba en el bloque 2 de la cache (010_{dos}). Este comportamiento permite a una cache aprovecharse de la localidad temporal: las palabras utilizadas recientemente reemplazan a las palabras que han sido menos utilizadas recientemente.

Esta situación es análoga a necesitar un libro de las estanterías y no disponer de más espacio sobre la mesa; alguno de los libros de la mesa debe volver a las estanterías. En una cache de correspondencia directa sólo existe un lugar donde alojar los elementos recientemente solicitados y de ahí que sólo exista una única opción para decidir qué reemplazar.

Sabemos dónde buscar en la cache cada una de las posibles direcciones: los bits menos significativos de una dirección pueden ser usados para encontrar la única entrada de la cache que se corresponde con la dirección. La figura 5.7 muestra cómo una dirección solicitada se divide en

- una etiqueta que se usa para compararla con el valor de la etiqueta almacenado en la cache
- un índice de la cache, que se utiliza para seleccionar el bloque de datos

El índice de un bloque de cache junto con el contenido de la etiqueta para este bloque, determina con precisión la dirección de memoria de la palabra almacenada en el bloque de cache. Ya que el campo índice es utilizado como una dirección para acceder a la cache y dado que un campo de n bits codifica hasta 2^n valores distintos, el número total de entradas de una cache de correspondencia directa debe ser potencia de dos. En la arquitectura MIPS, dado que las palabras están alineadas en múltiplos de 4 bytes, los 2 bits menos significativos de cada dirección determinan uno de los bytes que constituyen una palabra, y por ello se ignoran cuando se accede a la palabra de un bloque.

El número total de bits que se requiere para construir una cache está en función de la capacidad de la cache y del tamaño de la dirección debido a que la cache incluye tanto el almacenamiento para los datos como para las etiquetas. El tamaño del bloque utilizado más arriba fue de una palabra, pero normalmente es de varias palabras. Con las siguientes condiciones:

Índice	V	Etiqueta	Datos
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. Estado inicial de la cache después de encender el computador

Índice	V	Etiqueta	Datos
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{dos}	Memoria(10110_{dos})
111	N		

b. Después de resolver el fallo de la dirección (10110_{dos})

Índice	V	Etiqueta	Datos
000	N		
001	N		
010	Y	11_{dos}	Memoria (11010_{dos})
011	N		
100	N		
101	N		
110	Y	10_{dos}	Memoria (10110_{dos})
111	N		

c. Después de resolver el fallo de la dirección (11010_{dos})

Índice	V	Etiqueta	Datos
000	Y	10_{dos}	Memoria (10000_{dos})
001	N		
010	Y	11_{dos}	Memoria (11010_{dos})
011	N		
100	N		
101	N		
110	Y	10_{dos}	Memoria (10110_{dos})
111	N		

d. Después de resolver el fallo de la dirección (10000_{dos})

Índice	V	Etiqueta	Datos
000	Y	10_{dos}	Memoria (10000_{dos})
001	N		
010	Y	11_{dos}	Memoria (11010_{dos})
011	Y	00_{dos}	Memoria (00011_{dos})
100	N		
101	N		
110	Y	10_{dos}	Memoria (10110_{dos})
111	N		

e. Después de resolver el fallo de la dirección (00011_{dos})

Índice	V	Etiqueta	Datos
000	Y	10_{dos}	Memoria (10000_{dos})
001	N		
010	Y	10_{dos}	Memoria (10010_{dos})
011	Y	00_{dos}	Memoria (00011_{dos})
100	N		
101	N		
110	Y	10_{dos}	Memoria (10110_{dos})
111	N		

f. Después de resolver el fallo de la dirección (10010_{dos})

FIGURA 5.6 Estados de la cache después de resolver las peticiones de memoria que fallan en la cache, mostrando en binario los campos del índice y la etiqueta para la secuencia de direcciones de la página 461. La cache está inicialmente vacía, con todos los bits de validez (campo V de la cache) desactivados (N). El procesador solicita las siguientes direcciones 10110_{dos} (fallo), 11010_{dos} (fallo), 10110_{dos} (acierto), 11010_{dos} (acierto), 10000_{dos} (fallo), 00011_{dos} (fallo), 10000_{dos} (acierto), y 10010_{dos} (fallo). Las tablas muestran el estado de la cache después de que cada fallo a la cache haya sido resuelto. Cuando se solicita la dirección 10010_{dos} (18), la entrada de la cache para la dirección 11010_{dos} (26) debe ser reemplazada, y una posterior referencia a 11010_{dos} causará un nuevo fallo en la cache. La etiqueta contendrá sólo la parte más significativa de la dirección. La dirección completa de una palabra almacenada en el bloque i de la cache cuya etiqueta contiene j es $j \times 8 + i$, o lo que es lo mismo, la concatenación de la etiqueta j y el índice i. Por ejemplo, en la situación f de la cache de arriba, el índice 010_{dos} contiene una etiqueta 10_{dos} que corresponde a la dirección 10010_{dos}.

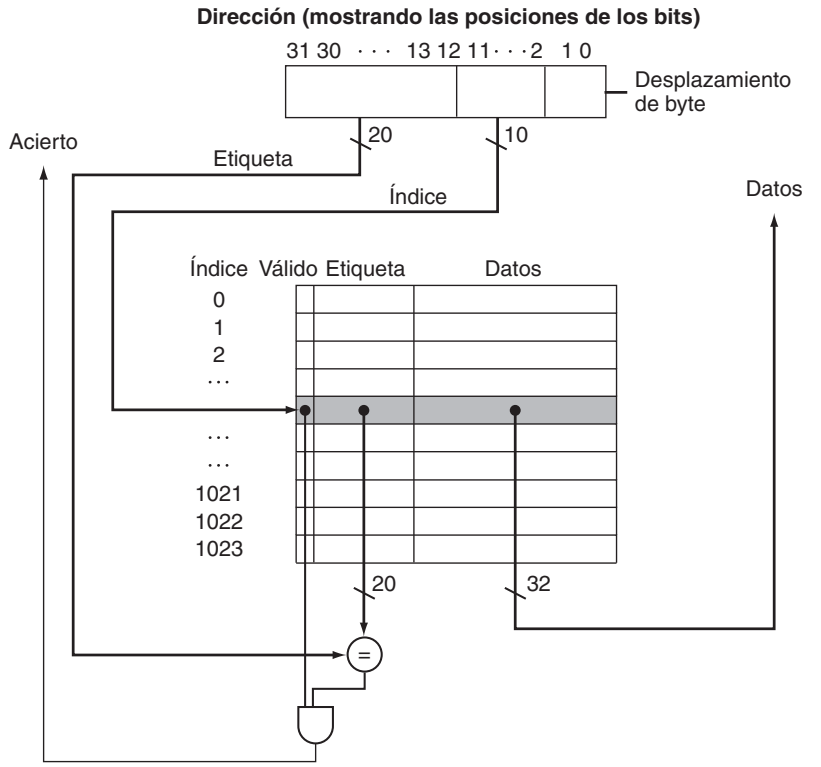


FIGURA 5.7 Para esta cache, la parte menos significativa de la dirección se utiliza para seleccionar una entrada de la cache que está formada por una palabra de datos y una etiqueta. Es una cache de 1024 palabras o 4 KB. En este capítulo supondremos que las direcciones son de 32 bits. La etiqueta almacenada en la cache se compara con la parte más significativa de la dirección para determinar si la entrada de la cache se corresponde con la dirección solicitada o no. Ya que la cache dispone de 2^{10} (o 1024) palabras y un tamaño de bloque de 1 palabra, se utilizan 10 bits para indexar la cache, dejando 32 – 10 – 2 = 20 bits para cada etiqueta. Si esta etiqueta y los 20 bits más significativos de la dirección coinciden y el bit de validez está activado, entonces la petición de memoria acierta en la cache, y la palabra es suministrada al procesador. En caso contrario, se produce un fallo.

- Direcciones de bytes de 32 bits
- Cache de correspondencia directa
- El tamaño de cache es 2^n bloques, por lo tanto se utilizan n bits para el índice
- El tamaño de bloque es 2^m palabras (2^{m+2} bytes), por lo tanto se utilizan m bits para identificar la palabra dentro del bloque y dos bits para identificar el byte de la dirección

el tamaño de la etiqueta, en bits, es

$$32 - (n + m + 2).$$

El número total de bits de una cache de correspondencia directa es

$2^n \times$ (tamaño de bloque + tamaño de la etiqueta + tamaño del campo de validez).

Puesto que el tamaño de bloque es 2^m palabras (2^{m+5} bits) y se necesita sólo 1 bit para el campo de validez, en número de bits de la cache es

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

Aunque este es el tamaño real en bits, la convención utilizada para referirse a la capacidad de la cache excluye el tamaño de la etiqueta y del campo de validez, para incluir sólo el tamaño de los datos. De este modo, la cache de la figura 5.7 se dice que es una cache de 4 KB.

Bits en una cache

EJEMPLO

¿Cuántos bits son necesarios para implementar una cache de correspondencia directa con 16 KB de datos y bloques de 4 palabras, suponiendo direcciones de 32 bits?

RESPUESTA

Sabemos que 16 KB se corresponden con 4K (2^{12}) palabras, y con un tamaño de bloque de 4 palabras (2^2), 1024 (2^{10}) bloques. Cada bloque tiene 4×32 ó 128 bits de datos además de una etiqueta, la cual está formada por 32 – 10 – 2 – 2 bits, a los que hay que añadir un bit de validez. De este modo, la capacidad total de la cache es

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

o 18.4 KB para una cache de 16 KB. Para esta cache, el número total de bits que forman la cache es aproximadamente 1.15 veces la capacidad necesaria para almacenar los datos.

Correspondencia de una dirección de memoria con un bloque multipalabra de cache

EJEMPLO

Suponga que existe de una cache con 64 bloques y un tamaño de bloque de 16 bytes. ¿Qué número de bloque se corresponde con la dirección de byte 1200?

RESPUESTA

Siguiendo la fórmula indicada en la página 457, el número de bloque viene dado por

(Dirección de bloque) módulo (Número de bloques de la cache)

donde la dirección del bloque es

$$\frac{\text{Dirección de byte}}{\text{Bytes por bloque}}$$

Observe que esta dirección de bloque es el bloque que contiene todas las direcciones que van desde

$$\left\lfloor \frac{\text{Dirección de byte}}{\text{Bytes por bloque}} \right\rfloor \times \text{Bytes por bloque}$$

y

$$\left\lfloor \frac{\text{Dirección de byte}}{\text{Bytes por bloque}} \right\rfloor \times \text{Bytes por bloque} + (\text{Bytes por bloque} - 1)$$

De esta manera, con 16 bytes por bloque, la dirección de byte 1200 se identifica con la siguiente dirección de bloque

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

la cual se corresponde con el bloque de cache número (75 módulo 64) = 11. De hecho, este bloque se corresponde con todas las direcciones que van desde 1200 hasta 1215.

Bloques más grandes aprovechan la localidad espacial para reducir la frecuencia de fallos. Como se puede observar en la figura 5.8, al aumentar el tamaño de bloque, la frecuencia de fallos normalmente se reduce. Sin embargo, la frecuencia de fallos puede aumentar si el tamaño de los bloques se hace demasiado grande, ya que el número total de bloques de la cache disminuye y habrá una gran competencia por ocupar esos bloques. Como resultado, el bloque será desalojado de la cache antes de que se acceda a muchas de sus palabras. Expresado de otra manera, la localidad espacial de las palabras de un bloque disminuye a medida que el bloque se hace muy grande; en consecuencia, los beneficios que proporciona la frecuencia de fallos se hacen cada vez más pequeños.

Un problema más serio asociado con el incremento del tamaño de bloque es que la penalización por fallo aumenta. La penalización por fallo viene determinada por el tiempo necesario para llevar el bloque desde el siguiente nivel inferior de la jerarquía hasta la cache. El tiempo para llevar el bloque tiene dos componentes: la latencia de la primera palabra y el tiempo de transferencia del resto del bloque. Obviamente, a no ser que cambiemos el sistema de memoria, el tiempo de transferencia (y por lo tanto, la penalización por fallo) aumentará a medida que el tamaño de bloque aumenta. Además, la mejora de la frecuencia de fallos comienza a disminuir a medida que el bloque es de tamaño mayor. Como resultado, el aumento de la penalización por fallo no es compensado por la disminución de la frecuencia de fallos para bloques grandes, y por eso las prestaciones de la cache disminuyen. Por supuesto, si se diseña el sistema de memoria de forma tal que los bloques más grandes son los que se transfieren más eficientemente, podemos aumentar el tamaño del bloque y mejorar las presentaciones de la cache. Discutiremos este tema en la siguiente sección.

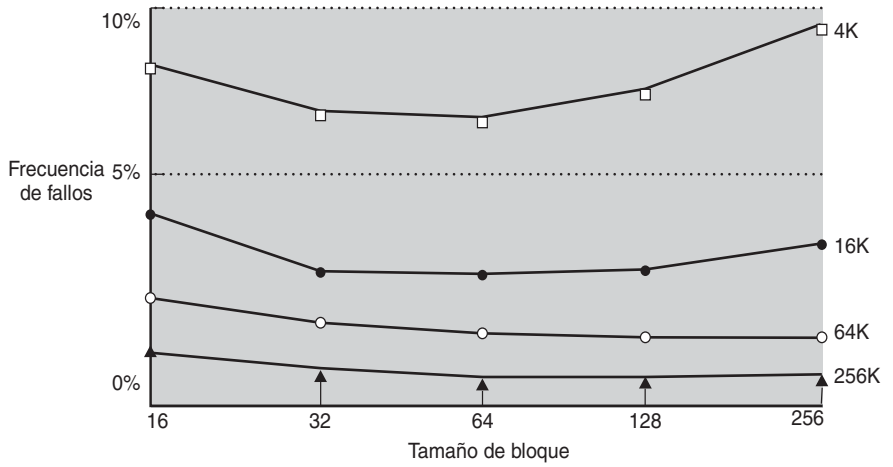


FIGURA 5.8 Frecuencia de fallos frente a tamaño de los bloques. Observe que la frecuencia de fallos aumenta realmente si el tamaño de los bloques es demasiado grande respecto a la capacidad de la cache. Cada curva representa una cache de distinta capacidad. (Esta figura es independiente de la asociatividad, la cual se discutirá posteriormente). Desafortunadamente, las simulaciones de los programas SPEC2000 tardarían demasiado tiempo si el tamaño de bloque fuera tomado en consideración, por eso, estos datos fueron obtenidos con programas SPEC92.

Extensión: Aunque es complicado mejorar la componente latencia de la penalización por fallo en bloques grandes, somos capaces de solapar parte del tiempo de transferencia de forma tal que el efecto de la penalización por fallo se reduce. La forma más sencilla de hacer esto, denominada *comienzo inmediato*, consiste simplemente en reanudar la ejecución tan pronto como la palabra solicitada del bloque es recibida, en vez de esperar a que termine de transferirse el bloque entero. Muchos procesadores usan esta técnica para acceder a las instrucciones, donde funciona mejor. Los accesos a instrucciones son en su mayor parte secuenciales, así que si el sistema de memoria puede proporcionar una palabra cada ciclo de reloj, el procesador puede ser capaz de comenzar a procesar la instrucción que ha producido el fallo cuando es recibida, y en los siguientes ciclos de reloj seguir recibiendo a tiempo las nuevas instrucciones. Esta técnica es normalmente menos efectiva para las caches de datos debido a que probablemente las palabras sean solicitadas de una forma menos predecible, y a la alta probabilidad de que el procesador necesite otra palabra de otro bloque antes que la transferencia finalice. Si el procesador no puede acceder a la cache de datos debido a que se está realizando una transferencia, entonces debe pararse.

Una técnica incluso más sofisticada consiste en organizar la memoria principal de forma tal que la palabra solicitada sea la primera en trasferirse desde la memoria principal a la cache. El resto del bloque se transfiere a continuación, comenzado con la dirección que se encuentra después de la palabra solicitada, y una vez llegado al final del bloque se sigue transfiriendo desde su comienzo. Esta técnica, denominada *palabra solicitada primero* o *palabra crítica primero*, puede ser ligeramente más rápida que la de comienzo inmediato, pero está limitada por las mismas propiedades que limitan el comienzo inmediato.

Manejo de los fallos de cache

Antes de analizar la cache de un sistema real, veamos cómo trata la unidad de control los **fallos de cache**. (El controlador de cache se describe en detalle en la sección 5.7.) La unidad de control debe detectar y procesar un fallo trayendo los datos solicitados

Fallo de cache: solicitud de acceso a datos de la cache que no puede ser atendida debido a que los datos no están presentes en la cache.

desde la memoria principal (o como veremos, desde un nivel inferior de cache). Si el acceso acierta en la cache, el computador continúa utilizando los datos como si lo que ha ocurrido es algo normal.

La modificación del control de un procesador para manejar aciertos es trivial; los fallos, sin embargo, requieren trabajo adicional. El manejo de los fallos de cache se realiza con la unidad de control del procesador y con un controlador independiente que pone en marcha el acceso a memoria y actualiza el contenido de la cache. El procesamiento de un fallo de cache genera una parada del procesador (capítulo 4) a diferencia de una interrupción, la cual requiere guardar el estado de todos los registros. Para un fallo de cache, podemos realizar la paralización de todo el procesador simplemente no actualizando los contenidos de los registros tanto temporales como los que son visibles al programador, mientras que se espera por la memoria principal. Procesadores más sofisticados con ejecución fuera de orden, permiten la ejecución de instrucciones durante la espera por un fallo de cache, pero en el resto de esta sección supondremos procesadores con ejecución en orden, que paran durante el fallo de cache.

Veamos con un poco más de detalle cómo se manejan los fallos de instrucciones; el mismo planteamiento puede aplicarse al manejo de los fallos de datos. Si un acceso a una instrucción produce un fallo, entonces el contenido del Registro de Instrucciones no es válido. Para obtener de la cache la instrucción correcta, debemos ser capaces de indicarle al nivel inferior de la jerarquía de memoria que realice una lectura. Ya que el contador de programa se incrementa en cada ciclo de reloj de la ejecución, la dirección de la instrucción que genera un fallo en la cache de instrucciones coincide con el valor del contador de programa restándole 4. Una vez que se conozca esta dirección, se indica a la memoria principal que realice una lectura. Esperamos a que la memoria principal responda (ya que el acceso tardará varios ciclos), y escribimos las palabras que contienen la instrucción solicitada en la cache.

Ahora podemos establecer los pasos involucrados en el manejo de un fallo de la cache de instrucciones:

1. Enviar el valor del PC original (PC actual – 4) a la memoria.
2. Indicarle a la memoria principal que realice una lectura y que después espere a que la memoria termine este acceso.
3. Actualizar la entrada de la cache, guardando el dato traído desde la memoria en la parte de la entrada destinada a los datos, actualizando los bits más significativos de la dirección (proporcionados por la ALU) en la parte destinada a la etiqueta y modificando el bit de validez para indicar entrada válida.
4. Reanudación de la ejecución de la instrucción desde su comienzo, que buscará de nuevo la instrucción y esta vez la encontrará en la cache.

El control de la cache cuando se produce un acceso a datos es fundamentalmente idéntico: en un fallo, simplemente paralizamos el procesador hasta que la memoria devuelva los datos.

Manejo de las escrituras

Las escrituras funcionan de una forma un poco distinta. Suponga que aparece una instrucción de almacenamiento con la que sólo escribimos datos en la cache de datos (sin modificar la memoria principal); entonces, después de actualizar la cache, la

memoria guarda un valor distinto del que se guarda en la cache. En tal caso se dice que la cache y la memoria son *incoherentes*. La forma más simple de mantener la coherencia de la memoria principal y la cache consiste en escribir siempre los datos tanto en la memoria como en la cache. Esta técnica se denomina **escritura directa**.

Escritura directa: técnica en la cual las escrituras siempre actualizan tanto la cache como la memoria principal, asegurando que los datos siempre son coherentes en ambas memorias.

El otro aspecto clave de las escrituras consiste en explorar lo que ocurre en los fallos de escritura. Primero se van a buscar a memoria las palabras del bloque. Después de que el bloque se haya recibido y guardado en la cache, podemos sobrescribir la palabra que originó el fallo en un bloque de cache. La palabra también se escribe en memoria principal utilizando toda la dirección.

Aunque este diseño maneja las escrituras de una forma muy simple, no proporcionaría muy buenas prestaciones. Con un esquema de escritura directa, cada escritura obliga a que los datos sean escritos en la memoria principal. Estas escrituras tardarán mucho tiempo, probablemente al menos 100 ciclos del reloj del procesador, y podrían ralentizar considerablemente al procesador. Por ejemplo, supongamos que el 10% de las instrucciones son almacenamientos. Si el CPI sin considerar fallos de cache era 1.0, esperar 100 ciclos adicionales en cada escritura llevaría a que el CPI fuera $1.0 + 100 \times 10\% = 11$, reduciendo las prestaciones en un factor superior a 10.

Una solución a este problema consiste en usar un **búfer de escritura**. Un búfer de escritura almacena los datos mientras se están escribiendo en memoria. Después de escribir los datos tanto en la cache como en el búfer de escritura, el procesador puede continuar la ejecución. Cuando finaliza una escritura a memoria principal, la correspondiente entrada en el búfer de escritura se libera. Si el búfer de escritura se llena cuando el procesador ejecuta una escritura, el procesador debe pararse hasta que exista una entrada libre en el búfer de escritura. Por supuesto, si el ritmo al que la memoria principal puede completar las escrituras es inferior al ritmo con el que el procesador está generando las escrituras, ningún tamaño del búfer puede impedir las paradas debido a que las escrituras se generan a mayor ritmo del que el sistema de memoria puede aceptarlas.

Búfer de escritura: cola que almacena temporalmente los datos mientras que los datos esperan a que sean escritos en memoria principal.

El ritmo con el cual se generan las escrituras puede también ser inferior al ritmo que la memoria puede aceptarlas, y todavía producirse paradas. Esto puede ocurrir cuando las escrituras se producen a ráfagas. Para reducir la aparición de tales paradas, los procesadores normalmente aumentan el número de entradas del búfer de escritura.

La alternativa a la técnica de escritura directa corresponde a la técnica denominada **escritura retardada**. En un esquema de escritura retardada, cuando se produce una escritura, el nuevo valor se escribe sólo en el bloque de la cache. El bloque modificado se guarda en el nivel inferior de la jerarquía de memoria sólo cuando va a ser reemplazado. Los esquemas de escritura retardada pueden mejorar las prestaciones, especialmente en los casos en que los procesadores pueden generar escrituras con tanta o mayor frecuencia que la utilizada por la memoria principal para manejarlas. Sin embargo, un esquema de escritura retardada es más complejo de implementar que el de escritura directa.

Escritura retardada: técnica que maneja las escrituras primero actualizando sólo los valores del bloque de la cache y posteriormente guardando el bloque en el nivel inferior de la jerarquía cuando este bloque sea reemplazado.

En el resto de esta sección describiremos caches integradas en procesadores reales, y examinaremos cómo se manejan tanto las lecturas como las escrituras. En la sección 5.5 describiremos con más detalle el manejo de las escrituras.

Extensión: Las escrituras introducen en la cache varias complicaciones que no están presentes en las lecturas. Veamos ahora dos de ellas: la política de los fallos de escritura y la implementación eficiente de las escrituras en la cache de escritura retardada.

Consideremos un fallo en una cache de escritura directa. La estrategia más habitual consiste en reservar un bloque en la cache, llamado *reserva de escritura*. El bloque se lee de la memoria y la parte correspondiente del bloque se sobreescribe. Una estrategia alternativa, llamada *sin reserva de escritura*, consiste en actualizar la parte del bloque en la memoria, pero sin traerla a la cache. La motivación de esta técnica es que a veces los programas escriben bloques enteros de datos, por ejemplo cuando el sistema operativo escribe ceros en todas las palabras de una página de memoria. En estos casos, la búsqueda asociada con los primeros fallos de escritura puede ser innecesaria. En algunos computadores la estrategia de reserva de escritura se cambia por una estrategia basada en páginas de memoria.

Implementar eficientemente los almacenamientos en una cache que utiliza la estrategia de escritura retardada es más complejo que en una cache de escritura directa. Una cache de escritura directa puede escribir el dato en la cache y luego leer la etiqueta. Si la etiqueta no coincide se produce un fallo. Como la cache es de escritura directa, la sobreescritura del bloque no es catastrófica ya que la memoria dispone del valor correcto. En una cache de escritura retardada, el contenido del bloque debe guardarse en la memoria principal si el dato en la cache ha sido modificado y se produce un fallo de cache. Si en un almacenamiento simplemente se sobreescribe el bloque de cache antes de saber si el acceso acierta en la cache (como ocurre en la cache de escritura directa), se perdería el contenido del bloque, que no ha sido actualizado previamente en el siguiente nivel de la jerarquía de memoria.

En una cache de escritura retardada, como no se puede sobrescribir el bloque, los almacenamientos requieren o bien dos ciclos (un ciclo para comprobar que se acierta seguido de otro ciclo en el que se realiza la escritura) o bien un registro adicional denominado *búfer de almacenamiento*, para alojar los correspondientes datos (permitiendo de hecho que el almacenamiento tarde un solo ciclo si se utiliza segmentación). Cuando se utiliza un búfer de almacenamiento, el procesador realiza la inspección de la cache y guarda los datos en el búfer de almacenamiento a lo largo del ciclo que se usa normalmente para el acceso a la cache. Suponiendo que se acierta en la cache, los nuevos datos se transfieren desde el búfer de almacenamiento hasta la cache durante un siguiente ciclo de acceso a la cache que no se utilice.

Por comparación, en una cache de escritura directa, los almacenamientos siempre pueden ser realizados en un ciclo. Se lee la etiqueta y se escribe la parte de los datos del bloque seleccionado. Si la etiqueta se corresponde con la dirección del bloque que está siendo parcialmente escrito, el procesador puede continuar normalmente, ya que el bloque correcto ha sido actualizado. Si la etiqueta no se corresponde, el procesador genera un fallo de escritura para buscar el resto del bloque al que corresponde tal dirección.

Muchas caches de escritura retardada también incluyen búferes de escritura que se utilizan para reducir la penalización por fallo cuando un fallo reemplaza un bloque modificado. En tal caso, el bloque modificado se almacena temporalmente en el búfer de escritura retardada que está asociado a la cache mientras el bloque solicitado se lee desde la memoria principal. El contenido del búfer de escritura retardada es posteriormente actualizado en la memoria principal. Suponiendo que otro fallo no aparece inmediatamente, esta técnica reduce a la mitad la penalización cuando un bloque no coherente debe ser reemplazado.

Un ejemplo de cache: el procesador FastMATH de Intrinsity

El FastMATH de Intrinsity es un microprocesador empotrado rápido que utiliza la arquitectura MIPS y una implementación sencilla de la cache. Cerca del final del capítulo, examinaremos el diseño de cache más complejo del AMD Opteron X4 (Barcelona), pero por razones pedagógicas comenzaremos con este ejemplo que es simple pero real. La figura 5.9 muestra la organización de la cache de datos del FastMATH de Intrinsity.

Este procesador tiene un camino de datos segmentado de 12 etapas, similar al descrito en el capítulo 4. Cuando opera a velocidad pico, el procesador puede solicitar una palabra de instrucción y otra palabra de datos en cada ciclo de reloj. Para satisfacer las demandas del camino de datos segmentado sin que se produzcan paradas, las caches de instrucciones y de datos están separadas. Cada cache es de 16 KB, o 4K palabras, con bloques de 16 palabras.

Las solicitudes de lectura para la cache son simples. Debido a que existen caches de datos e instrucciones separadas, se necesitarán distintas señales de control para leer y escribir cada cache. (Recuerde que se necesita actualizar la cache de instrucciones cuando se produce un fallo). De este modo, los pasos que se siguen en cada cache para una solicitud de lectura son los siguientes:

1. Se envía la dirección a la cache apropiada. La dirección proviene, bien desde el PC (para una instrucción), bien de la ALU (para datos).

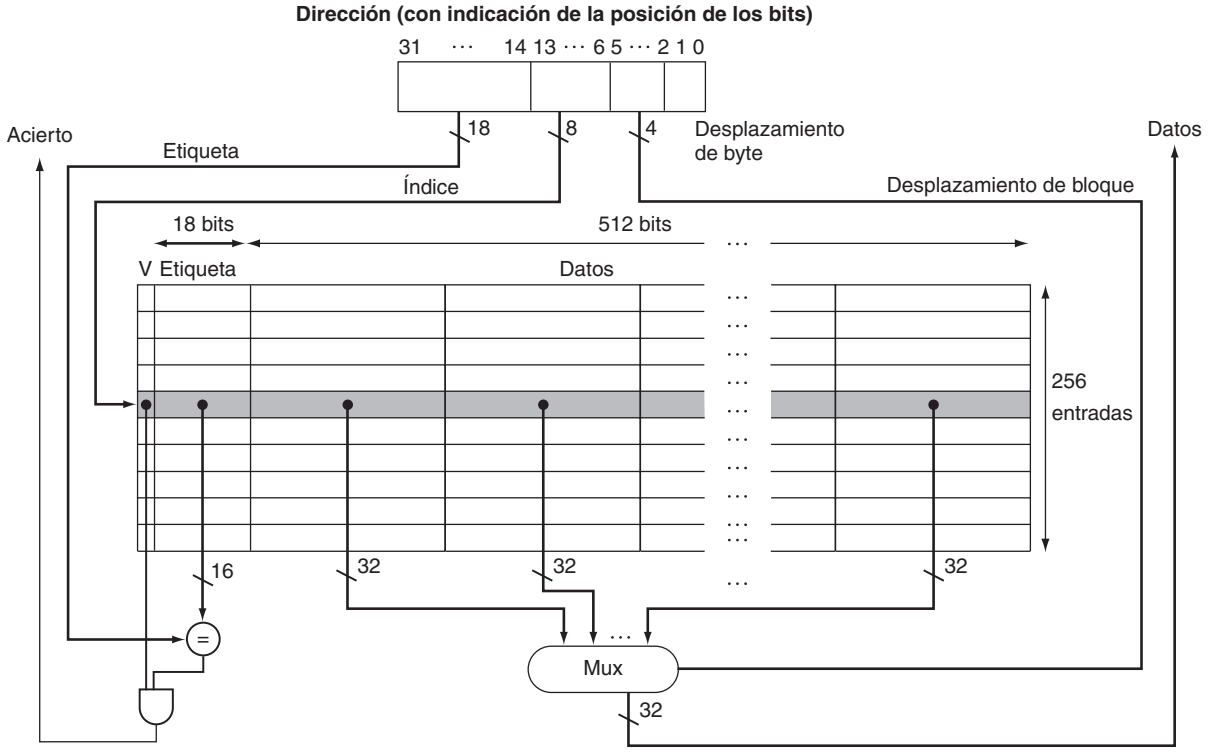


FIGURA 5.9 Las caches de 16 KB en el FastMATH de Intrinsity, cada una de ellas contiene 256 bloques con 16 palabras por bloque. El campo etiqueta ocupa 18 bits y el campo índice ocupa 8 bits, mientras que un campo de 4 bits (bits 5-2) se usa para indexar el bloque y seleccionar la palabra del bloque por medio de un multiplexor 16-a-1. En la práctica, para eliminar el multiplexor, las caches combinan una RAM con mayor rango de direccionamiento para los datos y una RAM con menor rango de direccionamiento para las etiquetas, con los bits adicionales de la dirección que determinan el desplazamiento dentro del bloque en la RAM de datos. En este caso, la RAM más grande tiene un ancho de palabra de 32 bits y debe disponer de un número de palabras que es 16 veces el número de bloques de la cache.

2. Si se acierta en la cache, la palabra solicitada se encuentra disponible en las líneas de datos. Ya que existen 16 palabras en el bloque deseado, necesitamos seleccionar la palabra que se pide. Un campo del índice del bloque se usa para controlar el multiplexor (que aparece en la parte inferior de la figura), el cual selecciona la palabra solicitada de entre las 16 palabras del bloque indexado.
3. Si se falla en la cache, la dirección se envía a la memoria principal. Cuando la memoria devuelve el dato, se escribe en la cache y luego éste se lee para completar la operación solicitada.

Para los almacenamientos, el FathMATH de Intrinsity ofrece tanto escrituras directas como escrituras retardadas, dejando al sistema operativo que decida qué estrategia usar para una aplicación. El microprocesador dispone de un búfer de escritura de una sola entrada.

¿Cuál es la frecuencia de fallos que se alcanza con una cache cuya estructura es como la que usa el FastMATH de Intrinsity? La figura 5.10 muestra las frecuencias de fallos de las caches de instrucciones y de datos. La frecuencia de fallos combinada es la frecuencia de fallos efectiva por acceso que muestra cada programa después de considerar las distintas frecuencias de accesos a instrucciones y datos.

Aunque la frecuencia de fallos es una característica importante del diseño de caches, la medida final reflejará el efecto del sistema de memoria sobre el tiempo de ejecución de los programas. Dentro de poco veremos cómo se relacionan la frecuencia de fallos y el tiempo de ejecución.

Cache separada: técnica en la cual un nivel de la jerarquía de memoria se compone de dos caches independientes que funcionan en paralelo, una de ellas destinada a manejar instrucciones y la otra a manejar datos.

Extensión: Una cache combinada con una capacidad total igual a la suma de dos **caches separadas** tendrá normalmente una mayor frecuencia de aciertos. Ellos se debe a que la cache combinada no diferencia estrictamente las entradas que pueden ser usadas por las instrucciones de las que son usadas por los datos. Aún así, muchos procesadores utilizan caches separadas para instrucciones y datos con el objetivo de aumentar el *ritmo de transferencia o ancho de banda* (*bandwidth*). (Puede haber, también, menos fallos de conflicto; véase sección 5.5.)

A continuación se dan las frecuencias de fallos para caches del tamaño que se encuentra en el procesador FastMATH de Intrinsity y para una cache combinada cuya capacidad es igual al total de las dos caches.

- Capacidad total de la cache: 32 KB
- Frecuencia de fallos efectiva de la cache separada: 3.24%
- Frecuencia de fallos de la cache combinada: 3.18%

La frecuencia de fallos de la cache separada resulta ser sólo ligeramente peor.

Frecuencia de fallos para las instrucciones	Frecuencia de fallos para los datos	Frecuencia de fallos combinada
0.4%	11.4%	3.2%

FIGURA 5.10 Frecuencias aproximadas de fallos para instrucciones y datos en el procesador FastMATH de Instrinsity que se obtienen con los programas de prueba SPEC2000. La frecuencia de fallos combinada es la frecuencia de fallos efectiva que experimenta una combinación formada por una cache de instrucciones de 16 KB y una cache datos de 16 KB. Se obtiene factorizando las frecuencias de fallos individuales por la frecuencia de accesos a instrucciones y datos.

La ventaja de doblar el ancho de banda de la cache a través del acceso simultáneo a una instrucción y a un dato, supera fácilmente la desventaja de una frecuencia de fallos ligeramente mayor. Esta observación nos recuerda que no podemos utilizar la frecuencia de fallos como única medida de las prestaciones de la cache, como se muestra en la sección 5.3.

Diseño del sistema de memoria para soportar caches

Los fallos de cache se resuelven con los datos de la memoria principal, la cual se fabrica con DRAMs. En la sección 5.1 vimos que el diseño de las DRAMs persigue optimizar la densidad del chip en vez del tiempo de acceso. Aunque es difícil reducir la latencia para recibir la primera palabra desde la memoria principal, podemos reducir la penalización por fallo si aumentamos el ancho de banda desde la memoria a la cache. Esta reducción permite usar tamaños más grandes de bloque manteniendo una baja penalización por fallo, similar a aquella que caracteriza a bloques más pequeños.

Por regla general, el procesador está conectado a la memoria a través de un bus. (Como se verá en el capítulo 6, esta forma de conexión está cambiando, pero en este capítulo nos tiene sin cuidado la tecnología actual de interconexión, por lo que usaremos el término bus.) La frecuencia de reloj del bus normalmente es muy inferior a la del procesador. La frecuencia de este bus afecta a la penalización por fallo.

Para comprender el impacto de distintas organizaciones de memoria, definamos un conjunto de tiempos de acceso a memoria hipotéticos. Tomemos

- 1 ciclo de reloj del bus de memoria para enviar la dirección
- 15 ciclos de reloj del bus de memoria para cada acceso iniciado en la DRAM
- 1 ciclo de reloj del bus de memoria para enviar una palabra de datos

Si tenemos un bloque de cache de cuatro palabras y un banco de DRAM de una palabra de ancho, la penalización por fallo sería $1 + 4 \times 15 + 4 \times 1 = 65$ ciclos de reloj del bus de memoria. De este modo, el número de bytes transferidos por ciclo de reloj en un fallo individual sería

$$\frac{4 \times 4}{65} = 0.25$$

La figura 5.11 muestra tres opciones para diseñar el sistema de memoria. La primera opción obedece a lo que hemos estado suponiendo: una memoria de una palabra de ancho, y todos los accesos se realizan de forma secuencial. La segunda opción aumenta el ancho de banda de la memoria utilizando una memoria y un bus procesador-memoria más anchos; esto permite realizar accesos paralelos a varias palabras del bloque. La tercera opción aumenta el ancho de banda por medio de una memoria más ancha pero sin aumentar el ancho del bus de interconexión. De este modo, todavía se requiere un tiempo para transmitir cada palabra, pero se evita esperar una latencia que sea superior a la de un solo acceso. Observemos en cuánto estas dos opciones mejoran la penalización por fallo de 65 ciclos que calculamos para la primera de las opciones (figura 5.11a).

Aumentando el ancho de la memoria y el ancho del bus se aumenta proporcionalmente el ancho de banda, y se reducen las componentes de la penalización por fallo que corresponden tanto al tiempo de acceso como al tiempo de transferencia. Con un ancho de memoria principal de dos palabras, la penalización por fallo se

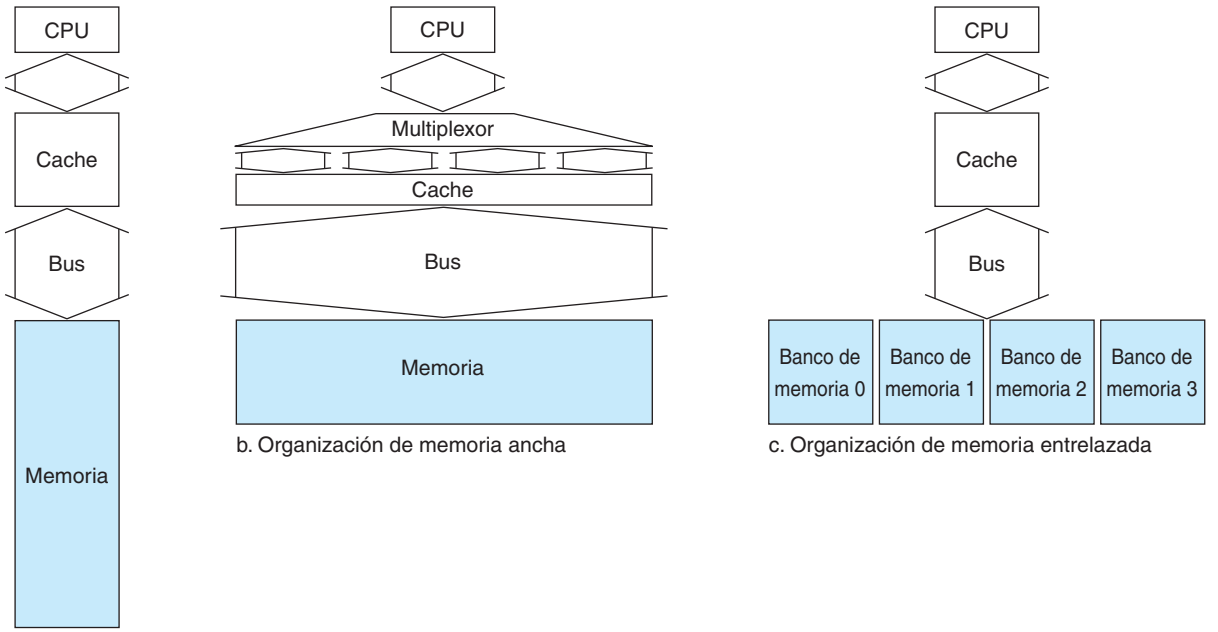


FIGURA 5.11 El principal método para conseguir un mayor ancho de banda de memoria consiste en incrementar el ancho físico y lógico del sistema de memoria. En esta figura, el ancho de banda de memoria se mejora de dos maneras. El diseño más simple, (a), utiliza una memoria donde todos los componentes son de una palabra de ancho; (b) muestra una memoria, bus, y cache más anchas; mientras que (c) muestra un bus y una cache más estrechos con una memoria entrelazada. En (b), la lógica entre la cache y el procesador está formada por un multiplexor que se usa en las lecturas y por una lógica de control que actualiza las palabras apropiadas de la cache en las operaciones de escritura.

reduce de los iniciales 65 ciclos de reloj del bus de memoria a $1 + 2 \times 15 + 2 \times 1 = 33$ ciclos de reloj del bus de memoria. El ancho de banda para un único fallo es de 0.48 (casi el doble) bytes por ciclo de reloj del bus considerando una memoria de dos palabras de ancho. Los costes más importantes de esta mejora corresponden a un bus más ancho y al posible incremento del tiempo de acceso a la cache debido al multiplexor y a la lógica de control que aparecen entre el procesador y la cache.

En vez de hacer más ancha toda la ruta entre la memoria y la cache, los chips de memoria pueden ser organizados en bancos para leer o escribir muchas palabras en un solo tiempo de acceso en vez de leer o escribir una única palabra cada vez. Cada banco podría ser de una palabra de ancho de forma que el ancho del bus y la cache no necesiten cambiar, pero el envío de una dirección a varios bancos permite leerlos todos simultáneamente. Este esquema que se denomina *entrelazado*, tiene la ventaja de que se necesita esperar sólo la latencia de un único acceso a memoria. Por ejemplo, con cuatro bancos, el tiempo para obtener un bloque de cuatro palabras se calcularía sumando 1 ciclo para transmitir la dirección y la petición de lectura a los bancos, 15 ciclos para que todos los bancos accedan a memoria, y 4 ciclos para enviar las cuatro palabras a la cache. Esto conlleva una penalización por fallo de $1 + 1 \times 15 + 4 \times 1 = 20$ ciclos de reloj del bus de

memoria. Esto implica que el ancho de banda por fallo es de 0.80 bytes por ciclo, o unas tres veces el ancho de banda de una memoria y un bus de una palabra de ancho. Los bancos son también valiosos para las escrituras. Cada banco puede escribir independientemente, cuadriplicando el ancho de banda de las escrituras y proporcionando menos paradas en una cache de escritura directa. Como veremos, una estrategia alternativa para las escrituras hace que el entrelazado sea una técnica todavía más atractiva.

Debido a la gran implantación de las caches y al deseo de disponer de tamaños de bloque mayores, los fabricantes de DRAM mantienen un acceso rápido a datos (*burst acces*) situados en una serie de posiciones consecutivas en la DRAM. El desarrollo más actual es la *DRAM de doble frecuencia de datos (Double data rate DRAM),* o DDR DRAM. En esta DRAM los datos se transfieren tanto en el flanco de subida del reloj como en el de bajada, doblando así el ancho de banda respecto al que se podría esperar viendo el ciclo de reloj y el ancho de los datos. Para alcanzar este ancho de banda, la DRAM se organiza internamente como una memoria entrelazada.

La ventaja de esta optimización es que se obtiene una mejora significativa en el ancho de banda utilizando mayoritariamente la circuitería ya disponible en la DRAM, con un pequeño coste añadido. En la sección C.9 en el **apéndice C** se describe la arquitectura interna de la DRAM y la implementación de estas optimizaciones.

Extensión: Los chips de memoria están organizados para proporcionar varios bits a su salida, normalmente entre 4 y 32, siendo 16 el valor más común en 2008. Describimos la organización de una RAM de $d \times w$, donde d es el número de posiciones de memoria direccionables (el rango de direccionamiento) y w es la salida (o ancho de cada posición de memoria). Las DRAMs están organizadas de forma lógica en matrices rectangulares, y su tiempo de acceso se divide en una parte que cuantifica el acceso a las filas y otra parte que cuantifica el acceso a las columnas. Las DRAMs utilizan un búfer para almacenar temporalmente una fila de bits en el interior de la DRAM para el posterior acceso a las columnas. También disponen de señales opcionales de temporización que permiten accesos sucesivos al búfer sin que se tenga que incurrir de nuevo en un tiempo de acceso a la fila. El búfer actúa como una SRAM; cambiando la dirección de la columna, se puede acceder a bits aleatorios hasta que se produzca el siguiente acceso a una fila. Esta posibilidad modifica significativamente el tiempo de acceso, ya que el tiempo de acceso a los bits de la fila es mucho más corto. La figura 5.12 muestra cómo la densidad, el coste económico y el tiempo de acceso de las DRAMs han cambiado a lo largo de los años.

Para mejorar la interfaz con los procesadores, se han añadido señales de reloj a las DRAM y se denominan de forma más apropiada DRAM Síncronas, o SDRAM. La ventaja de las SDRAM es que la presencia de una señal de reloj elimina el tiempo necesario para la sincronización del procesador y la memoria.

Extensión: Una manera de medir las prestaciones del sistema de memoria detrás de la cache son los programas de prueba Stream [McCalpin, 1995]. Miden las prestaciones de operaciones vectoriales largas. No tienen localidad temporal y acceden a vectores y matrices que son mayores que la cache del computador que se está probando.

Extensión: La estructura mejorada que se ha descrito para la memoria DRAM, *burst access*, se utiliza también en los buses de memoria, por ejemplo, en el bus Intel Duo Core Front Side.

Año de aparición	Capacidad del chip	Dólares por GB	Tiempo total de acceso a una nueva fila/columna	Tiempo de acceso a una fila existente
1980	64 Kbit	$1 500 000	250 ns	150 ns
1983	256 Kbit	$500 000	185 ns	100 ns
1985	1 Mbit	$200 000	135 ns	40 ns
1989	4 Mbit	$50 000	110 ns	40 ns
1992	16 Mbit	$15 000	90 ns	30 ns
1996	64 Mbit	$10 000	60 ns	12 ns
1998	128 Mbit	$4 000	60 ns	10 ns
2000	256 Mbit	$1 000	55 ns	7 ns
2004	512 Mbit	$250	50 ns	5 ns
2007	1 Gbit	$50	40 ns	1.25 ns

FIGURA 5.12 Hasta 1996, la capacidad de las DRAMs se cuadruplicaba cada tres años aproximadamente, y desde entonces el aumento es mucho menor. Las mejoras en el tiempo de acceso han sido más lentas pero continuas, y el coste económico casi sigue la trayectoria marcada por las mejoras en la densidad del chip, aunque al coste a menudo le afectan otros aspectos como la disponibilidad y la demanda. En el coste por megabyte no se ha considerado la inflación.

Resumen

Comenzamos la sección anterior examinando las caches más sencillas: una cache de correspondencia directa con un bloque de una palabra. En esta cache, tanto los aciertos como los fallos son sencillos, ya que una palabra puede alojarse sólo en una posición y existe una etiqueta distinta para cada palabra. Para mantener la coherencia entre la cache y la memoria principal, se puede usar el método de escritura directa, tal que cada escritura en la cache obliga a actualizar la memoria principal. La alternativa a la escritura directa es método de escritura retardada que guarda un bloque en memoria principal cuando es reemplazado en la cache; este método se abordará con mayor detalle en las siguientes secciones.

Para aprovechar la localidad espacial, una cache debe tener un tamaño de bloque superior a una palabra. El uso de bloques más grandes disminuye la frecuencia de fallos y mejora la eficiencia de la cache al reducir el almacenamiento de etiquetas en relación con la cantidad de almacenamiento de datos en la cache. Aunque un mayor tamaño de bloque disminuye la frecuencia de fallos, la penalización por fallo también se puede incrementar. Si la penalización por fallo aumenta linealmente con el tamaño del bloque, bloques más grandes podrían fácilmente ocasionar peores prestaciones.

Para evitar esto, el ancho de banda de la memoria principal se aumenta para transferir bloques de cache más eficientemente. Los métodos que se utilizan normalmente para incrementar el ancho de banda de la DRAM consisten en aumentar el ancho de la memoria y en utilizar el entrelazamiento. Los diseñadores de DRAM han mejorado constantemente la interfaz entre el procesador y la memoria para aumentar el ancho de banda del modo de transferencia rápido (*burst mode*) y reducir el coste de tamaños de bloque de cache más grandes.

Autoevaluación

La frecuencia del sistema de memoria influye en la decisión que debe tomar el diseñador sobre el tamaño del bloque de cache. ¿Cuáles de las siguientes directrices del diseñador de caches son generalmente válidas?

1. A medida que el tamaño del bloque de cache se hace más pequeño, la latencia de memoria disminuye.
2. A medida que el tamaño del bloque de cache se hace más grande, la latencia de memoria disminuye.
3. A medida que el tamaño del bloque de cache se hace más pequeño, el ancho de banda de la memoria es mayor.
4. A medida que el tamaño del bloque de cache se hace más grande, el ancho de banda de la memoria es mayor.

5.3 Evaluación y mejora de las prestaciones de la cache

En esta sección comenzamos describiendo cómo evaluar y analizar las prestaciones de la cache. Luego estudiaremos dos técnicas distintas para mejorar las prestaciones de la cache. Una de ellas se centra en reducir la frecuencia de fallos disminuyendo la probabilidad de que dos bloques de memoria distintos compitan por la misma posición de cache. La segunda técnica reduce la penalización por fallo añadiendo un nivel más a la jerarquía. Esta técnica, denominada *cache multinivel*, apareció por primera vez en computadores de altas prestaciones que se vendían en 1990 por unos 100 000 dólares. Desde entonces, su uso se ha extendido a computadores de sobremesa que se venden por menos de 500 dólares.

El tiempo de CPU se puede separar en los ciclos de reloj que la CPU utiliza para ejecutar el programa y en los ciclos de reloj que se requieren cuando la CPU espera a la memoria principal. Normalmente se supone que los tiempos de acceso que corresponden a aciertos forman parte de los ciclos normales de ejecución de la CPU. De este modo,

$$\begin{aligned}\text{Tiempo CPU} = (&\text{Ciclos de reloj de la ejecución de la CPU} + \\ &\text{Ciclos de reloj de parada debido a la memoria}) \times \\ &\text{Tiempo del ciclo de reloj}\end{aligned}$$

Los ciclos de reloj de parada debido a la memoria se deben principalmente a los fallos de cache, y así lo suponemos de aquí en adelante. También restringimos la exposición a un modelo simplificado del sistema de memoria. En procesadores reales, las paradas originadas por lecturas y escrituras pueden ser bastante complejas, y la predicción precisa de las prestaciones normalmente requiere simulaciones muy detalladas del procesador y del sistema de memoria.

Los ciclos de reloj de parada debido a la memoria pueden ser definidos como la suma de los ciclos de parada que se originan por las lecturas más los que se producen por las escrituras:

$$\begin{aligned}\text{Ciclos de reloj de parada debido a la memoria} = &\text{Ciclos de reloj de parada por} \\ &\text{lecturas} + \text{Ciclos de reloj de parada por escrituras}\end{aligned}$$

Los ciclos de reloj de parada por lecturas pueden ser definidos en términos del número de accesos de lectura por programa, de la penalización por fallos en ciclos de reloj para una lectura y de la frecuencia de fallos de las lecturas:

$$\text{Ciclos reloj de parada debido a lecturas} = \frac{\text{Número lecturas}}{\text{Programa}} \times \text{Frecuencia de fallos de lecturas} \times \text{Penalización por fallo lectura}$$

Las escrituras son más complicadas. Para la técnica de escritura directa, tenemos dos fuentes que originan paradas: los fallos de escritura, los cuales normalmente requieren que se busque el bloque antes de continuar la escritura (véase la Extensión de la página 467 para conocer más detalles sobre el manejo de las escrituras), y las paradas debidas al búfer de escritura, las cuales se producen cuando el búfer de escritura está lleno y aparece una nueva escritura. De este modo, los ciclos de parada para las escrituras se calculan con la siguiente expresión:

$$\text{Ciclos de bloqueo por escritura} = \left(\frac{\text{Escrituras}}{\text{Programa}} \times \text{Frecuencia fallos de escritura} \times \text{Penalización por fallo de escritura}\right) + \text{Bloqueo por búfer de escritura}$$

Debido a que las paradas del búfer de escritura dependen de la concentración de escrituras a lo largo del tiempo, y no precisamente de la frecuencia, no es posible proporcionar una ecuación sencilla que calcule tales penalizaciones. Afortunadamente, en los sistemas que disponen de un búfer de escritura de tamaño razonable (p. ej., cuatro o más palabras) y una memoria capaz de aceptar un ritmo de operaciones de escritura que exceda significativamente al ritmo promedio de ejecución de operaciones de escritura en los programas (p. ej., el doble), las penalizaciones originadas por el búfer de escritura serán pequeñas, y se pueden ignorar sin peligro alguno. Si un sistema no cumple con estos criterios, no estaría bien diseñado. En lugar de ello, el diseñador debe haber usado, o bien un búfer de escritura más grande, o bien una organización basada en escrituras retardadas.

Las técnicas de escritura retardada también pueden sufrir penalizaciones adicionales originadas por la necesidad de escribir un bloque de cache en la memoria cuando éste es reemplazado. Desarrollaremos esto en la sección 5.5.

En la mayoría de las organizaciones de cache de escritura directa, las penalizaciones por fallos de lectura y escritura son iguales (el tiempo necesario para buscar un bloque de la memoria). Si suponemos que las paradas debidas al búfer de escritura son insignificantes, podemos combinar las lecturas y escrituras utilizando una única frecuencia de fallos y la penalización por fallo:

$$\text{Ciclos de reloj de parada debidos a la memoria} = \frac{\text{Acceso a memoria}}{\text{Programa}} \times \text{Frecuencia de fallos} \times \text{Penalización por fallo}$$

Esta ecuación se puede transformar en esta otra:

$$\text{Ciclos de reloj de parada debidos a la memoria} = \frac{\text{Instrucciones}}{\text{Programa}} \times \frac{\text{Fallos}}{\text{Instrucciones}} \times \text{Penalización por fallo}$$

Consideraremos un ejemplo sencillo para ayudarnos a entender el impacto de las prestaciones de la cache sobre las prestaciones del procesador.

Cálculo de las prestaciones de la cache

EJEMPLO

Suponga que en un programa la frecuencia de fallos de la cache de instrucciones es el 2%, y la frecuencia de fallos de la cache de datos es el 4%. Si un procesador tiene un CPI igual a 2 sin incluir paradas debidas a la memoria, y la penalización por fallo es de 100 ciclos para todo tipo de fallo, determinar cuánto más rápido es un procesador cuya cache es perfecta, es decir, que nunca falla. Suponga que la frecuencia de las instrucciones de carga y almacenamiento es el 36%.

RESPUESTA

El número de ciclos por fallo de memoria debidos a los accesos a instrucciones en términos del número de instrucciones (I) es

$$\text{Ciclos debidos a fallos por accesos a instrucciones} = I \times 2\% \times 100 = 2.00 \times I$$

Como la frecuencia de instrucciones de carga y almacenamiento es el 36%, podemos calcular el número de ciclos por fallo de memoria cuando se acceden a los datos:

$$\text{Ciclos debidos a fallos por accesos a datos} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

El número total de ciclos de parada por accesos a memoria es 2.00 I + 1.44 I = 3.44 I. Este valor indica que se producen más de 3 ciclos de parada por instrucción. Por consiguiente, el CPI con paradas por accesos a memoria es 2 + 3.44 = 5.44. Ya que no se modifica el número de instrucciones o la frecuencia de reloj, la relación de tiempos de CPU es

$$\frac{\text{Tiempo CPU con parada}}{\text{Tiempo CPU con cache perfecta}} = \frac{I \times \text{CPI}_{\text{paradas}} \times \text{Ciclo de reloj}}{I \times \text{CPI}_{\text{perfecta}} \times \text{Ciclo de reloj}}$$

$$= \frac{\text{CPI}_{\text{parada}}}{\text{CPI}_{\text{perfecta}}} = \frac{5.44}{2}$$

Las prestaciones con la cache perfecta son mejores en un factor $\frac{5.44}{2} = 2.72$.

¿Qué ocurre si el procesador fuera más rápido, pero el sistema de memoria no? La cantidad de tiempo que se consume en paradas por memoria constituirá un porcentaje del tiempo de ejecución que será cada vez mayor; la ley de Amdahl, que se estudió en el capítulo 1, nos justifica este hecho. Unos pocos ejemplos sencillos muestran la gravedad de este problema. Suponga que aumentamos las prestaciones del computador del ejemplo anterior mediante la reducción de su CPI de 2 a 1 sin modificar la frecuencia de reloj, lo cual se podría conseguir mejorando el camino de datos segmentado. El sistema con fallos de cache obtendría un CPI de 1 + 3.44 = 4.44, y el sistema con cache perfecta sería

$$\frac{4.44}{1} = 4.44 \text{ veces más rápido}$$

El tiempo de ejecución que se origina en las paradas de memoria ha aumentado del

$$\frac{3.44}{5.44} = 63\%$$

al

$$\frac{3.44}{5.44} = 77\%$$

De manera semejante, el aumento de la frecuencia de reloj sin cambiar el sistema de memoria también incrementan las prestaciones que se pierden debido a los fallos de cache.

Los ejemplos y ecuaciones anteriores suponen que el tiempo de acierto no es un factor que se use para determinar las prestaciones de la cache. Obviamente, si el tiempo de acierto aumenta, el tiempo total para acceder a una palabra del sistema de memoria aumentará, causando posiblemente un aumento del tiempo de ciclo del procesador. Aunque dentro de poco veremos otros ejemplos de factores que aumentan el tiempo de acierto, un ejemplo es aumentar la capacidad de la cache. Una cache de mayor capacidad podría tener un tiempo de acceso mayor, ya que, al igual que si su mesa en la biblioteca fuera muy grande (de 3 metros cuadrados, por ejemplo), se necesitaría más tiempo para localizar uno de los libros de la mesa. Un aumento del tiempo de acierto probablemente requiere añadir otra etapa de segmentación, ya que un acierto de cache puede tardar varios ciclos. Aunque es más complejo calcular el impacto sobre las prestaciones en un procesador con un mayor número de etapas, en algún momento el aumento del tiempo de ciclo en una cache más grande podría dominar sobre la mejora en la frecuencia de aciertos, ocasionando que las prestaciones del procesador disminuyan.

Para incluir el hecho de que el tanto el tiempo de acceso a los datos en caso de acierto como en caso de fallo afecta a las prestaciones, los diseñadores utilizan a veces el *tiempo medio de acceso a memoria* (*average memory acces time*, *AMAT*) como una forma de analizar diseños alternativos de cache. El tiempo medio de acceso a memoria es el tiempo medio que se obtiene considerando tanto los fallos como los aciertos y la frecuencia de los diferentes accesos; es igual a

$$\text{AMAT} = \text{tiempo de un acierto} + \text{frecuencia de fallos} \times \text{penalización por fallo}$$

EJEMPLO

Cálculo del tiempo medio de acceso a memoria

Determinar el AMAT para un procesador con un ciclo de reloj de 1 ns, una penalización por fallo de 20 ciclos, una frecuencia de fallos de 0.05 fallos por instrucción y un tiempo de acceso a cache (incluyendo detección de acierto) de 1 ciclo. Suponga que las penalizaciones por fallo de lectura y escritura son iguales e ignorar otras paradas de escritura.

El tiempo medio de acceso a memoria por instrucción es

RESPUESTA

AMAT = tiempo de un acierto + frecuencia de fallos × penalización por fallo
= 1 0.005 × 20
= 2 ciclos

o 2 ns.

La siguiente subsección describe organizaciones alternativas de cache que disminuyen la frecuencia de fallos, pero a veces pueden aumentar el tiempo de acierto; otros ejemplos aparecen en las Falacias y errores habituales de la sección 5.11.

Reducción de los fallos de cache mediante un emplazamiento más flexible de los bloques

Hasta ahora hemos utilizado un método sencillo para emplazar los bloques: un bloque puede alojarse exactamente en un solo lugar de la cache. Como se mencionó previamente, se denomina *correspondencia directa* porque existe una asignación entre la dirección de memoria de un bloque y una única posición del nivel superior de la jerarquía. Existe realmente una amplia variedad de técnicas para realizar el emplazamiento de bloques. En un extremo está la correspondencia directa, donde un bloque puede ser emplazado en una única posición.

En el otro extremo se encuentra un método en el que un bloque puede ser emplazado en cualquier posición de la cache. Tal método se denomina **completamente asociativo** porque un bloque de memoria puede estar asociado a cualquiera de las entradas de la cache. Para encontrar un bloque determinado en una cache completamente asociativa, todas las entradas de la cache deben ser inspeccionadas, ya que un bloque puede encontrarse en cualquiera de ellas. Para que la búsqueda sea factible, se realiza en paralelo a través del comparador asociado a cada entrada de la cache. Estos comparadores incrementan significativamente el coste del hardware, haciendo que el emplazamiento completamente asociativo sea factible sólo para caches con un pequeño número de bloques.

Cache completamente asociativa: estructura de cache en la cual un bloque puede ser emplazado en cualquier posición de la cache.

La variedad de diseños que se encuentra entre la de correspondencia directa y la completamente asociativa se denomina **asociativa por conjuntos**. En una cache asociativa por conjuntos, existe un número establecido de posiciones donde cada bloque puede ser emplazado. Una cache asociativa por conjuntos con *n* posiciones posibles para un bloque se denomina cache *asociativa por conjuntos de n vías*. Una cache asociativa por conjuntos de *n* vías está formada por un número de conjuntos, cada uno de los cuales está constituido por *n* bloques. Cada bloque de la memoria se corresponde con un único *conjunto* de la cache que viene dado por el campo índice, y un bloque puede ser emplazado en cualquiera de los elementos de ese conjunto. De este modo, un emplazamiento aso-

Cache asociativa por conjuntos: cache que tiene un número establecido de posiciones (al menos dos) donde cada bloque puede ser emplazado.

ciativo por conjuntos combina emplazamiento de correspondencia directa con emplazamiento completamente asociativo: un bloque se corresponde con un conjunto, y todos los bloques del conjunto son inspeccionados para encontrar una posible coincidencia. Por ejemplo, la figura 5.13 muestra donde se emplazaría el bloque 12 en una cache con 8 bloques, según cada una de las tres estrategias de emplazamiento.

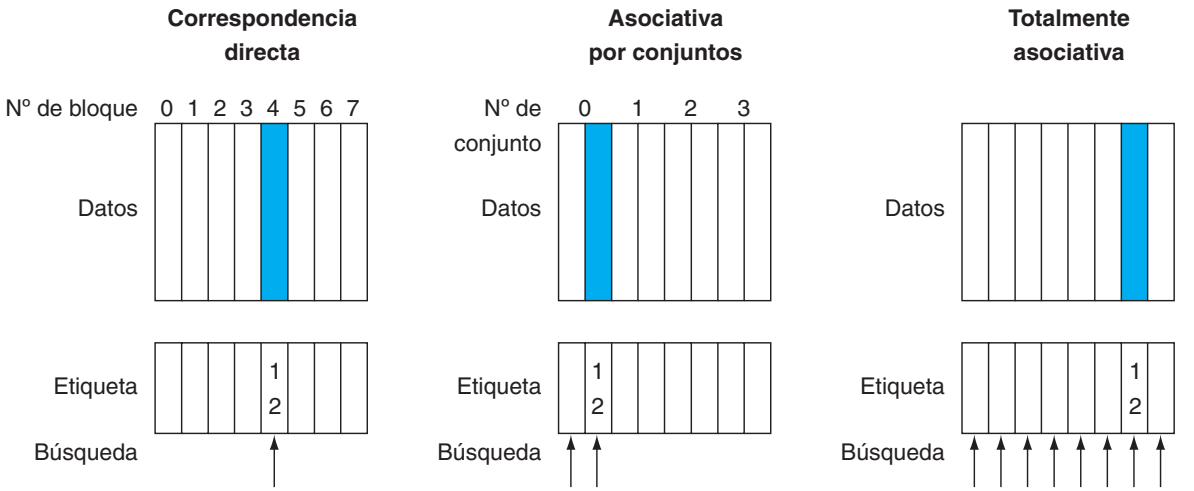


FIGURA 5.13 La posición de un bloque de memoria cuya dirección es 12 en una cache con 8 bloques varía según sea el emplazamiento: de correspondencia directa, asociativo por conjuntos o completamente asociativo. En el emplazamiento de correspondencia directa sólo existe un bloque de cache donde el bloque 12 de memoria puede ser encontrado, y ese bloque viene dado por (12 módulo 8) = 4. En una cache asociativa por conjuntos de dos vías, hay cuatro conjuntos, y el bloque 12 de memoria debe estar en el conjunto (12 mod 4) = 0; el bloque de memoria podría estar en cualquier elemento de ese conjunto. En un emplazamiento completamente asociativo, el bloque de memoria para la dirección 12 de bloque puede aparecer en cualquiera de los ocho bloques de cache.

Recuerde que en una cache de correspondencia directa la posición de un bloque de memoria viene dada por

(Número de bloque) módulo (Número de *bloques* de cache)

En una cache asociativa por conjuntos, el conjunto que contiene a un bloque de memoria viene dado por

(Número de bloque) módulo (Número de *conjuntos* de la cache)

Ya que el bloque puede ser emplazado en cualquier elemento del conjunto, *todas las etiquetas de todos los elementos del conjunto* deben ser inspeccionadas. En una cache completamente asociativa, el bloque puede estar en cualquier parte, y se deben analizar *todas las etiquetas de todos los bloques de la cache*.

Podemos considerar que cada estrategia de emplazamiento de bloques es una variación de la asociatividad por conjuntos. La figura 5.14 muestra las posibles estructuras asociativas para una cache de ocho bloques. Una cache de correspondencia directa es simplemente una cache asociativa por conjuntos de una vía: cada entrada de la cache guarda un bloque y cada conjunto tiene un único elemento. Una cache completamente asociativa con *m* entradas es simplemente una cache asociativa por conjuntos de *m* vías; tiene un único conjunto de *m* elementos, y un bloque puede alojarse en cualquiera de los elementos de ese único conjunto.

La ventaja de aumentar el grado de asociatividad consiste en que normalmente se produce una disminución de la frecuencia de fallos, como se muestra con el siguiente ejemplo. La principal desventaja, que discutiremos posteriormente con más detalle, consiste en que se incrementa el tiempo de acierto.

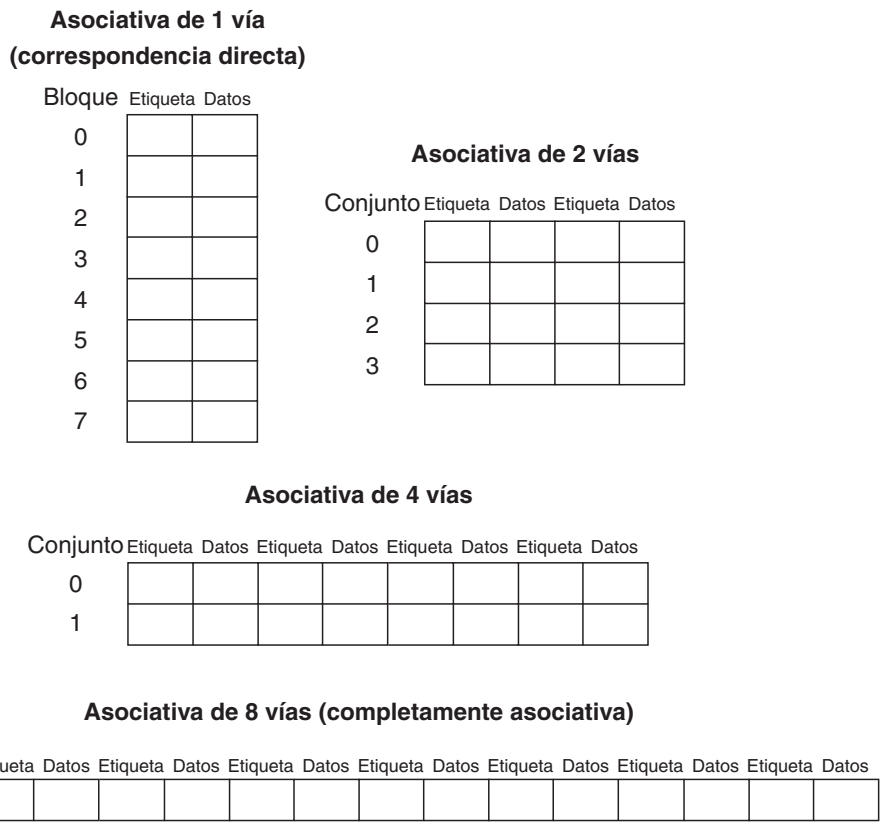


FIGURA 5.14 Una cache de ocho bloques configurada bien en correspondencia directa, asociativa por conjuntos de dos vías, asociativa por conjuntos de cuatro vías, o completamente asociativa. La capacidad total de la cache en bloques es igual al número de conjuntos multiplicado por la asociatividad. De esta forma, dado un tamaño de cache, el aumento de la asociatividad disminuye el número de conjuntos, mientras se aumenta el número de elementos en cada conjunto. Con ocho bloques, una cache asociativa por conjuntos de ocho vías es igual que una cache completamente asociativa.

EJEMPLO

Fallos y asociatividad en las caches

Suponga que se existen tres pequeñas caches, cada una de ellas formada por cuatro bloques de una palabra. La primera de las caches es completamente asociativa, la segunda es asociativa por conjuntos de dos vías, y la tercera es de correspondencia directa. Encuentre el número de fallos en cada organización de cache para la siguiente secuencia de direcciones de bloque: 0, 8, 0, 6, 8.

RESPUESTA

El caso de la cache de correspondencia directa es el más sencillo. Primero, determinemos a qué bloque de cache se corresponde cada dirección de bloque:

Dirección de bloque	Bloque de cache
0	(0 módulo 4) = 0
6	(6 módulo 4) = 2
8	(8 módulo 4) = 0

Ahora se puede completar el contenido de la cache después de que cada acceso se haya realizado. Utilizaremos el convenio que indica que una entrada en blanco significa que el bloque no es válido; si el texto está coloreado, indica que una nueva entrada de cache que se ha asociado a un acceso se ha actualizado; y si el texto está en negro, indica que la entrada de la cache es antigua.

Dirección del bloque de memoria accedido	Acierto o fallo	Contenido de los bloque de cache después de cada acceso			
		0	1	2	3
0	fallo	Memorial[0]			
8	fallo	Memorial[8]			
0	fallo	Memorial[0]			
6	fallo	Memorial[0]		Memorial[6]	
8	fallo	Memorial[8]		Memorial[6]	

La cache de correspondencia directa ocasiona cinco fallos en los cinco accesos.

La cache asociativa por conjuntos tiene dos conjuntos (con índice 0 y 1) con dos elementos por conjunto. Sepamos primero a qué conjunto corresponde cada dirección de bloque:

Dirección de bloque	Conjunto de la cache
0	(0 módulo 2) = 0
6	(6 módulo 2) = 0
8	(8 módulo 2) = 0

Como tenemos que elegir el elemento del conjunto que se va a sustituir cuando se produce un fallo, necesitamos establecer una política de reemplazos. Las caches asociativas por conjuntos normalmente reemplazan el bloque del conjunto que se ha usado menos recientemente; esto es, se reemplaza el bloque que fue utilizado por última vez hace más tiempo. (Seguidamente discutiremos las políticas de reemplazos con más detalle). Utilizando esta política para los reemplazos, el contenido de la cache asociativa por conjuntos después de cada acceso es el siguiente:

Dirección del bloque de memoria accedido	Acierto o fallo	Contenido de los bloque de cache después de cada acceso			
		Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	fallo	Memorial[0]			
8	fallo	Memorial[0]	Memorial[8]		
0	acierto	Memorial[0]	Memorial[8]		
6	fallo	Memorial[0]	Memorial[6]		
8	fallo	Memorial[8]	Memorial[6]		

Observe que cuando el bloque 6 es accedido, reemplaza al bloque 8, ya que el bloque 8 ha sido accedido menos recientemente que el bloque 0. La cache asociativa por conjuntos de dos vías experimenta cuatro fallos, uno menos que la cache de correspondencia directa.

La cache completamente asociativa tiene cuatro bloques de cache (en un solo conjunto); cualquier bloque de memoria puede ser almacenado en cualquier bloque de cache. La cache completamente asociativa tiene las mejores prestaciones, con sólo tres fallos:

Dirección del bloque de memoria accedido	Acierto o fallo	Contenido de los bloque de cache después de cada acceso			
		Bloque 0	Bloque 1	Bloque 2	Bloque 3
0	fallo	Memorial[0]			
8	fallo	Memorial[0]	Memorial[8]		
0	acierto	Memorial[0]	Memorial[8]		
6	fallo	Memorial[0]	Memorial[8]	Memorial[6]	
8	acierto	Memorial[0]	Memorial[8]	Memorial[6]	

Para este conjunto de accesos, tres fallos es lo menos que podemos conseguir porque se accede a tres direcciones distintas de bloques. Observe que si tuviéramos ocho bloques en la cache, no habría reemplazos en la cache asociativa por conjuntos de dos vías (compruebe esto usted mismo), y se experimentaría el mismo número de fallos que con la cache completamente asociativa. De manera parecida, si tuviéramos 16 bloques, las tres caches experimentarían el mismo número de fallos. Este cambio en frecuencia de fallos nos indica que la capacidad y la asociatividad de la cache influyen sobre las prestaciones de la cache.

¿Qué parte de la reducción de la frecuencia de fallos se debe a la asociatividad? La figura 5.15 muestra las mejoras obtenidas en una cache de datos de 64 KB, bloques de 16 palabras y una asociatividad que va desde la correspondencia directa hasta las ocho vías. Pasando de una asociatividad de una vía a dos vías, la frecuencia de fallos disminuye en un 15% aproximadamente, pero existe un poco más de mejora cuando se pasa a una mayor asociatividad.

Localización de un bloque en la cache

Consideremos ahora la tarea de encontrar un bloque en una cache asociativa por conjuntos. Igual que en una cache de correspondencia directa, cada bloque de una cache asociativa por conjuntos incluye una etiqueta de dirección que viene determinada por la dirección del bloque. La etiqueta de cada bloque de cache dentro del conjunto apropiado es inspeccionada para comprobar si se corresponde con la dirección del bloque que envía el procesador. La figura 5.16 muestra cómo se descompone la dirección. El valor del índice se usa para seleccionar el conjunto que contiene la dirección de interés, y las etiquetas de todos los bloques del conjunto son inspeccionadas. Como el tiempo de ejecución es primordial, todas las etiquetas de todos los bloques del conjunto se inspeccionan en paralelo. Al igual que en una cache completamente asociativa, una búsqueda secuencial ocasionaría que el tiempo de acierto de una cache asociativa por conjuntos fuera muy largo.

Si la capacidad total de la cache no cambia, el aumento de la asociatividad aumenta el número de bloques por conjunto, el cual coincide con el número de comparaciones simultáneas que son necesarias para realizar la búsqueda en paralelo: cada incremento de la asociatividad en un factor dos, dobla el número de bloques por conjunto y reduce a la mitad el número de conjuntos. Por consiguiente, cada incremento de la asociatividad en un factor dos, disminuye el tamaño del índice en 1 bit e incrementa el tamaño de la etiqueta en 1 bit. En una cache completamente asociativa sólo existe efectivamente un conjunto, y todos los bloques deben ser comprobados en paralelo. De este modo, no existe índice, y la dirección entera,

Asociatividad	Frecuencia de fallos de datos
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURA 5.15 Frecuencias de fallos de la cache de datos organizada como la del procesador Fast-MATH de Intrinsity y probada con programas SPEC2000 con asociatividades que varían desde una vía a ocho vías. Los resultados de los 10 programas SPEC200 fueron extraídos del libro Hennessy y Patterson [2003].

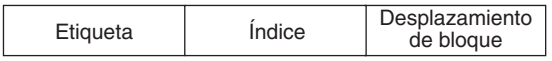


FIGURA 5.16 Las tres partes de una dirección de una cache asociativa por conjuntos o de correspondencia directa. El índice se usa para seleccionar el conjunto, y la etiqueta se usa para elegir el bloque después de compararla con las etiquetas de los bloques del conjunto seleccionado. El desplazamiento del bloque es la dirección deseada de los datos dentro del bloque.

excluyendo el desplazamiento del bloque, es comparada con la etiqueta de cada bloque. En otras palabras, se inspecciona toda la cache sin que exista indexación.

En una cache de correspondencia directa, se necesita un solo comparador, debido a que el dato solicitado puede encontrarse en un único bloque de la cache, al cual se accede indexándolo. La figura 5.17 muestra que en una cache asociativa por conjuntos de cuatro vías, se necesitan cuatro comparadores, además de un multiplexor 4 a 1 que realiza la elección entre los cuatro componentes del conjunto seleccionado. El acceso a la cache se realiza indexando el conjunto apropiado y después inspeccionando sus etiquetas. Los costes adicionales de una cache asociativa corresponden a los comparadores que se han añadido y a los retardos temporales impuestos por la necesidad de realizar la comparación y seleccionar entre los elementos del conjunto.

La elección entre correspondencia directa, asociativa por conjuntos o completamente asociativa en cualquier jerarquía de memoria dependerá del resultado de sopesar entre el coste asociado a un fallo y el coste de la implementación de la asociatividad, tanto en tiempo con en hardware adicional.

Extensión: Una *memoria direccionable por contenido (content addressable memory, CAM)* es un circuito que combina comparaciones y almacenamientos en un único dispositivo. En lugar enviar una dirección y leer la palabra almacenada en esa posición, como ocurre en las RAM, se envían el dato y la CAM comprueba si tiene una copia y devuelve el índice la fila que contiene la copia. Con las CAM, el diseñador de caches puede abordar la implementación de conjuntos de asociatividad mucho mayores que teniendo que construir el hardware a partir de SRAM y comparadores. En 2008, el mayor tamaño y consumo de potencia de las CAM, ha llevado a que generalmente los conjuntos de asociatividad de 2 y 4 vías se construyan a partir de SRAM y comparadores, y que los conjuntos de 8 o más vías se construyan con CAM.

Elección del bloque a reemplazar

Cuando se produce un fallo en una cache de correspondencia directa, el bloque solicitado sólo puede guardarse en una única posición, y por ello, el bloque que ocupaba esa posición debe ser reemplazado. En una cache asociativa tenemos la posibilidad de elegir dónde guardar el bloque solicitado, ya que podemos elegir qué bloque sustituir. En una cache asociativa por conjuntos, se puede elegir entre los bloques del conjunto seleccionado.

La política que normalmente se utiliza es la de **menos recientemente usado** (LRU), que es la que utilizamos en el ejemplo anterior. En una política LRU, el bloque reemplazado es el que no ha sido utilizado por el periodo de tiempo más largo. El ejemplo de conjunto de asociatividad de la página 482 utiliza la estrategia LRU, motivo por el que se reemplaza Memoria(0) en lugar de Memoria(6).

Menos recientemente usado (*Least recently used*, LRU): política de reemplazamientos en la cual el bloque reemplazado corresponde al que no ha sido usado por el periodo de tiempo más largo.

La implementación de reemplazos LRU consiste en realizar el seguimiento de cuándo se utiliza cada elemento del conjunto en relación con el resto de elementos de ese mismo conjunto. Para una cache asociativa por conjuntos de dos vías, podemos saber cuándo fueron utilizados los dos elementos del conjunto manteniendo un único bit en cada conjunto que indique el elemento referenciado en cada acceso. A medida que la asociatividad aumenta, la implementación de la LRU se complica. En la sección 5.5 describiremos una política de reemplazos alternativa.

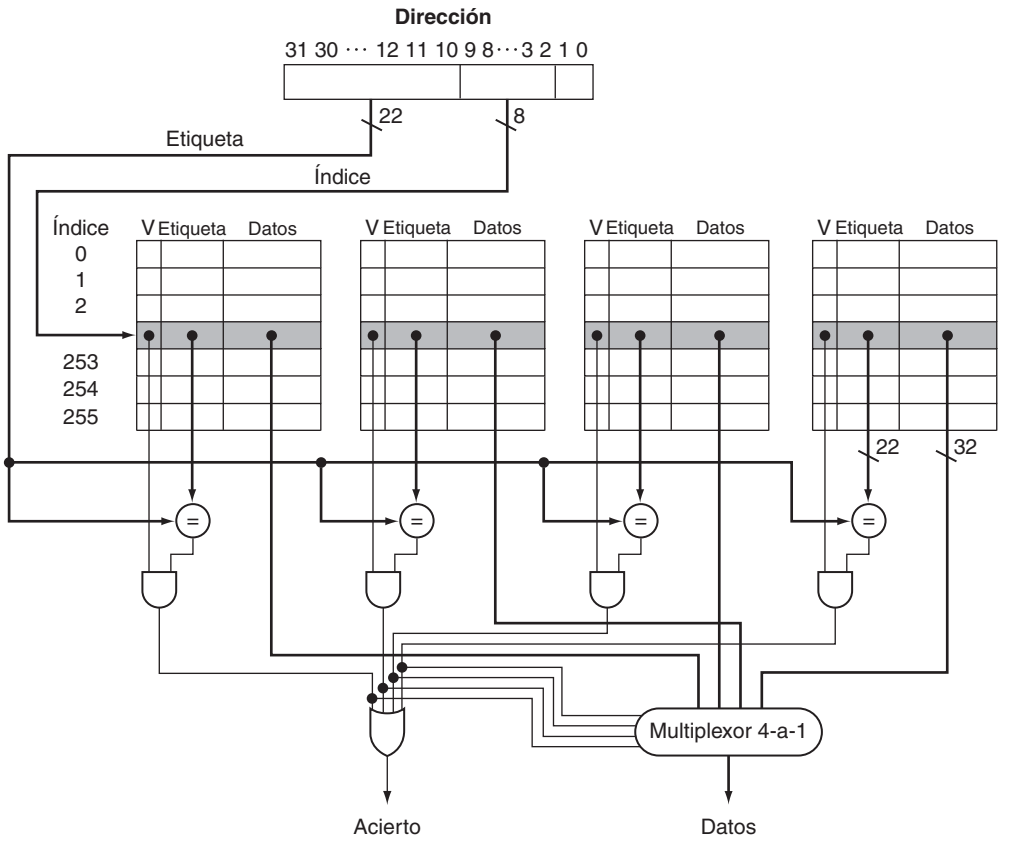


FIGURA 5.17 La implementación de una cache asociativa por conjuntos de cuatro vías requiere cuatro comparadores y un multiplexor 4 a 1. Los comparadores determinan qué elemento del conjunto seleccionado (si existe) tiene la misma etiqueta. La salida de los comparadores se utiliza para seleccionar el dato entre los cuatro bloques del conjunto indexado, haciendo uso de un multiplexor y de una señal de selección que se descodifica. En algunas implementaciones, las señales de habilitación de los datos de salida de las RAMs que constituyen la cache pueden ser usadas para seleccionar el elemento del conjunto y dirigirlo hacia la salida. La señal de habilitación de la salida se genera en los comparadores, ocasionando que el elemento seleccionado dirija el dato hacia la salida. Esta organización hace innecesaria la existencia del multiplexor.

EJEMPLO

Tamaño de las Etiquetas respecto la asociatividad por conjuntos

Incrementar la asociatividad significa que se necesitan más comparadores y más bits de etiqueta por bloque de cache. Suponiendo una cache de 4K bloques, un tamaño de bloque de cuatro palabras y una dirección de 32 bits, obtenga el número total de conjuntos y la cantidad de bits utilizados en las etiquetas de las caches de correspondencia directa, asociativas por conjunto de dos y cuatro vías, y completamente asociativa.

RESPUESTA

Ya que existen 16 ($= 2^4$) bytes en cada bloque, una dirección de 32 bits tiene asociada 32 – 4 = 28 bits para índice y etiqueta. La cache de correspondencia directa tiene el mismo número de conjuntos que de bloques, y de ahí que 12 bits se destinen al índice, ya que $\log_2(4K) = 12$. Por lo tanto, la cantidad total de bits dedicados a las etiquetas es (28 – 12) × 4K = 16 × 4K = 64 Kbits.

Cada nivel de asociatividad disminuye el número de conjuntos en un factor dos, y de este modo se disminuye en uno el número de bits dedicados al índice de la cache y se incrementa en uno el número de bits destinados a la etiqueta. En consecuencia, para una cache asociativa por conjuntos de dos vías existen 2K conjuntos, y el número de bits de etiquetas es (28 –11) × 2 × 2K = 34 × 2K = 68 Kbits. Para una cache asociativa por conjuntos de cuatro vías, el número total de conjuntos es 1K, y el número total de bits de etiquetas es (28 – 10) × 4 × 1K = 72 × 1K = 72 Kbits.

Para una cache completamente asociativa existe un solo conjunto con 4K bloques, y la etiqueta es de 28 bits, requiriéndose para las etiquetas un total de 28 × 4K × 1 = 112 Kbits.

Reducción de la penalización por fallo utilizando caches multinivel

Todos los computadores modernos hacen uso de caches. Para reducir aún más el hueco existente entre la alta frecuencia de reloj de los procesadores modernos y el largo tiempo que se requiere para acceder a las DRAMs, muchos microprocesadores se apoyan en un nivel adicional de cache. Este segundo nivel de cache se encuentra en el mismo chip y se accede cuando se produce un fallo en la cache principal. Si el segundo nivel de cache contiene el dato deseado, la penalización por fallo en el primer nivel de la cache será el tiempo de acceso al segundo nivel de la cache, que es mucho menor que el tiempo de acceso a la memoria principal. Si los datos no se encuentran ni en la cache principal ni en la secundaria, se realiza un acceso a la memoria principal, y se sufre una penalización más larga por fallo.

¿Cuál es la mejora de las prestaciones cuando se utiliza una cache secundaria? El próximo ejemplo nos lo muestra.

EJEMPLO

Prestaciones de las cache multinivel

Suponga que tenemos un procesador con un CPI base de 1.0, e imagine que todas las referencias aciertan en la cache principal y que la frecuencia de reloj es de 4 GHz. Tome un tiempo de acceso a la memoria principal de 100 ns, incluido todo el manejo de fallos. Suponga que la frecuencia de fallos por instrucción en la cache principal es del 2%. ¿Cuánto más rápido será el procesador si añadiéramos una cache secundaria que tiene un tiempo de acceso de 5 ns tanto para un acierto como para un fallo, y es lo suficientemente largo como para reducir la frecuencia de fallos de la memoria principal a 0.5%?

RESPUESTA

La penalización por fallo cuando se requiere acceder a la memoria principal es

$$\frac{100 \text{ ns}}{0.25 \frac{\text{ns}}{\text{ciclo de reloj}}} = 400 \text{ ciclos de reloj}$$

El CPI efectivo con un único nivel de cache viene dado por

CPI Total = CPI Base + Ciclos de parada por instrucción debido a memoria

Para el procesador con un nivel de cache,

CPI Total = 1.0 + Ciclos de parada por instrucción debido a la memoria = 1.0 + 2% × 400 = 9

Con dos niveles de cache, un fallo en la cache principal puede ser resuelto bien por la cache secundaria o por la memoria principal. La penalización por fallo para un acceso al segundo nivel de la cache es

$$\frac{5 \text{ ns}}{0.25 \frac{\text{ns}}{\text{ciclo de reloj}}} = 20 \text{ ciclos de reloj}$$

Si el fallo se resuelve en la cache secundaria, entonces ésta es toda la penalización por fallo. Si el fallo requiere que se acceda a la memoria principal, entonces la penalización total por fallo es la suma del tiempo de acceso a la cache secundaria y el tiempo de acceso a la memoria principal.

De este modo, para una cache de dos niveles, el CPI total es la suma de los ciclos de parada originados en ambos niveles de cache y el CPI base:

CPI Total = 1 + Ciclos de parada por instrucción en la cache principal
+ Ciclos de parada por instrucción en la cache secundaria
= 1 + 2% × 20 + 0.5% × 400 = 1 + 0.4 + 2.0 = 3.4

De este modo, el procesador con la cache secundaria es más rápido en un factor

$$\frac{9.0}{3.4} = 2.6$$

Alternativamente, podríamos haber calculado los ciclos de parada sumando los ciclos de paradas de aquellas referencias que aciertan en la cache secundaria ((2% – 0.5%) × 20 = 0.3) más los originados por los accesos a la memoria principal, los cuales deben incluir el tiempo de acceso a la cache secundaria así como el tiempo de acceso a la memoria principal (0.5% × (20 + 400) = 2.1). La suma, 1.0 + 0.3 + 2.1, de nuevo resulta ser 3.4.

Las decisiones de diseño que se toman para una cache primaria y una cache secundaria son significativamente diferentes, porque la presencia de la otra cache cambia la mejor elección respecto a una cache con un único nivel. En particular, una estructura de dos niveles permite que la cache principal se oriente a minimizar el tiempo de acierto para conseguir un ciclo de reloj lo más pequeño posible, mientras que la cache secundaria se orienta a reducir su frecuencia de fallos para reducir la penalización por fallo debida a los largos tiempos que se requieren para acceder a la memoria principal.

Comprender las prestaciones de los programas

Los métodos de ordenación se han analizado exhaustivamente para encontrar mejores algoritmos: Bubble Sort, Quicksort, Radix Sort, etc. La figura 5.18(a) muestra las instrucciones ejecutadas por elemento buscado tanto en Radix Sort como en Quicksort. De hecho, para matrices grandes, Radix Sort aventaja algorítmicamente a Quicksort en cuanto al número de operaciones. La figura 5.18(b) muestra el tiempo por clave en vez de instrucciones ejecutadas. Se observa que las líneas comienzan con la misma trayectoria que se refleja en la figura 5.18(a), pero en un momento determinado la línea de Radix Sort cambia de tendencia a medida que aumenta el volumen de datos a clasificar. ¿Qué ocurre? La figura 5.18(c) contesta a esta pregunta analizando los fallos de la cache por elemento clasificado: Quicksort tiene sistemáticamente menos cantidad de fallos por elemento que tiene que ser clasificado.

Desafortunadamente, el análisis algorítmico convencional ignora el impacto de la jerarquía de memoria. A medida que las frecuencias de reloj y la ley de Moore permiten a los arquitectos aprovechar la ejecución de instrucciones para aumentar las prestaciones, la utilización apropiada de la jerarquía de memoria es crítica para alcanzar altas prestaciones. Como expresamos en la introducción, la comprensión del comportamiento de la jerarquía de memoria es crítica para comprender las prestaciones de los programas en los computadores de hoy en día.

El efecto de estos cambios sobre los dos niveles de cache puede hacerse visible cuando se comparan con el diseño óptimo de un único nivel de cache. Cuando se compara con un único nivel de cache, la cache principal de una **cache multinivel** es a menudo más pequeña. Además, la cache principal utiliza a menudo un tamaño más pequeño de bloque, para acompañar a la reducción tanto del tamaño de cache como de la penalización por fallo. En comparación, la cache secundaria será a menudo de mayor capacidad que la cache de un solo nivel, ya que el tiempo de acceso de la cache secundaria es menos crítico. Con una capacidad total mayor, la cache secundaria a menudo usará un tamaño de bloque más grande que el apropiado para una cache de nivel único. Dado que el objetivo es reducir la frecuencia de fallos, a menudo utiliza una asociatividad mayor que la cache primaria.

Cache multinivel: jerarquía de memoria donde existen varios niveles de cache, en lugar de una sola y la memoria principal.

Extensión: Las caches multinivel originan varias complicaciones. En primer lugar, se producen distintos tipos de fallos con sus correspondientes frecuencias de fallos. En el ejemplo de la página 482 analizamos la frecuencia de fallos de la cache principal y la frecuencia de fallos global: el porcentaje de referencias a memoria que fallaron en todos los niveles de cache. Existe también una frecuencia de fallos para la cache secundaria, que se obtiene como el cociente entre todos los fallos en la cache secundaria y el número de sus accesos. Esta frecuencia de fallos se denomina frecuencia de fallos local de la cache secundaria. Puesto que la cache principal filtra los accesos, especialmente aquellos con buena localidad tanto espacial como temporal, la frecuencia de fallos local de la cache secundaria es mucho más alta que la frecuencia de fallos global. Para el ejemplo de la página 482, podemos calcular la frecuencia de fallos local de la cache secundaria como: 0.5% / 2% = 25%! Afortunadamente, la frecuencia de fallos global impone con qué frecuencia debemos acceder a la memoria principal.

Frecuencia de fallos global: fracción de accesos que fallan en todos los niveles de cache.

Frecuencia de fallos local: fracción de accesos a un nivel de cache que fallan; utilizada en caches multinivel.

Extensión: Con procesadores fuera de orden (véase capítulo 4), el análisis de las prestaciones es más complejo, ya que se ejecutan instrucciones durante el tiempo de

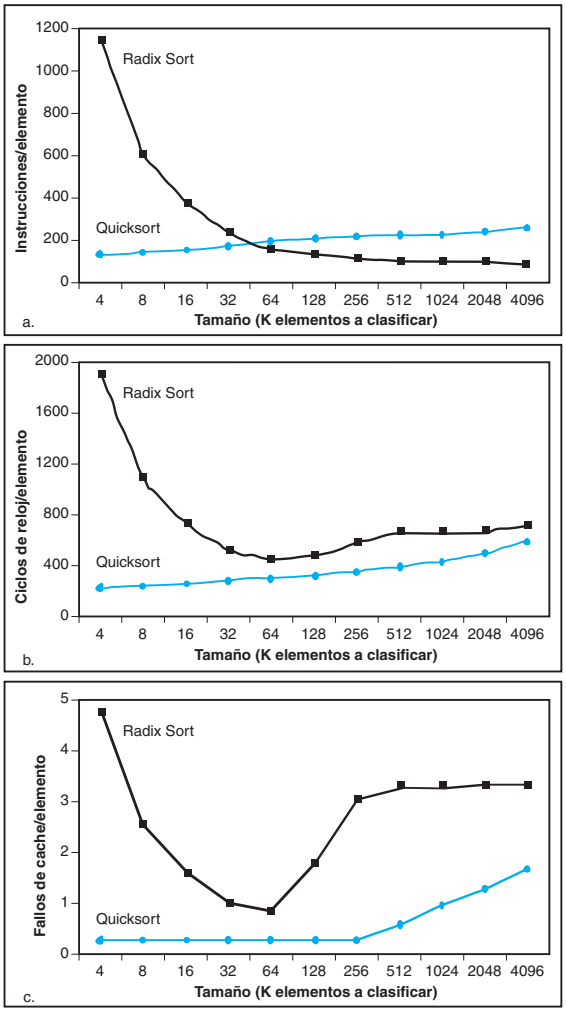


FIGURA 5.18 Comparación de Quicksort y Radix Sort utilizando (a) las instrucciones ejecutadas por elemento clasificado, (b) el tiempo por elemento clasificado, y (c) los fallos de cache por elemento clasificado. Estos datos fueron obtenidos del artículo de LaMarca y Ladner [1996]. Aunque los números pueden cambiar si se utilizan computadores más modernos, la idea sigue siendo válida. A causa de estos resultados, se inventaron nuevas versiones de Radix Sort que tienen en cuenta la jerarquía de memoria, con el objetivo de recuperar sus ventajas algorítmicas (véase la sección 5.11). La idea básica de las optimizaciones de la cache consiste en usar repetidamente los datos de un bloque antes de reemplazarlo cuando se produzca un fallo.

penalización de un fallo. En vez de frecuencia de fallos de instrucciones y frecuencia de fallo de datos, utilizamos fallos por instrucción, además de esta fórmula:

$$\frac{\text{Ciclos de parada debido a la memoria}}{\text{Instrucción}} =$$

$$\frac{\text{Fallos}}{\text{Instrucción}} \times \text{Latencia total por fallo} - \text{Latencia solapada por fallo}$$

No existe una manera generalizada de calcular la latencia solapada por fallo, por lo que el análisis de las jerarquías de memoria en procesadores fuera de orden requiere inevitablemente que el procesador y la jerarquía de memoria se simulen. Solamente observando la ejecución del procesador durante cada fallo podremos ver si el procesador se para esperando a que lleguen los datos o simplemente encuentra otra tarea que realizar. Frecuentemente, el procesador puede solapar la penalización ocasionada por un fallo en la cache L1 que acierta en la cache L2, pero rara vez solapa un fallo ocasionado en la cache L2.

Extensión: El reto de las prestaciones de los algoritmos consiste en que, según las distintas implementaciones de la misma arquitectura, la jerarquía de memoria varía en términos de capacidad, asociatividad y tamaño de bloque de la cache, así como del número de caches. Para acomodarse a esta variabilidad, algunas bibliotecas numéricas de reciente creación parametrizan sus algoritmos con el objetivo de realizar durante el tiempo de ejecución una sintonización de parámetros para aplicar la mejor combinación a un determinado computador. Esta alternativa se llama *autosintonización*.

Autoevaluación

¿Cuál de las siguientes afirmaciones sobre el diseño con varios niveles de caches es generalmente cierta?

1. A las caches de primer nivel les afectan más el tiempo de acierto, y a las caches de segundo nivel les afecta más la frecuencia de fallos.
2. A las caches de primer nivel les afectan más la frecuencia de fallos, y a las caches de segundo nivel les afecta más el tiempo de acierto.

Resumen

En esta sección nos hemos concentrado en tres temas: las prestaciones de las caches, la utilización de la asociatividad para reducir frecuencias de fallos y el uso de jerarquías multinivel de caches para reducir las penalizaciones por fallo.

El sistema de memoria afecta de forma significativa al tiempo de ejecución de los programas. El número de ciclos de parada debido a la memoria depende tanto de la frecuencia de fallos como de la penalización por fallo. El reto, como veremos en la sección 5.5, consiste en reducir uno de estos factores sin afectar significativamente a otros factores críticos de la jerarquía de memoria.

Para reducir la frecuencia de fallos, examinamos el uso de las políticas de emplazamiento asociativas. Tales políticas pueden reducir la frecuencia de fallos de una cache mediante la mayor flexibilidad que se permite al emplazamiento de bloques dentro de la cache. Las políticas completamente asociativas permiten emplazar los bloques en cualquier lugar, pero también requieren que cada bloque de la cache sea inspeccionado para resolver una solicitud de acceso a memoria. Los altos costes hacen impracticables las caches completamente asociativas de gran tamaño. Las caches asociativas por conjuntos son alternativas más factibles, ya que sólo necesitamos inspeccionar entre los elementos de un único conjunto que es seleccionado por indexación. Las caches asociativas por conjuntos tienen frecuencias de fallos más elevadas, pero son más rápidas de acceder. La cantidad de asociatividad que proporciona las mejores prestaciones depende tanto de la tecnología como de los detalles de la implementación.

Finalmente examinamos las caches multinivel como una técnica que reduce la penalización por fallo al permitir una cache secundaria más grande que maneja los fallos de la cache principal. Las caches de segundo nivel se han convertido en elementos habituales a medida que los diseñadores encuentran que las restricciones en área de

silicio, así como los objetivos marcados por las elevadas frecuencias de reloj, impiden que las caches principales puedan ser más grandes. La cache secundaria, que a menudo es 10 o más veces mayor que la cache principal, resuelve muchos accesos que fallan en la cache principal. En tales casos, la penalización por fallo viene dada por el tiempo de acceso a la cache secundaria (normalmente < 10 ciclos del procesador), en comparación con el tiempo de acceso a la memoria principal (normalmente > 100 ciclos del procesador). Como se hizo con la asociatividad, las compensaciones establecidas en el diseño entre el tamaño de la cache secundaria y su tiempo de acceso dependen de un conjunto de aspectos asociados a la implementación.

. . . se ha ideado un sistema para hacer que la combinación formada por la memoria principal y el tambor aparezca al programador como un único nivel de almacenamiento, en el que las transferencias solicitadas tienen lugar de forma automática.

Kilburn et al., "Sistema de almacenamiento de un nivel" 1962

5.4 Memoria virtual

En la sección anterior vimos cómo las caches proporcionaban rápidos accesos a los datos y al código de un programa que fueron utilizados recientemente. De un modo parecido, la memoria principal puede funcionar como una "cache" para el almacenamiento secundario, normalmente implementado con discos magnéticos. Esta técnica se denomina **memoria virtual**. Históricamente, existen dos motivaciones importantes que justifican la existencia de la memoria virtual: permitir que la memoria sea compartida de forma eficiente y segura por varios programas, y eliminar los inconvenientes de programación asociados a una memoria principal cuya capacidad sea pequeña y limitada. Cuatro décadas después de su invención, la justificación que reina hoy en día es la primera.

Memoria virtual: técnica que utiliza la memoria principal como una "cache" para el almacenamiento secundario.

Considere un conjunto de programas que se ejecutan a la vez en un computador. Evidentemente, para permitir que varios programas compartan la memoria principal, debemos ser capaces de proteger los programas de manera que no interfieran entre sí, asegurando que un programa sólo pueda leer y escribir en las porciones de memoria principal que se le han asignado. La memoria principal sólo necesita almacenar las partes activas de varios programas, del mismo modo que la cache almacena la parte activa de un solo programa. Así, el principio de localidad permite la existencia de la memoria virtual además de las caches, y la memoria virtual nos permite compartir eficientemente el procesador y la memoria principal.

No podemos saber qué programas compartirán la memoria cuando los compilemos. De hecho, los programas que comparten la memoria cambian dinámicamente mientras se están ejecutando. Debido a esta interacción dinámica, nos gustaría compilar cada programa en su propio *espacio de direcciones:* conjunto diferenciado de posiciones de memoria que sólo son accesibles por un programa. La memoria virtual implementa la traducción del espacio de direcciones de un programa a **direcciones físicas**. Este proceso de traducción impone la **protección** del espacio de direccionamiento de un programa frente al de los otros programas.

Dirección física: una de las direcciones de la memoria principal.

Protección: conjunto de mecanismos utilizados para asegurar que varios procesos que comparten el procesador, la memoria o los dispositivos de entrada/salida no interfieren entre sí realizando lecturas o escritura mutuas, de forma tanto intencionada como desintencionada. Estos mecanismos también aíslan al sistema operativo de los procesos de usuario.

La segunda motivación de la memoria virtual consiste en permitir a un único programa de usuario excederse en la capacidad de la memoria principal que utiliza. Antiguamente, si un programa era demasiado grande para la memoria, se le dejaba al programador que realizara el ajuste. Los programadores dividían los programas en trozos e identificaban los trozos que se excluían mutuamente. Estos trozos de pro-

grama eran guardados o desalojados bajo la supervisión del programa de usuario durante su ejecución, y era el programador quien aseguraba que el programa nunca intentaba acceder a una parte del programa que no estuviera alojada en memoria y que los trozos almacenados en memoria nunca excedieran la capacidad total de la memoria. Tradicionalmente, los trozos de programa que se encontraban en memoria eran organizados como módulos que contenían tanto código como datos. Las llamadas que se realizaban entre procedimientos asignados a módulos diferentes ocasionaría la coexistencia en memoria de un módulo y otro.

Es fácil imaginarse que esta responsabilidad constituía un inconveniente importante para los programadores. La memoria virtual, que fue inventada para que los programadores se despreocuparan de este problema, gestiona automáticamente los dos niveles de la jerarquía de memoria representados por la memoria principal (a veces denominada *memoria física* para distinguirla de la memoria virtual) y el almacenamiento secundario.

Aunque los fundamentos del funcionamiento de la memoria virtual y de las caches son los mismos, sus diferentes raíces históricas nos han llevado a usar una terminología distinta. A un bloque de memoria virtual se le denomina *página*, y a un fallo de memoria virtual se le denomina **fallo de página**. Con la memoria virtual, el procesador genera una **dirección virtual**, la cual se convierte mediante una combinación de hardware y software en una *dirección física*, la cual a su vez puede ser utilizada para acceder a la memoria principal. La figura 5.19 muestra la memoria direccionada de forma virtual con páginas asignadas a la memoria principal. Este proceso se denomina *conversión de direcciones* o **traducción de direcciones**. Hoy en día, los dos niveles de la jerarquía de memoria controlados por la memoria virtual corresponden a las DRAMs y a los discos magnéticos (véase el capítulo 1, páginas 22-23). Si volvemos a nuestra analogía de la biblioteca, podemos pensar que una dirección virtual se corresponde con el título de un libro; y que una dirección física se corresponde con su ubicación en la biblioteca donde se encuentra el libro, que le fue asignada según el código de identificación de la biblioteca.

Fallo de página: suceso que ocurre cuando se quiere acceder a una página que no se encuentra en la memoria principal.

Dirección virtual: dirección que corresponde a una posición en el espacio virtual y es convertida mediante una regla de asignación de direcciones en una dirección física cuando se usa para acceder a la memoria principal.

Traducción de direcciones (conversión de direcciones): proceso por el cual una dirección virtual se convierte en una dirección que se utiliza para acceder a la memoria principal.

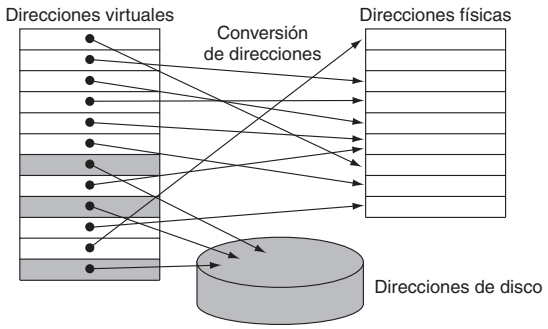


FIGURA 5.19 En la memoria virtual, los bloques de memoria (denominados páginas) son referenciados bien por un conjunto de direcciones denominadas virtuales o por otro conjunto denominado direcciones físicas. El procesador genera direcciones virtuales mientras que a la memoria se accede utilizando direcciones físicas. Tanto la memoria virtual como la memoria física se dividen en páginas, de forma tal que una página virtual se corresponde con una página física. Obviamente, también es posible que una página virtual se encuentre ausente de la memoria principal y no se le haya asignado una dirección física, por lo que se encuentra en el disco. Las páginas físicas pueden ser compartidas a través del uso de dos direcciones virtuales que apuntan a la misma dirección física. Esta posibilidad se utiliza para permitir que dos programas distintos compartan los mismos datos y código.

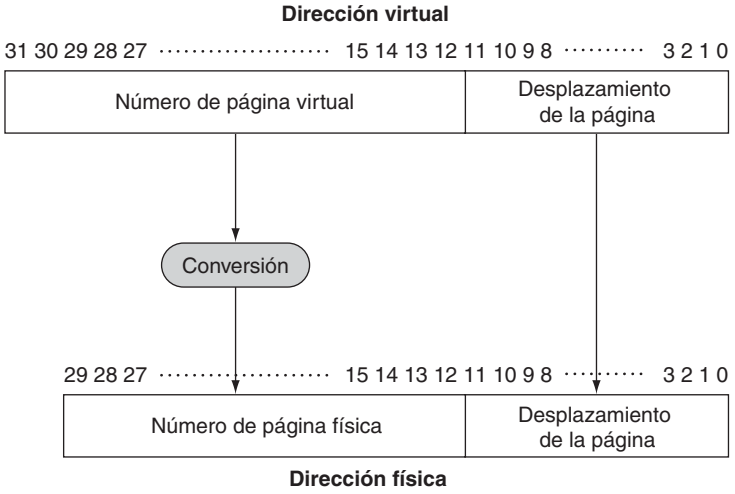


FIGURA 5.20 Conversión de una dirección virtual en una física. El tamaño de la página es de $2^{12} = 4$ KB. El número máximo de páginas físicas es 2^{18}, ya que el número de la página física es un código de 18 bits. De este modo, la memoria principal puede tener como máximo 1 GB, mientras que el espacio de direcciones virtuales es de 4 GB.

Mediante mecanismos de *reubicación*, la memoria virtual también facilita el emplazamiento del programa en la memoria principal para su posterior ejecución. Estos mecanismos de reubicación asocian las direcciones virtuales utilizadas por un programa a distintas direcciones físicas antes de que las direcciones sean utilizadas para acceder a la memoria principal. Esta reubicación nos permite almacenar el programa en cualquier parte de la memoria principal. Además, todos los sistemas de memoria virtual que actualmente están en uso reubican el programa considerándolo como un conjunto de bloques de tamaño fijo (páginas), por lo que se elimina la necesidad de encontrar un bloque contiguo de memoria para asignárselo a un programa. En vez de eso, el sistema operativo sólo necesita encontrar un número suficiente de páginas en la memoria principal.

En la memoria virtual, la dirección se divide en un *número de la página virtual* y en un *desplazamiento de la página*. La figura 5.20 muestra la transformación del número de la página virtual a un *número de la página física*. El número de la página física forma la parte más significativa de la dirección física, mientras que el desplazamiento de la página, el cual no cambia, constituye la parte menos significativa. El número de bits del campo de la dirección correspondiente al desplazamiento de la página determina el tamaño de la página. No es necesario que el número de páginas direccionables con la dirección virtual coincida con el número de páginas direccionables con la dirección física. La disponibilidad de un número mayor de páginas virtuales que de páginas física es la base de la impresión de que se dispone de una cantidad ilimitada de memoria virtual.

Muchas de las alternativas que aparecen en el diseño de los sistemas de memoria virtual están motivadas por el alto coste de un fallo, que en el ámbito de la memoria virtual se denomina tradicionalmente *fallo de página*. Un fallo de página tardará en resolverse millones de ciclos de reloj. (La tabla de la página 453 muestra que la memoria principal es alrededor de 100 000 veces más rápida que el disco). Esta

enorme penalización por fallo, dominada para tamaños de página típicos por el tiempo en recibir la primera palabra, nos lleva a tomar varias decisiones que son clave para el diseño de los sistemas de memoria virtual:

- Las páginas deben ser lo suficientemente grandes para intentar amortizar el largo tiempo de acceso. Los tamaños que típicamente se encuentran hoy en día van desde los 4 KB hasta los 16 KB. Se están desarrollando nuevos sistemas de sobremesa y servidores que soportan páginas de 32 KB y 64 KB, pero en los nuevos sistemas empotrados la tendencia es opuesta, hacia páginas de 1 KB.
- Las organizaciones que reducen la frecuencia de los fallos de página son atractivas. La principal técnica utilizada aquí consiste en permitir el emplazamiento totalmente asociativo de páginas en la memoria.
- Los fallos de página pueden ser manejados en el software debido a que el retardo temporal adicional será pequeño en comparación con el tiempo de acceso al disco. Además, el software puede permitirse usar algoritmos más inteligentes para elegir cómo emplazar las páginas, porque incluso pequeñas reducciones en la frecuencia de fallos compensarán el coste de tales algoritmos.
- La escritura directa no funcionará para la memoria virtual, ya que las escrituras tardan demasiado. En su lugar, los sistemas de memoria virtual utilizan escrituras retrasadas.

Las siguientes subsecciones dirigen la atención hacia estos elementos del diseño de la memoria virtual.

Extensión: Aunque normalmente pensemos en una dirección virtual como una dirección mayor que la dirección física, lo contrario puede ocurrir cuando el tamaño de la dirección del procesador es pequeño en relación con las posibilidades que proporciona la tecnología de las memorias. Ningún programa individual puede beneficiarse, pero una colección de programas que se están ejecutando a la vez sí puede hacerlo, bien al no tener que ser desalojados de memoria, bien al ser ejecutados en procesadores paralelos. Los procesadores de 32 bits ya empiezan a ser problemáticos para los servidores y para los computadores de sobremesa.

Extensión: La descripción de la memoria virtual que aparece en este libro se centra en la paginación, la cual utiliza bloques de tamaño fijo. Existe también una técnica denominada segmentación que utiliza bloques de tamaño variable. En la segmentación, una dirección se divide en dos partes: un número de segmento y un desplazamiento de segmento. El registro de segmento se corresponde con una dirección física, y el desplazamiento se añade para obtener la dirección física real. Como el segmento puede variar en tamaño, también se necesita realizar una comprobación de límites para asegurar que el desplazamiento se encuentra dentro del segmento. El uso más destacado de la segmentación se produce en la aplicación de métodos más potentes de protección y en la compartición del espacio de direcciones. La mayoría de los libros de texto sobre sistemas operativos contienen extensas descripciones de la segmentación en comparación con la paginación y del uso de la segmentación para compartir el espacio lógico de direcciones. La desventaja más importante de la segmentación consiste en que divide el espacio lógico de direcciones en trozos diferenciados que deben ser manipulados con direcciones de dos partes: el número de segmento y el desplazamiento. Por el contrario, la paginación hace que la frontera entre el número de página y el desplazamiento sea invisible a los programadores y compiladores.

Segmentación: asignación de direcciones de tamaño variable en la cual una dirección se considera que está formada por dos partes: un número de segmento, que se asocia a una dirección física, y un desplazamiento de segmento.

Los segmentos también fueron utilizados como método para extender el espacio de direcciones sin modificar el tamaño de palabra del computador. Estos intentos no han tenido éxito debido a lo poco práctico que resulta y a las penalizaciones sobre las prestaciones inherentes a las direcciones de dos partes que los programadores y compiladores deben tener conciencia.

Muchas arquitecturas dividen el espacio de direcciones en bloques grandes de tamaño fijo que simplifican la protección entre el sistema operativo y los programas de usuario, e incrementan la eficiencia de la implementación de la paginación. Aunque estas divisiones son a menudo denominadas "segmentos", este mecanismo es mucho más simple que la segmentación con tamaño variable de bloque y no es visible a los programas de usuario. Seguidamente abordaremos esto con más detalle.

Emplazamiento de una página y la forma de encontrarla de nuevo

Debido a la increíble alta penalización de un fallo de página, los diseñadores reducen la frecuencia de fallos de página mediante la optimización del emplazamiento de las páginas. Si permitimos que una página virtual sea asignada a cualquier página física, el sistema operativo puede reemplazar cualquier página que desee cuando se produzca un fallo de página. Por ejemplo, el sistema operativo puede utilizar un algoritmo sofisticado y complejas estructuras de datos que realicen un seguimiento de la utilización de las páginas, para intentar elegir una página que no se necesite por un largo periodo de tiempo. La ventaja que se aprovecha al usar una técnica de reemplazos que sea inteligente y flexible consiste en que la frecuencia de fallos de páginas se reduce y el uso de un emplazamiento de páginas completamente asociativo se simplifica.

Tabla de páginas: tabla que contiene las conversiones de dirección virtual a física en un sistema de memoria virtual. Esta tabla se guarda en memoria y es normalmente indexada por el número de la página virtual. Cada entrada de la tabla contiene el número de la página física para esa página virtual si la página se encuentra realmente en memoria.

Como se mencionó en la sección 5.3, la dificultad cuando se utiliza el emplazamiento completamente asociativo consiste en la forma de localizar la entrada que contiene la información, ya que ésta se puede encontrar en cualquier lugar del nivel más alto de la jerarquía. Una búsqueda completa es impracticable. En los sistemas de memoria virtual, las páginas se localizan mediante una tabla que indexa la memoria principal. Esta estructura se denomina **tabla de páginas** y se guarda en memoria. Una tabla de páginas se

Interfaz hardware software

La tabla de páginas junto con el contador de programa y los registros determinan el estado de un programa. Si queremos permitir que otro programa haga uso del procesador, debemos guardar este estado. Posteriormente, después de restaurar este estado, el programa puede continuar la ejecución. A menudo nos referimos a este estado como *proceso*. El proceso se considera *activo* cuando se encuentra en posesión del procesador; en caso contrario, se considera *inactivo*. El sistema operativo puede hacer que un proceso se active mediante la inicialización del estado de un proceso, incluyendo el contador de programa, el cual iniciará la ejecución en la instrucción apuntada por el contador de programa que estaba guardado.

El espacio de direcciones de un proceso y, por lo tanto, todos los datos a los que se puede acceder en memoria, está determinado por su tabla de páginas, la cual reside en memoria. En vez de guardar la tabla de páginas entera, el sistema operativo inicia simplemente el registro de la tabla de página para apuntar a la tabla de páginas del proceso que desea activar. Cada proceso tiene su propia tabla de páginas, ya que distintos procesos utilizan las mismas direcciones virtuales. El sistema operativo es responsable de reservar la memoria física y de actualizar las tablas de páginas, de forma tal que los espacios de direcciones virtuales de los distintos procesos no colisionen. Como veremos dentro de poco, el uso de tablas de páginas separadas también protege un proceso de otro.

indexa con el número de página que forma parte de la dirección virtual con el objetivo de obtener el número de la página física. Cada programa tiene su propia tabla de páginas con la correspondencia entre el espacio de direccionamiento virtual del programa y la memoria principal. En nuestra analogía de la biblioteca, la tabla de páginas se corresponde con una asociación entre los títulos de los libros y sus ubicaciones en la biblioteca. De la misma forma que en el archivo de fichas de los libros puede haber fichas que hagan referencia a libros que se encuentran en otras bibliotecas del campus en vez de la biblioteca local, veremos que la tabla de página puede contener entradas para páginas que no están presentes en memoria. Para indicar la localización de la tabla de páginas en la memoria, el hardware incluye un registro que apunta a la posición inicial de la tabla de páginas; lo denominamos *registro de la tabla de páginas*. Suponga por ahora que la tabla de páginas se almacena en un área de memoria fija y contigua.

La figura 5.21 utiliza el registro de la tabla de páginas, la dirección virtual, y la tabla de páginas mostrada para indicar cómo el hardware puede formar una dirección física. Se usa un bit de validez en cada entrada de la tabla de páginas, tal y como se hizo para una cache. Si el bit está desactivado, la página no está presente en la memoria principal y se produce un fallo de página. Si el bit está activado, la página está en memoria y la entrada contiene el número de la página física.

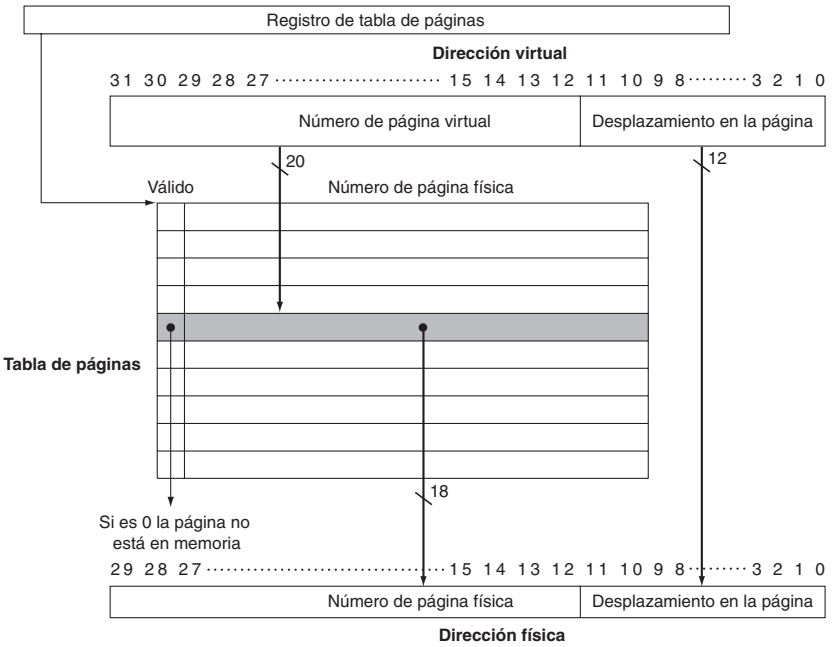


FIGURA 5.21 La tabla de páginas es indexada con el número de página virtual para obtener la correspondiente parte de la dirección física. Supondremos direcciones de 32 bits. La dirección donde comienza la tabla de páginas viene dada por el puntero de la tabla de páginas. En esta figura, el tamaño de página es 2^{12} bytes, o 4 KB. El espacio de direcciones virtuales es 2^{32} bytes, o 4 GB, y el espacio de direcciones física es 2^{30} bytes, el cual permite que la memoria principal sea de hasta 1 GB. El número de entradas en la tabla de páginas es 2^{20}, o 1 millón de entradas. El bit de validez para cada entrada indica si la asignación que contiene es válida. Si el bit está desactivado, la página no está presente en memoria. Aunque la entrada de la tabla de páginas mostrada aquí sólo requiere 19 bits de ancho, normalmente se redondea a 32 bits para facilitar su indexado. Los bits adicionales se utilizarían para almacenar información adicional que se necesita guardar de cada página, como puede ser la relativa a su protección.

Como la tabla de páginas contiene una asignación para cada una de las posibles páginas virtuales, no se requieren etiquetas. En la terminología usada para las caches, el índice que se usa para acceder a la tabla de páginas coincide con la dirección completa del bloque, la cual coincide a su vez con el número de la página virtual.

Fallos de páginas

Si el bit de validez asociado a una página virtual está desactivado, se produce un fallo de página. El sistema operativo debe tomar el control. Este cambio de contexto se realiza con el mecanismo para manejo de excepciones, el cual describiremos más tarde en esta misma sección. Una vez que el sistema operativo adquiere el control, debe encontrar la página en el siguiente nivel de la jerarquía (normalmente el disco magnético) y decidir en qué lugar de la memoria principal emplaza la página solicitada.

La dirección virtual sola no nos indica inmediatamente en qué lugar del disco se encuentra la página. Volviendo a nuestra analogía de la biblioteca, no podemos encontrar la localización de un libro de la biblioteca en las estanterías conociendo solamente su título. En lugar de ello, nos dirigimos al archivo de fichas y buscamos el libro, después de lo cual obtenemos una dirección que nos permite conocer su localización en las estanterías, tal como permite hacer el número de identificación de la Biblioteca del Congreso de EEUU. Asimismo, en un sistema de memoria virtual, debemos realizar el seguimiento de la localización en el disco de cada página del espacio de direcciones virtuales.

Debido a que no sabemos con anticipación cuándo una página de memoria será seleccionada para ser reemplazada, el sistema operativo normalmente crea el espacio en disco necesario para todas las páginas de un proceso cuando éste se crea. Este espacio en disco se denomina **zona de intercambio**. En ese momento, también se crea una estructura de datos que guarda el lugar del disco donde se almacena cada página virtual. Esta estructura de datos puede ser parte de la tabla de páginas o puede ser una estructura de datos auxiliar que es indexada de la misma forma que la tabla de páginas. La figura 5.22 muestra la organización cuando una única tabla guarda bien el número de la página física o bien la dirección en disco.

Zona de intercambio: espacio reservado en disco para el espacio completo de memoria virtual asignado a un proceso.

El sistema operativo también crea una estructura de datos que permite conocer cuáles son los procesos y las direcciones virtuales que utilizan cada página física. Cuando se produce un fallo de página, si todas las páginas de la memoria principal están siendo utilizadas, el sistema operativo debe escoger una página para reeplazarla. Debido a que se desea minimizar el número de fallos de página, la mayoría de los sistemas operativos intentan escoger una página que hipotéticamente no se necesite en un futuro próximo. Utilizando el pasado para predecir el futuro, los sistemas operativos siguen el método de reemplazar el menos recientemente utilizado (LRU), el cual mencionamos en la sección 5.3. El sistema operativo busca la página menos recientemente utilizada, haciendo la suposición que una página que no ha sido utilizada desde hace más tiempo se necesite con menos probabilidad que una página que ha sido recientemente accedida. Las páginas reemplazadas se guardan en la zona de intercambio del disco. Si le parece extraño, considere que el sistema operativo es otro proceso, y que las tablas que controlan la memoria se encuentran en memoria. Los detalles de esta aparente contradicción se explicarán dentro de poco.

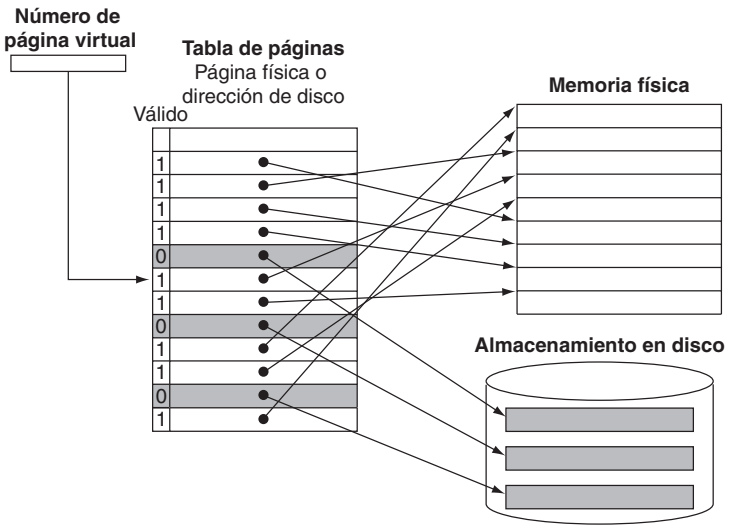


FIGURA 5.22 La tabla de páginas asigna cada página de la memoria virtual o a una página en memoria principal o a una página almacenada en el disco, que es el siguiente nivel de la jerarquía. El número de página virtual se utiliza para indexar la tabla de páginas. Si el bit de validez está activado, la tabla de páginas suministra el número de la página física (es decir, la primera dirección de la página en memoria) correspondiente a la página virtual. Si el bit de validez está desactivado, la página actual se encuentra sólo en disco, y la dirección especificada corresponde a una dirección en disco. En muchos sistemas, la tabla de direcciones de páginas físicas y de direcciones de páginas en disco se almacenan en dos estructuras de datos separadas, aunque lógicamente forman parte de la misma tabla. Las tablas dobles están justificadas en parte debido a que debemos guardar las direcciones en disco de todas las páginas, incluso si se encuentran presentes en la memoria principal. Recuerde que las páginas en memoria principal y las páginas en disco son de igual tamaño.

La implementación de un esquema LRU que sea completamente preciso es muy costosa, ya que requiere actualizar una estructura de datos en *cada* acceso a memoria. En lugar de ello, la mayoría de los sistemas operativos se aproximan a LRU haciendo un seguimiento de tanto las páginas que han sido utilizadas recientemente como las que no. Para ayudar al sistema operativo a conocer las páginas LRU, algunos computadores proporcionan un **bit de uso** o **bit de referencia**, que se activa cuando una página es accedida. El sistema operativo desactiva periódicamente los bits de referencia y al cabo de un cierto tiempo los guarda de forma que pueda determinar qué páginas fueron accedidas durante un determinado periodo de tiempo. Con esta información relacionada con el uso de las páginas, el sistema operativo puede seleccionar una página que está entre las menos recientemente utilizadas (detectada por tener desactivado su bit de referencia). Si este bit no es suministrado por el hardware, el sistema operativo debe encontrar otra forma de estimar qué páginas han sido accedidas.

Interfaz hardware software

Bit de referencia: también denominado bit de uso. Campo que es activado cuando una página es accedida y que se utiliza para implementar LRU u otras políticas de reemplazamiento.

Extensión: Con una dirección virtual de 32 bits, páginas de 4 KB, y 4 bytes por entrada de la tabla, podemos calcular el tamaño total de la tabla de páginas:

$$\text{Número de entradas de la tabla de páginas} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Tamaño tabla páginas} = 2^{20}\ \text{entradas tabla páginas} \times 2^{2}\ \frac{\text{bytes}}{\text{entradas tabla páginas}} = 4\ \text{MB}$$

Esto es, necesitaríamos usar 4 MB de memoria por cada programa en ejecución en cada instante de tiempo. Esta cantidad de memoria no está mal para un solo programa, pero ¿qué sucede si hay cientos de programas, cada uno con su propia tabla de páginas? ¿Cómo deberíamos manejar direcciones de 64 bits, que con este cálculo necesitarían 2^{52} palabras?

Para reducir la cantidad de almacenamiento que se requiere para la tabla de páginas, se utilizan varias técnicas. Las cinco que aparecen debajo se centran en la reducción del almacenamiento total máximo requerido así como en la minimización de la memoria principal dedicada a las tablas de páginas.

1. La técnica más sencilla consiste en utilizar un registro límite que restrinja el tamaño de la tabla de páginas para un proceso dado. Si el número de página virtual es mayor que lo que indica el registro límite, se añaden entradas a la tabla de páginas. Esta técnica permite que la tabla de páginas aumente a medida que un proceso vaya consumiendo más espacio de memoria. De este modo, la tabla de páginas sólo será grande si el proceso utiliza muchas páginas del espacio de direccionamiento virtual. Esta técnica requiere que el espacio de direcciones se expanda en una única dirección.

2. Permitir que el crecimiento se realice en una única dirección no es suficiente, ya que la mayoría de los lenguajes requieren dos áreas cuyos tamaños varíen independientemente: un área guarda la pila (*stack*) y la otra guarda la memoria dinámica (*heap*). Debido a esta dualidad, es conveniente dividir la tabla de páginas y permitir que crezcan desde las direcciones más altas hacia direcciones inferiores, y por otro lado, desde las más bajas hacia direcciones superiores. Esto significa que existirán dos tablas de páginas separadas y dos límites distintos. El uso de dos tablas de páginas divide el espacio de direcciones en dos segmentos. El bit más significativo de una dirección determina a qué segmento pertenece y de este modo, qué tabla de páginas usa. Ya que el segmento se determina a través del bit más significativo de la dirección, cada segmento puede ser tan grande como la mitad del espacio de direccionamiento. Un registro límite para cada segmento determina el tamaño real del segmento, el cual aumenta en unidades de páginas. Este tipo de segmentación se utiliza en muchas arquitecturas, entre ellas MIPS. A diferencia del tipo de segmentación descrita en la segunda Extensión de la página 495, este tipo de segmentación no es visible para el programa, aunque sí lo es para el sistema operativo. La desventaja más destacada de esta técnica se debe a que no funciona bien cuando el espacio de direccionamiento está muy troceado en vez de estar en una zona contigua del espacio de direccionamiento virtual.

3. Otro enfoque orientado a la reducción del tamaño de la tabla de páginas consiste en aplicar una función de hashing a la dirección virtual, de forma tal que el tamaño de la estructura de datos de la tabla de páginas requiere solamente ser igual al número de páginas que residen en la memoria principal. Esta estructura se denomina *tabla invertida de páginas*. Evidentemente, el proceso de acceso a esta tabla es ligeramente más complejo debido a que no podemos simplemente realizar el acceso a través de indexación.

4. También se pueden utilizar varios niveles de tablas para reducir la cantidad total de almacenamiento destinado a las tablas de páginas. El primer nivel asigna bloques de páginas del espacio de direccionamiento virtual que son grandes y de tamaño fijo, quizás de 64 a 256 páginas en total. A veces, a estos grandes bloques se les denomina segmentos, y a esta tabla de asignaciones de primer nivel, aunque los segmentos son invisibles para usuario. Cada entrada de la tabla de segmentos indica si alguna de las páginas de ese segmento se encuentra alojada en memoria y, si es así, apunta a su ta-

bla de páginas. La traducción de direcciones se produce primeramente a través del acceso a la tabla de segmentos, haciendo uso de los bits más significativos de la dirección. Si la dirección del segmento es válida, el siguiente conjunto de bits más significativo se utiliza para indexar la tabla de páginas indicada por la entrada de la tabla de segmentos. Este método permite que el espacio de direcciones pueda estar troceado (varios segmentos no contiguos de memoria pueden estar activos) sin tener que reservar hueco para la tabla de páginas entera. Tales métodos son particularmente útiles con espacios de direccionamiento muy grandes, así como en sistemas software que requieren una asignación de direcciones no contigua. La principal desventaja de esta asignación de dos niveles es que requiere un proceso más complejo para la traducción de direcciones.

5. Para reducir la cantidad de memoria principal destinada a las tablas de páginas, la mayoría de los sistemas modernos también permiten paginar las tablas de páginas. Aunque esto parezca que tiene truco, funciona correctamente debido al uso de las mismas ideas en las que se basa la memoria virtual. Simplemente se permite que las tablas de páginas residan en el espacio de direccionamiento virtual. Existen algunos pequeños problemas, aunque críticos, que deben ser evitados, como las series interminables de fallos de páginas. La forma de resolver estos problemas es muy minuciosa, así como altamente dependiente del procesador. En resumen, estos problemas se evitan mediante el emplazamiento de todas las tablas de páginas en el espacio de direccionamiento del sistema operativo, así como del emplazamiento de al menos algunas de las tablas de páginas del sistema en una parte de la memoria principal que se direcciona físicamente y está siempre presente en memoria principal, y por lo tanto, nunca en disco.

¿Qué ocurre con las escrituras?

La diferencia entre los tiempos de acceso a la cache y a la memoria principal es de decenas a cientos de ciclos, por lo que es posible utilizar los métodos de escritura directa, aunque necesitemos un búfer de escritura para ocultar la latencia de las instrucciones de almacenamiento. En un sistema de memoria virtual, las escrituras al siguiente nivel de la jerarquía (el disco) requieren millones de ciclos de reloj del procesador. Por lo tanto, la utilización de un búfer de escritura para permitir que el sistema escriba directamente a disco sería completamente impracticable. En su lugar,

Interfaz hardware software

Un método de escritura retardada tiene otra importante ventaja cuando se utiliza en un sistema de memoria virtual. Debido a que el tiempo de transferencia a disco es pequeño comparado con el tiempo de latencia del disco, la escritura retardada de una página entera es mucho más eficiente que la escritura retardada en disco de palabras individuales. Una operación de escritura retardada, aunque más eficiente que la transferencia individual de palabras, sigue siendo costosa. Así que, nos gustaría saber si una página *necesita o no* ser copiada de forma retardada cuando se decida reemplazarla. Para saber si una página ha sido escrita desde que fue llevada a memoria principal, se añade a la tabla de páginas un *bit de consistencia (dirty bit)* que se activa cuando cualquier palabra de la página es actualizada desde el procesador. Si el sistema operativo decide reemplazar la página, el bit de consistencia indica si ésta requiere ser actualizada en disco antes de que el espacio que ocupa en memoria sea asignado a otra página. Así, una página modificada se suele denominar *página inconsistente (dirty page)*.

los sistemas de memoria virtual deben usar escritura retardada, realizando las distintas escrituras en la página de memoria, y almacenando posteriormente la página en disco cuando sea reemplazada de la memoria.

Aumento de la rapidez con la que se realiza la traducción de direcciones: el TLB

Ya que las tablas de páginas se almacenan en memoria principal, cada acceso a memoria generado desde un programa puede tardar el doble: un acceso para obtener la dirección física y el segundo acceso para obtener los datos. La clave utilizada para mejorar las prestaciones consiste en aprovechar la localidad de los accesos a la tabla de páginas. Cuando se realiza una conversión de un número de página virtual, es probable que en un futuro próximo se necesite hacer otra vez, debido a que los accesos a las palabras de una página tienen tanto localidad temporal como espacial.

Búfer de traducción lateral (TLB): cache que guarda las conversiones recientes de direcciones para evitar acceder a la tabla de páginas.

Por consiguiente, los procesadores modernos incluyen una cache especial que guarda las conversiones recientemente usadas. Esta cache especial para conversiones de direcciones tradicionalmente se denomina **búfer de traducción lateral** (translation-lookaside búfer, TLB), aunque sería más preciso denominarla cache de conversiones. La cache TLB es equivalente a una hoja de papel que normalmente usamos para apuntar la ubicación del conjunto de libros que hemos consultado en el archivo de fichas de la biblioteca. En vez de buscar reiteradamente el archivo entero, apuntamos la ubicación de varios libros y utilizamos el pedazo de papel como una cache de los números de identificación de la biblioteca.

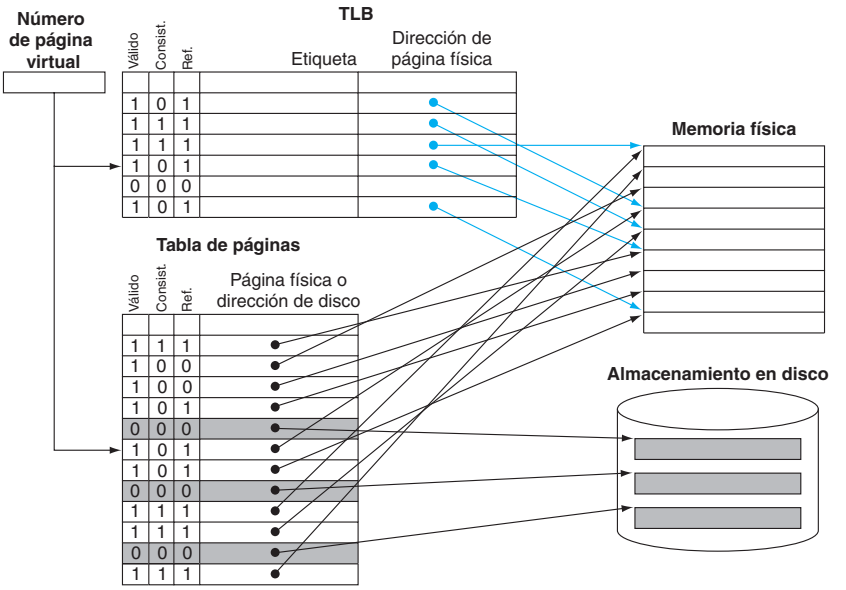


FIGURA 5.23 La TLB se comporta como una cache de la tabla de páginas para las entradas que sólo asignan páginas físicas. La cache TLB contiene un subconjunto de correspondencias entre páginas virtuales y páginas físicas que se encuentran en la tabla de páginas. Las asignaciones de la cache TLB se indican en color. Debido a que la TLB es una cache, debe disponer de un campo de etiqueta. Si no existiera una entrada en la cache TLB que se correspondiera a una página, la tabla de páginas debe ser consultada. La tabla de páginas bien suministra un número de página física (que puede ser utilizado para conformar una entrada de la cache TLB), o bien indica que la página se aloja en disco, en cuyo caso se produce un fallo de página. Ya que la tabla de páginas tiene una entrada para cada página virtual, no necesita un campo para etiquetas; en otras palabras, no es una cache.

La figura 5.23 muestra que en cada etiqueta de la cache TLB se guarda una parte del número de página virtual, y que en la parte de datos de la cache TLB se guarda un número de página física. Debido a que se evita consultar la tabla de páginas en cada acceso, ya que se consulta la cache TLB, cada entrada de esta cache TLB necesita incluir otros bits, tales como los bits de consistencia y los de referencia (o de uso).

En cada acceso a memoria, se comprueba si el número de página virtual se encuentra en la cache TLB. Si se produce un acierto, se utiliza el número de la página física para formar la dirección, y el bit de uso asociado a la correspondiente entrada se activa para indicar que ha sido accedida. Si el procesador está ejecutando una escritura, el bit de consistencia también se activa. Si se produce un fallo en la TLB, debemos determinar si es un fallo de página o simplemente un fallo de TLB. Si la página se encuentra en memoria, el fallo de TLB indica sólo que la conversión no está guardada. En tales casos, el procesador puede manejar el fallo de TLB guardando en la cache TLB la conversión que se encuentra en la tabla de páginas, y posteriormente realizar de nuevo el acceso a la cache TLB. Si la página no está presente en memoria, el fallo de TLB indica un fallo verdadero de página. En este caso, el procesador invoca al sistema operativo a través de una excepción. Debido a que la cache TLB tiene muchas menos entradas que el número de páginas en memoria principal, los fallos de TLB serán muchos más frecuentes que los fallos verdaderos de página.

Los fallos de TLB pueden gestionarse tanto mediante hardware como de software. En la práctica, si se hace con cuidado, puede haber pequeñas diferencias en las prestaciones entre los dos enfoques, ya que las operaciones básicas son las mismas en ambos casos.

Después de que se produzca un fallo de TLB y se haya recuperado la conversión ausente desde la tabla de páginas, necesitaremos seleccionar una entrada de la cache TLB para reemplazarla. Debido a que los bits de acceso y consistencia se guardan en la entrada de la cache TLB, necesitamos actualizar estos bits en la correspondiente entrada de la tabla de páginas antes de reemplazar la entrada de la cache TLB. Estos bits constituyen la única parte de la entrada de la cache TLB que puede ser modificada. El uso de las escrituras retardadas –esto es, las entradas de la tabla de páginas se actualizan durante el tiempo en que se maneja un fallo de TLB, en lugar de cuando se modifican– es muy eficiente, ya que se espera que la frecuencia de fallos de la cache TLB sea pequeña. Algunos sistemas utilizan otras técnicas para evitar la necesidad de la existencia de los bits de acceso y consistencia, eliminando así la necesidad de escribir en la cache TLB excepto para guardar una nueva entrada de la tabla cuando se produce un fallo.

Valores típicos para la cache TLB podrían ser los siguientes:

- Capacidad TLB: 16–512 entradas
- Tamaño de bloque: 1–2 entradas de la tabla de páginas (normalmente 4–8 bytes cada una)
- Tiempo de acierto: 0.5–1 ciclos de reloj
- Penalización por fallo: 10–100 ciclos de reloj
- Frecuencia de fallos: 0.01% –1%

Los diseñadores han usado en las caches TLB un amplio rango de asociatividades. Algunos sistemas utilizan caches TLB que son pequeñas y completamente asociativas debido a que una correspondencia completamente asociativa tiene una menor frecuencia de fallos. Además, ya que la cache TLB es pequeña, el coste de una correspondencia completamente asociativa no es demasiado alto. Otros sistemas utilizan grandes caches

TLB, a menudo con asociatividad baja. Con una correspondencia completamente asociativa, la elección de la entrada a reemplazar se complica, ya que la implementación hardware de un método para la LRU es demasiado costosa. Además, ya que los fallos de TLB son mucho más frecuentes que los fallos de página, y por lo tanto deben ser manejados de forma menos costosa, no nos podemos permitir el uso de un costoso algoritmo software, como el que disponemos para los fallos de páginas. Por consiguiente, muchos sistemas proporcionan recursos para elegir aleatoriamente la entrada a reemplazar. Examinaremos las políticas de reemplazos con más detalle en la sección 5.5.

La cache TLB del FastMATH de Intrinsity

Para visualizar estas ideas en un procesador real, veamos con más detalle la cache TLB del FastMATH de Intrinsity. El sistema de memoria utiliza páginas de 4 KB y un espacio de direccionamiento de 32 bits. De este modo, el número de página virtual tiene 20 bits, como se observa en la parte superior de la figura 5.24. La dirección física es del mismo tamaño que la dirección virtual. La cache TLB tiene 16 entradas, es completamente asociativa y está compartida por las referencias a instrucciones y datos. Cada entrada tiene un ancho de 64 bits y contiene una etiqueta de 20 bits (que conforman el número de página virtual de una entrada de la cache TLB), el número de página física correspondiente (también de 20 bits), un bit de validez, un bit de consistencia y otros bits para gestión interna.

La figura 5.24 muestra la cache TLB y otra de las memorias cache, mientras que en la figura 5.25 se indican los pasos a seguir en el procesamiento de una petición de lectura o de escritura. Cuando se produce un fallo de TLB, el hardware MIPS guarda el número de página del acceso en un registro especial y genera una excepción. La excepción invoca al sistema operativo, el cual maneja el fallo a través de software. Para encontrar la dirección física de la página ausente, la rutina que maneja el fallo de la cache TLB indexa la tabla de páginas usando el número de página de la dirección virtual y el registro de la tabla de páginas, el cual apunta a la dirección inicial de la tabla del proceso activo. Usando un conjunto de instrucciones especiales a las que se les permite actualizar la TLB, el sistema operativo almacena la dirección física obtenida de la tabla de páginas en la cache TLB. Un fallo de TLB tarda alrededor de 13 ciclos de reloj, suponiendo que el código y la entrada de la tabla de páginas se encuentran en las caches de instrucciones y de datos respectivamente. (Veremos el código MIPS que maneja un fallo de la cache TLB en la página 513.) Un fallo de página verdadero se produce si la entrada de la tabla de páginas no tiene una dirección física válida. El hardware mantiene un índice que indica la entrada que se recomienda reemplazar; la entrada recomendada se escoge aleatoriamente.

Existe una complicación adicional para las solicitudes de operaciones de escritura: concretamente, el bit de acceso por escritura de la cache TLB debe ser comprobado. Este bit evita que el programa escriba en páginas que sólo disponen de permiso para realizar lecturas. Si el programa intenta escribir y el bit de acceso por escritura está desactivado, se produce una excepción. El bit de acceso por escritura forma parte del mecanismo de protección, que describiremos en breve.

Integración de la memoria virtual, las TLB y las caches

Nuestra memoria virtual y los sistemas de cache funcionan conjuntamente como una jerarquía, así que los datos no pueden encontrase en la cache a no ser que se encuentren presentes en la memoria principal. El sistema operativo juega un papel importante en el

mantenimiento de esta jerarquía eliminando de la cache el contenido de cualquier página cuando decide llevar esa página a disco. Al mismo tiempo, el sistema operativo modifica las tablas de páginas y la cache TLB, de forma que cualquier intento de acceder a un dato de la página generará un fallo de página.

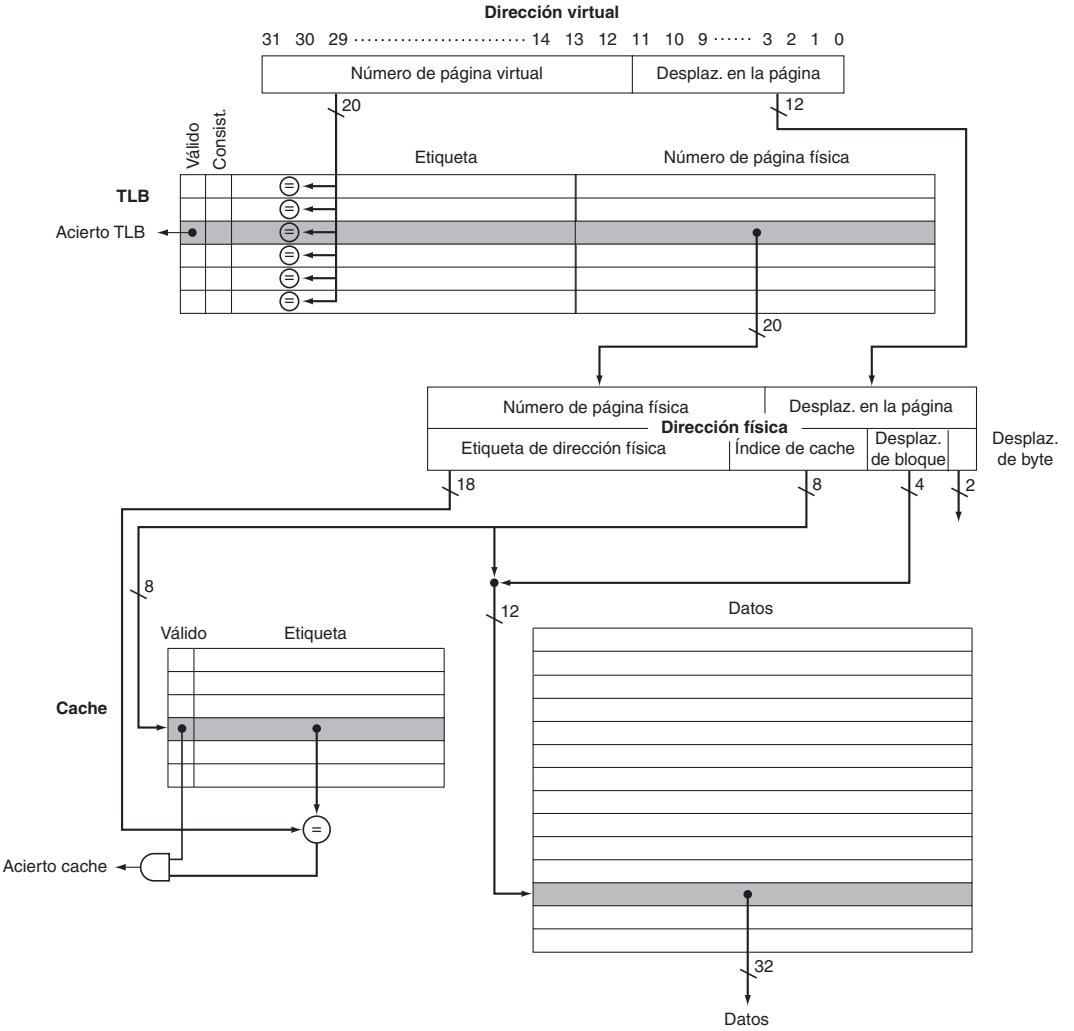


FIGURA 5.24 La TLB y la cache implementan el proceso de asociar una dirección virtual a un dato en el FastMATH de Intrinsity. Esta figura muestra la organización de la cache TLB y de la cache de datos suponiendo un tamaño de página de 4 KB. Este diagrama se orienta hacia una operación de lectura. La figura 5.25 describe cómo se manejan las escrituras. Observe que a diferencia de la figura 5.9, las RAMs de etiquetas y datos están separadas. Direccionando una RAM de datos grande pero estrecha con el índice de la cache concatenado con el desplazamiento de bloque, seleccionamos la palabra deseada dentro del bloque sin la necesidad de un multiplexor 16:1. Mientras la cache de datos es de correspondencia directa, la cache TLB es completamente asociativa. La implementación de una cache TLB completamente asociativa requiere que cada etiqueta de la cache TLB sea comparada con el número de página virtual, ya que la entrada solicitada puede encontrarse en cualquier parte de la cache TLB. (Véase la descripción de las memorias direccionables por contenido en la Extensión de la página 485.) Si el bit de validez de la entrada correspondiente está activado, el acceso es un acierto en la cache TLB, y los bits del número de la dirección de la página física junto con los bits del desplazamiento de página conforman el índice que es utilizado para acceder a la cache de datos.

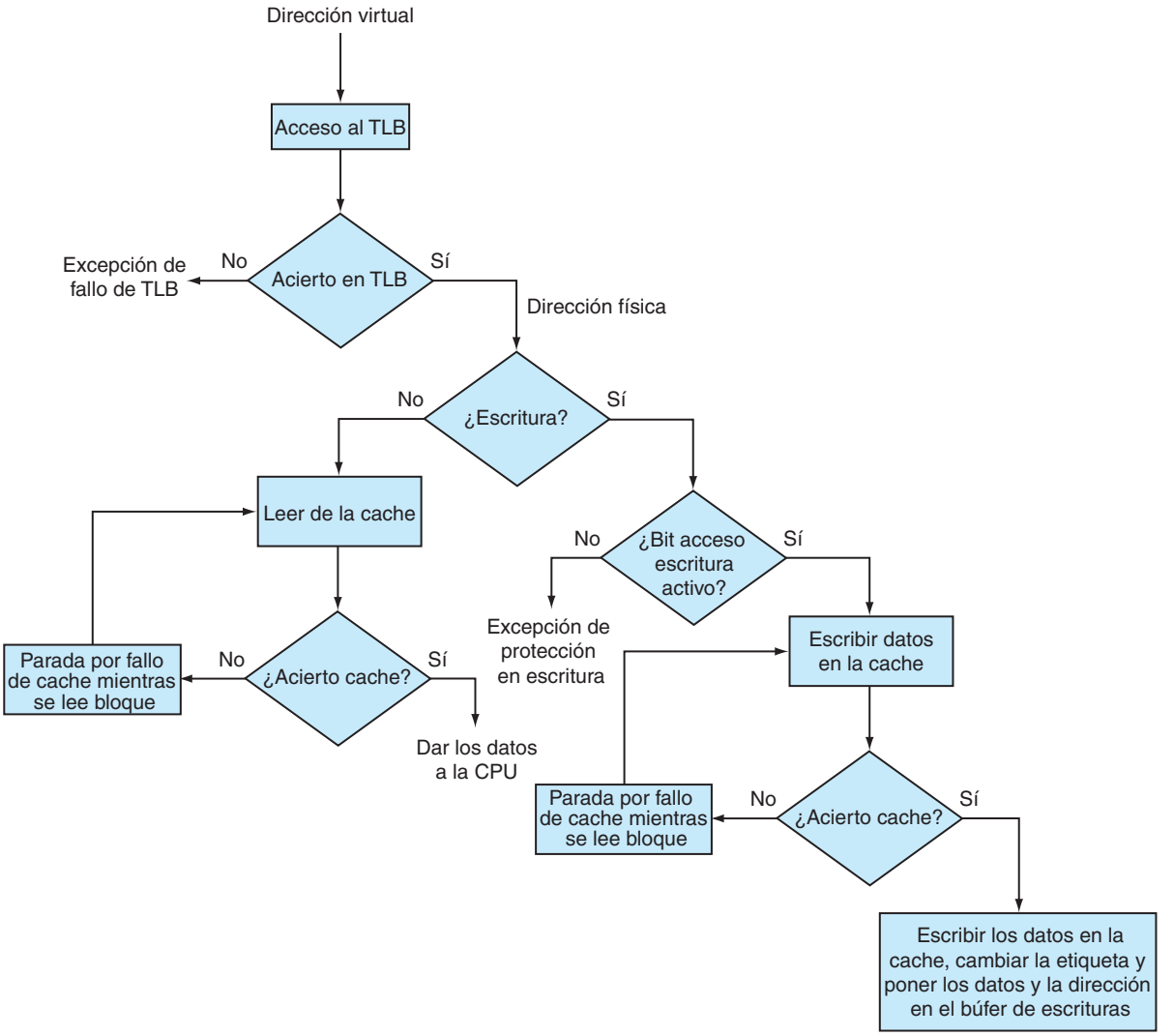


FIGURA 5.25 Procesamiento de una operación de lectura o de escritura directa de las caches TLB y de datos del FastMATH de Intrinsity. Si la cache TLB genera un acierto, la cache de datos puede ser accedida con la correspondiente dirección física. Para una operación de lectura, la cache de datos genera un acierto o un fallo, y suministra los datos u ocasiona una parada mientras los datos son traídos desde la memoria. Si la operación es de escritura, una parte de la entrada de la cache de datos se sobrescribe en un acierto y los datos son enviados al búfer de escritura si suponemos escritura directa. Un fallo de escritura es como un fallo de lectura excepto que el bloque es modificado después de que se lea desde la memoria. Las escrituras retardadas requieren que el bit de consistencia del bloque se active para las operaciones de escritura, y en el búfer de escritura se guarda todo el bloque sólo en un fallo de lectura o un fallo de escritura si el bloque a reemplazar no es coherente. Observe que un acierto en una cache TLB y un acierto en una cache de datos son sucesos independientes, pero un acierto en la cache de datos sólo puede ocurrir después de que se produzca un acierto en la cache TLB, lo cual significa que los datos deben encontrarse en memoria. La relación entre los fallos de TLB y los fallos de la cache de datos se examina con más detalle en el siguiente ejemplo y en los ejercicios del final del capítulo.

En el mejor de los casos, la cache TLB convierte una dirección virtual y la envía a la cache, donde los datos apropiados son encontrados, recuperados y enviados al procesador. En el peor caso, un acceso puede fallar en los tres componentes de la jerarquía de memoria: la TLB, la tabla de páginas y la cache. Los siguientes ejemplos ilustran estas interacciones con mayor detalle.

Funcionamiento global de la jerarquía de memoria

EJEMPLO

En una jerarquía de memoria como la mostrada en la figura 5.24 que incluye una TLB y una cache organizada como se muestra, un acceso a memoria puede encontrarse con tres tipos distintos de fallos: un fallo de TLB, un fallo de página y un fallo de la cache de datos. Considere todas las combinaciones cuando se producen uno o más de estos sucesos (siete posibilidades). En cada caso, indique si el suceso realmente puede producirse y en qué circunstancias.

RESPUESTA

La figura 5.26 muestra los casos posibles y si en la práctica pueden producirse o no.

Cache TLB	Tabla de páginas	Cache de datos	¿Es posible? Si es así, ¿en qué circunstancias?
acierto	acierto	fallo	Posible, aunque realmente nunca se comprueba la tabla de páginas cuando se produce un acierto en la cache TLB.
fallo	acierto	acierto	Fallos de TLB, pero la entrada se encuentra en la tabla de páginas; después de reintentarlo, los datos se encuentran en la cache de datos.
fallo	acierto	fallo	Fallos de TLB, pero la entrada se encuentra en la tabla de páginas; después de reintentarlo, los datos fallan en la cache de datos.
fallo	fallo	fallo	Fallos de TLB, a los que le sigue un fallo de página; después de reintentarlo, los datos fallan en la cache de datos.
acierto	fallo	fallo	Imposible: no puede realizarse una conversión en la cache TLB si la página no está presente en la memoria.
acierto	fallo	acierto	Imposible: no puede realizarse una conversión en la cache TLB si la página no está presente en la memoria.
fallo	fallo	acierto	Imposible: no se puede permitir que los datos estén en la cache si la página no se encuentra en memoria.

FIGURA 5.26 Posibles combinaciones de sucesos ocurridos en la cache TLB, sistema de memoria virtual y cache de datos. Tres de estas combinaciones son imposibles, y una de ellas es posible (acierto en la cache TLB, acierto en la memoria virtual, fallo en la cache de datos) pero nunca se detecta su aparición.

Extensión: La figura 5.26 supone que todas las direcciones de memoria se convierten a direcciones físicas antes de acceder a la cache de datos. En esta organización, la dirección física se utiliza para indexar la cache de datos y comprobar las etiquetas (tanto el índice como la etiqueta de la cache de datos corresponden a direcciones físicas en vez de virtuales). En este sistema, el tiempo necesario para acceder a memoria, suponiendo un acierto en la cache de datos, debe incluir tanto un acceso a la cache TLB como un acceso a la cache de datos. Evidentemente, estos accesos pueden segmentarse.

Cache direccionada virtualmente: cache a la que se accede con una dirección virtual en lugar de una dirección física.

Alternativamente, el procesador puede indexar la cache de datos con una dirección que es completa o parcialmente virtual. A esto se le denomina **cache direccionada virtualmente**, y utiliza etiquetas que corresponden a las direcciones virtuales. Por lo tanto, este tipo de cache es *indexada virtualmente* y *etiquetada virtualmente*. En estas caches, el hardware para realizar la conversión de direcciones (la TLB) no se utiliza durante un acceso normal a la cache de datos, ya que se accede a ella con una dirección virtual que no ha sido convertida en una dirección física. Esto hace que la TLB se encuentre fuera del camino crítico, reduciéndose así la latencia de la cache de datos. Sin embargo, cuando se produce un fallo en la cache de datos, el procesador necesita convertir la dirección a dirección física, de forma que pueda ir a buscar el bloque de la cache de datos a la memoria principal.

Existencia de alias: situación en la cual se accede al mismo objeto desde dos direcciones. Puede ocurrir en la memoria virtual cuando existen dos direcciones virtuales para la misma página física.

Cuando se accede a la cache de datos con una dirección virtual y las páginas son compartidas por los programas (los cuales pueden acceder a ellas con diferentes direcciones virtuales), existe la posibilidad de la **existencia de alias** (*aliasing*). La aparición de alias ocurre cuando el mismo objeto tiene dos nombres; en este caso, dos direcciones virtuales para la misma página. Esta ambigüedad genera un problema porque una palabra de esa página puede encontrarse en dos posiciones de la cache de datos, correspondiéndose cada una de ellas con una dirección virtual distinta. Esta ambigüedad permitiría que un programa escribiera los datos sin que el otro programa sea consciente de que los datos habían cambiado. Las caches direccionadas completamente con direcciones virtuales o bien introducen limitaciones en el diseño de las caches y TLB para reducir la aparición de alias, o bien requieren que el sistema operativo, y posiblemente el usuario, realice los pasos necesarios para asegurar que no aparezcan alias.

Cache físicamente direccionada: cache que es direccionada por una dirección física.

Un compromiso que se utiliza frecuentemente y que se sitúa entre estos dos tipos de diseños, consiste en indexar virtualmente las caches (a veces utilizando sólo la parte de la dirección que corresponde al desplazamiento de la página, la cual realmente es una dirección física ya que no se transforma), pero se usan etiquetas físicas. Estos diseños que son *indexados virtualmente* pero *etiquetados físicamente*, tratan de conseguir las ventajas de las prestaciones propio de las caches indexadas virtualmente con las ventajas arquitectónicas de una **cache direccionada físicamente**. Por ejemplo, no existe problema de alias en este caso. La cache de datos L1 del Pentium 4 es un ejemplo, como lo sería el procesador de Intrinsity, si el tamaño de página fuera de 4 KB. Para lograr que funcione, debe elegirse con mucho cuidado el tamaño mínimo de página, el tamaño de la cache y la asociatividad.

Implementación de la protección con la memoria virtual

Una de las funciones más importantes de la memoria virtual consiste en permitir que varios procesos compartan una misma memoria principal, mientras se proporciona protección de memoria entre los distintos procesos y el sistema operativo. El mecanismo de protección debe asegurar que aunque varios procesos compartan la misma memoria principal, un proceso renegado no pueda escribir en el espacio de direccionamiento de otro proceso de usuario o en el del sistema operativo, tanto de forma intencionada como no intencionada. El bit de acceso de escritura en la TLB puede proteger a una página de que sea escrita. Sin este nivel de protección, los virus de los computadores estarían aún más extendidos.

También queremos evitar que un proceso lea datos de otro proceso. Por ejemplo, no queremos que un programa hecho por un estudiante lea las notas mientras se encuentren en la memoria del procesador. Una vez que comencemos a compartir la memoria principal, debemos proporcionar a un proceso la posibilidad de proteger sus

Interfaz hardware software

Para posibilitar que el sistema operativo implemente la protección en el sistema de memoria virtual, el hardware debe proporcionar al menos tres funciones básicas que se resumen a continuación.

1. Que soporte al menos dos modos que indiquen si el proceso en ejecución es de usuario o del sistema operativo, el cual se denomina proceso **supervisor**, proceso **núcleo** o proceso *ejecutivo*.
2. Que permita que una parte del estado del procesador pueda ser leída pero no modificada por un proceso de usuario. Esto incluye el bit de modo usuario/supervisor, que determina si el procesador está en modo usuario o supervisor, el puntero de la tabla de páginas y la TLB. Para modificar estos elementos, el sistema operativo utiliza instrucciones especiales que están sólo disponibles en el modo supervisor.
3. Que proporcione mecanismos para que el procesador pueda cambiar del modo usuario al modo supervisor, y al revés. El primero de los cambios normalmente se realiza mediante una excepción de **llamada al sistema**, y es implementado por una instrucción especial (*syscall* para el repertorio de instrucciones MIPS) que transfiere el control a una posición determinada en el espacio donde se aloja el código del supervisor. Como con cualquier otra excepción, desde el punto en que se realiza la llamada al sistema, el contador de programa se guarda en el registro de excepciones EPC, y el procesador se sitúa en modo supervisor. Para volver al modo usuario desde la excepción, se utiliza la instrucción ERET (*return from exception*), la cual reinicia el modo usuario y salta a la dirección apuntada por EPC.

Con la utilización de estos mecanismos y el almacenamiento de las tablas de páginas en el espacio de direccionamiento del sistema operativo, el sistema operativo puede modificar las tablas de páginas a la vez que evita que un proceso de usuario las modifique, y así se asegura que un proceso de usuario sólo pueda acceder al almacenamiento que le otorga el sistema operativo.

Modo núcleo (modo supervisor): modo que indica que el proceso en ejecución es un proceso del sistema operativo.

Llamada al sistema: instrucción especial que transfiere el control desde el modo usuario a una posición determinada en el espacio donde se encuentra el código del supervisor, e invoca al mecanismo de excepciones durante el proceso.

datos de lecturas y escrituras realizadas desde otros procesos. De lo contrario, ¡la compartición de la memoria principal sería una ventaja a medias!

Recuerde que cada proceso dispone de su propio espacio de direccionamiento virtual. De este modo, si el sistema operativo organiza las tablas de páginas de forma tal que las páginas virtuales independientes se corresponden con páginas físicas disjuntas, un proceso no será capaz de acceder a los datos de otros procesos. Evidentemente, esto también requiere que un proceso de usuario sea incapaz de modificar las correspondencias de las tablas de páginas. El sistema operativo puede garantizar seguridad si evita que el proceso de usuario modifique sus propias tablas de páginas. Sin embargo, el sistema operativo sí debe ser capaz de modificar las tablas de páginas. Colocando las tablas de páginas en el espacio de direccionamiento protegido del sistema operativo se satisfacen ambos requerimientos.

Cuando los procesos desean intercambiar información de una forma restringida, el sistema operativo debe ayudarlos, ya que el acceso a la información de otro pro-

ceso requiere cambiar la tabla de páginas del proceso que realiza la visita. El bit de acceso por escritura puede ser utilizado para restringir que el intercambio sólo sea para leer, y como el resto de la tabla de páginas, este bit sólo puede ser modificado por el sistema operativo. Para permitir que otro proceso, digamos P1, lea una página que pertenece al proceso P2, P2 pediría al sistema operativo que cree una entrada en la tabla de páginas de P1 para una página virtual de su espacio de direccionamiento que apunta a la misma página física que P2 quiere compartir. El sistema operativo podría usar el bit de protección de escritura para evitar que P1 escriba datos, si ese fue el deseo de P2. Cualquier bit que determine los derechos de acceso de una página debe ser incluido tanto en la tabla de páginas como en la TLB, debido a que sólo se accede a la tabla de páginas cuando se produce un *fallo* en la TLB.

Cambio de contexto: cambio del estado interno del procesador que permite a un proceso distinto utilizar el procesador, y que tiene asociado el almacenamiento de toda la información necesaria del estado para poder retornar al proceso que se está actualmente en ejecución.

Extensión: Cuando el sistema operativo decide cambiar la ejecución desde el proceso P1 al proceso P2 (denominado **cambio de contexto** o *cambio de proceso*), se debe asegurar que P2 no puede acceder a las tablas de páginas de P1 debido a que se comprometería la protección. Si no existe TLB, es suficiente cambiar el registro de la tabla de páginas para apuntar a la tabla de páginas de P2 (en vez de a la de P1). Cuando exista una TLB, debemos limpiar las entradas de la TLB que pertenecen a P1, tanto para proteger los datos de P1 como para forzar que la TLB cargue las entradas para P2. Si la frecuencia de cambio de proceso fuera alta, resultaría bastante ineficiente. Por ejemplo, P2 sólo podría cargar unas pocas entradas de la cache TLB antes de que el sistema operativo volviera a P1. Desafortunadamente, P1 encontraría que todas sus entradas en la cache TLB habrían desaparecido y tendría que pagar los fallos de TLB para recargarlos. Este problema surge debido a que las direcciones virtuales utilizadas por P1 y P2 son las mismas, y debemos limpiar la cache TLB para evitar que estas direcciones se confundan.

Una alternativa común consiste en extender el espacio de direccionamiento virtual por medio de la inclusión de un *identificador de proceso* o *identificador de tarea*. El FastMATH de Intrinsity dispone para este propósito un campo identificador del espacio de direccionamiento que consta de 8 bits. Este pequeño campo identifica el proceso que se está ejecutando actualmente, y se guarda en un registro inicializado por el sistema operativo cuando se cambia de proceso. El identificador de proceso se concatena con la parte de etiquetas de la TLB, de forma tal que un acierto de TLB se produce sólo si el número de página y el identificador de proceso coinciden. Esta combinación elimina la necesidad de limpiar la cache TLB, excepto en raras ocasiones.

En una cache pueden aparecer problemas similares, ya que en un cambio de proceso la cache contendrá datos de los procesos en ejecución. Estos problemas surgen de forma diferente en las caches direccionadas físicamente y en las direccionadas virtualmente. Para resolverlos se aplica un rango de soluciones, tales como los identificadores de procesos, con el objetivo de asegurar que un proceso toma sus propios datos.

Manejo de los fallos de la cache TLB y los fallos de página

Aunque la conversión de dirección virtual a dirección física utilizando una cache TLB es sencilla cuando se produce un acierto de TLB, el manejo de los fallos de TLB y de los fallos de páginas son más complejos. Un fallo de TLB se produce cuando ninguna entrada de la TLB se corresponde con una dirección virtual. Un fallo de TLB puede indicar una de las dos posibilidades:

1. La página se encuentra en memoria, y sólo necesitamos inicializar una entrada en la TLB.

2. La página no se encuentra en memoria, y necesitamos transferir el control al sistema operativo para que se encargar del fallo de página.

¿Cómo sabemos cuál de estas dos circunstancias se ha producido? Cuando procesamos el fallo de TLB, buscamos una entrada de la tabla de páginas para llevarla a la TLB. Si la entrada de la tabla de páginas seleccionada tiene desactivado su bit de validez, la correspondiente página no se encuentra en memoria y sufrimos un fallo de página, en lugar de solamente un fallo de TLB. Si el bit de validez está activado, simplemente cargamos en la TLB la entrada de la tabla de páginas deseada.

Un fallo de TLB puede ser manejado en software o en hardware, debido a que requerirá una pequeña secuencia de operaciones para copiar la entrada de la tabla de páginas desde memoria a la TLB. MIPS maneja tradicionalmente en software un fallo de TLB. Se trae la entrada de la tabla de páginas desde memoria y después vuelve a ejecutar la instrucción que causó el fallo de TLB. En la repetición de la ejecución se producirá un acierto de TLB. Si la entrada de la tabla de páginas indica que la página no está en memoria, esta vez se producirá una excepción por fallo de página.

El manejo de un fallo de TLB o de un fallo de página requiere la utilización del mecanismo de excepción para interrumpir al proceso activo, transfiriendo el control al sistema operativo, y posteriormente reanudando la ejecución del proceso interrumpido. Un fallo de página será reconocido a veces durante el ciclo de reloj utilizado para acceder a memoria. Para volver a comenzar la instrucción después de manejar el fallo de página, el contador de programa de la instrucción que causó el fallo de página debe guardarse. Como se describió en el capítulo 4, el contador de programas para excepciones (EPC) se utiliza para guardar este valor.

Además, una excepción por fallo de TLB o por fallo de página debe activarse al final del mismo ciclo de reloj en el que se produce el acceso a memoria, de forma tal que el procesamiento de la excepción comience en el siguiente ciclo de reloj, en vez de continuar normalmente con la ejecución de instrucciones. Si el fallo de página no fuera reconocido en este ciclo de reloj, una instrucción de carga podría sobrescribir un registro, y esto podría ser desastroso cuando intentemos volver a ejecutar la instrucción. Por ejemplo, considere la instrucción `lw $1,0($1)`: el computador debe ser capaz de evitar que la etapa de segmentación de post-escritura en el banco de registros se complete; de lo contrario, no se podría comenzar a ejecutar la instrucción adecuadamente, ya que el contenido del registro `$1` habría sido destruido. Una complicación similar surge en las operaciones de almacenamiento. Debemos evitar que las escrituras en memoria realmente se completen

Registro	Número de registro CP0	Descripción
EPC	14	Dónde volver a comenzar después de la excepción
Causa	13	Causa de la excepción
BadVAddr	8	Dirección que causó la excepción
Index	0	Posición de la cache TLB que se lee o se escribe
Random	1	Posición pseudoaleatoria de la cache TLB
EntryLo	2	Dirección de la página física y banderas
EntryHi	10	Dirección de la página virtual
Context	4	Dirección de la tabla de páginas y número de página

FIGURA 5.27 Registros de control MIPS. Estos registros se encuentran en el coprocesador 0, y por lo tanto son leídos utilizando `mfc0` y escritos utilizando `mtc0`.

cuando existe un fallo de página. Esto se hace normalmente por medio de la desactivación de la línea de control de escritura de la memoria.

Una vez que el sistema operativo conoce la dirección virtual que causó el fallo de página, debe completar tres pasos:

1. Consultar la tabla de páginas utilizando la dirección virtual y encontrar en disco la posición de la página referenciada.
2. Elegir una página física a reemplazar; si la página elegida no es coherente, debe ser actualizada en disco antes de que una nueva página sustituya a esta página física.
3. Comenzar la lectura para traer la página referencia desde disco a la página física elegida.

Evidentemente, el ultimo paso llevará millones de ciclos de reloj del procesador (ocurrirá lo mismo en el segundo paso si la página reemplazada no es coherente). Por consiguiente, el sistema operativo seleccionará normalmente otro proceso para que el procesador lo ejecute hasta que el acceso a disco se termine. Debido a que el sistema operativo ha guardado el estado del proceso, puede libremente darle el control del procesador a otro proceso.

Cuando se ha completado la lectura desde disco de la página, el sistema operativo puede restaurar el estado del proceso que originalmente causó el fallo de página, y ejecutar la instrucción que activa la vuelta desde la excepción. Esta instrucción cambia el estado del procesador desde el modo núcleo al modo usuario, y también restaura el contador de programa. Después de ese momento, el proceso de usuario vuelve a ejecutar la instrucción que causó el fallo, accede a la página solicitada, esta vez con éxito, y continúa la ejecución.

Interfaz hardware software

Habilitación de excepciones: también denominado habilitación de interrupciones. Una señal o acción que controla si el proceso reacciona a una excepción o no; necesario para evitar la aparición de excepciones durante los intervalos de tiempo que aparecen antes de que el procesador haya guardado con seguridad la información de estado que es necesaria para volver a continuar con la ejecución del programa.

Entre el instante en que comienza a ejecutarse el manejador de excepciones en el sistema operativo y el instante en que el sistema operativo ha guardado toda la información del estado del proceso, el sistema operativo es especialmente vulnerable. Por ejemplo, si se produce otra excepción cuando se está empezando a procesar la primera en el sistema operativo, la unidad de control sobrescribiría el contador de programas para las excepciones, ¡haciendo imposible que el programa pudiera volver a la instrucción que originó el fallo de página! Podríamos evitar este desastre permitiendo la posibilidad de deshabilitar o de **habilitar la aparición de excepciones**. Cuando una excepción se produce por primera vez, el procesador activa un bit que deshabilita la aparición de nuevas excepciones; esto podría ocurrir al mismo tiempo en que el procesador activa el bit del modo supervisor. El sistema operativo guardará entonces la suficiente información del estado del procesador para permitir restaurarlo si se produce otra excepción –concretamente, el contador de programa para excepciones (EPC) y el registro de causa. Los registros EPC y de causa corresponden a dos registros especiales de control que intervienen en el manejo de excepciones, los fallos de TLB y los fallos de página; la figura 5.27 muestra el resto de los registros que también intervienen. A partir de ese momento el sistema operativo puede habilitar de nuevo la aparición de excepciones. Estos pasos aseguran que las excepciones no provoquen la pérdida de cualquier información de estado en el pocesador, haciendo que éste sea incapaz de comenzar de nuevo desde la instrucción que activó la interrupción.

Las excepciones por fallo de página debidas a accesos a datos son difíciles de implementar adecuadamente en un procesador debido a una combinación de tres características:

1. Se producen en el medio de una instrucción, no como en los fallos de páginas cuando se accede a instrucciones.
2. La instrucción no puede completarse antes de terminar de procesar la excepción.
3. Después de procesar la excepción, la instrucción debe volver a ejecutarse como si nada hubiera ocurrido.

Hacer que las instrucciones sean de **ejecución repetida**, de forma tal que las excepciones puedan ser procesadas y con posterioridad la instrucción pueda continuar, es relativamente fácil en una arquitectura como MIPS. Debido a que cada instrucción escribe sólo un dato y esta escritura se produce al finalizar el ciclo de instrucción, simplemente podemos evitar que la instrucción finalice (no escribiendo en el banco de registros) y que comience a ejecutarse desde el principio.

Instrucción de ejecución repetida: instrucción que puede reanudar la ejecución después que una excepción se resuelva, sin que la excepción afecte al resultado de la instrucción.

Observemos con más detalle lo que ocurre en MIPS. Cuando se produce un fallo de TLB, el hardware del MIPS guarda el número de página del acceso en un registro especial denominado `BadVAddr` y genera una excepción.

La excepción invoca al sistema operativo, el cual maneja el fallo a través de software. El control es transferido a la dirección 8000 0000_{hex}, que es la posición inicial del **manejador** de fallos de TLB. Para encontrar la dirección física de la página ausente, la rutina de fallos de TLB indexa la tabla de páginas utilizando el número de página de la dirección virtual y el registro de la tabla de páginas, el cual apunta la dirección inicial de la tabla de página del proceso activo. Para hacer que esta indexación sea rápida, el hardware de MIPS sitúa todo lo necesario en el registro especial `Context`: los 12 bits más significativos determinan la dirección base de la tabla de páginas y los siguientes 18 bits determinan la dirección virtual de la página ausente. Cada entrada de la tabla de páginas es de una palabra, así que los últimos 2 bits son 0. De este modo, las primeras dos instrucciones copian el contenido del registro Context en el registro temporal del núcleo `$k1`, y posteriormente se copia la entrada de la tabla de páginas apuntada por esa dirección en `$k1`. Recuerde que `$k0` y `$k1` están reservados para que el sistema operativo los utilice sin que se necesite guardarlos; una razón muy importante para establecer este convenio es la de hacer que el manejador de fallos de TLB funcione rápidamente. A continuación se muestra el código MIPS para un típico manejador de fallos de TLB.

Manejador: nombre de una rutina software que es invocada para "manejar" una excepción o una interrupción.

```
TLBmiss:
  mfc0   $k1,Context    # copia la dirección de la entrada de la tabla
                          de páginas (PTE) en el registro temporal $k1
  lw     $k1, 0($k1)    # carga el contenido de la dirección PTE en $k1
  mtc0   $k1,EntryLo    # copia PTE en el registro especial EntryLo
  tlbwr                 # copia EntryLo en una entrada aleatoria de la
                          cache TLB
  eret                  # retorna de la excepción del manejo de
                          excepciones por fallo de TLB
```

Como se muestra más arriba, MIPS tiene un conjunto de instrucciones especiales del sistema para actualizar la TLB. La instrucción `tlbwr` copia el contenido del registro `EntryLo` en la entrada de la TLB seleccionada por el registro de control `Random`. `Random` es un registro que implementa un reemplazo aleatorio, así

que corresponde básicamente a un contador de barrido libre. Un fallo de TLB tarda alrededor de una docena de ciclos de reloj.

Observe que el manejador de fallos de TLB no comprueba si la entrada de la tabla de páginas es válida. Como la excepción originada por la entrada ausente de la TLB es mucho más frecuente que la originada por un fallo de página, el sistema operativo actualiza la TLB desde la tabla de páginas sin examinar la entrada y vuelve a ejecutar la instrucción. Si la entrada no es válida, se produce otra excepción que es distinta, y el sistema operativo reconoce el fallo de página. Este método procesa rápidamente el caso frecuente de fallo de TLB, con una pequeña penalización para el caso infrecuente del fallo de página.

Una vez que el proceso que generó el fallo de página ha sido interrumpido, el control se transfiere a $8000\ 0180_{hex}$, una dirección distinta que la del manejador de fallo de TLB. Esta es una dirección para manejo de excepciones en general; el fallo de TLB tiene un punto especial de entrada para reducir la penalización por un fallo de TLB. El sistema operativo utiliza el registro de excepción Cause para diagnosticar la causa de la excepción. Debido a que la excepción corresponde a un fallo de página, el sistema operativo sabe que el procesamiento requerido es considerable. De este modo, a diferencia de un fallo de TLB, si guarda todo el estado del proceso activo. Este estado incluye todos los registros de propósito general y de punto flotante, el registro con la dirección de la tabla de páginas, el EPC y el registro que indica la causa de la excepción (Cause). Ya que los manejadores de excepción no utilizan normalmente los registros de punto flotante, el punto de entrada general a la rutina de manejo de excepciones no los salva, dejando esa operación a los pocos manejadores que lo requieren.

La figura 5.28 esboza el código MIPS de un manejador de excepciones. Observe en el código MIPS que salvamos y restauramos el estado, teniendo cuidado cuando habilitamos o deshabilitamos las excepciones, e invocamos un código C para manejar una excepción en particular.

La dirección virtual que causó el fallo depende de si el fallo fue debido a fallo de instrucción o de dato. La dirección de la instrucción que originó el fallo se encuentra en el registro EPC. Si fue un fallo de página por instrucción, el registro EPC contiene la dirección virtual de la página que causó el fallo; de lo contrario, la dirección virtual que causa el fallo puede ser obtenida examinando la instrucción (cuya dirección está en EPC) para encontrar el registro base y el desplazamiento.

Extensión: Esta versión simplificada supone que el puntero de pila (sp) es válido. Para evitar el problema de un fallo de página durante la ejecución de este código de excepción de bajo nivel, MIPS reserva una parte de su espacio de direccionamiento que no puede sufrir fallos de páginas, denominada no asignada. El sistema operativo guarda en memoria no asignada el código del punto de entrada de la excepción y la pila de excepciones. El hardware de MIPS convierte las direcciones virtuales que van desde $8000\ 0000_{hex}$ a $BFFF\ FFFF_{hex}$ en direcciones físicas, simplemente ignorando los bits más significativos de la dirección virtual, situándolas así en la parte baja de la memoria física. Por lo tanto, el sistema operativo sitúa los puntos de entrada de excepción y las pilas de excepciones en memoria no asignada.

No asignada: parte del espacio de direccionamiento que no tiene fallos de página.

Extensión: El código de la figura 5.28 muestra la secuencia MIPS-32 de retorno desde una excepción. MIPS-I utiliza `rfe` y `jr` en vez de `eret`.

Guardar estado			
Guardar registros de propósito general (GPR)	addi	$k1,$sp, -XCPtamaño	# reservar espacio en la pila para estado
	sw	$sp, XCT_SP($k1)	# guardar $sp en la pila
	sw	$v0, XCT_V0($k1)	# guardar $v0 en la pila
	...		# guardar $v1, $ai, $si, $ti, ... en la pila
	sw	$ra, XCT_RA($k1)	# guardar $ra en la pila
Guardar Hi, Lo	mfhi	$v0	# copiar Hi
	mflo	$v1	# copiar Lo
	sw	$v0, XCT_HI($k1)	# guardar Hi en la pila
	sw	$v1, XCT_LI($k1)	# guardar Lo en la pila
Guardar los registros de excepción	mfc0	$a0, $cr	# copiar el registro de causa
	sw	$a0, XCT_CR($k1)	# guardar $cr en la pila
	...		# guardar $v1, ...
	mfc0	$a3, $sr	# copiar el registro de estado
	sw	$a3, XCT_SR($k1)	# guardar $sr en la pila
Inicializar sp	move	$sp, $k1	# sp = sp - XCPtamaño
Habilitar las excepciones anidadas			
	andi	$v0, $a3, MASK1	# $v0 = $sr & MASK1, habilitar excepciones
	mtc0	$v0, $sr	# $sr = valor que habilita excepciones
Llamada al manejador C de excepciones			
Inicializar $gp	move	$gp, GPINIT	# inicializa $gp para apuntar zona dinámica (heap)
Invocar el código C	move	$a0, $sp	# arg1 = puntero a la pila de excepciones
	jal	xcpt_deliver	# invocar el código C para manejar excepción
Restaurar el estado			
Restaurar la mayoría de GPR, Hi, Lo	move	$at, $sp	# valor temporal de $sp
	lw	$ra, XCT_RA($at)	# restaurar $ra desde la pila
	...		# restaurar $t0, ..., $a1
	lw	$a0, XCT_A0($k1)	# restaurar $a0 desde la pila
Restaurar los registros de estado	lw	$v0, XCT_SR($at)	# load old $sr desde la pila
	li	$v1, MASK2	# mascara para desbilitar excepciones
	and	$v0, $v0, $v1	# $v0 = $sr & MASK2, desbilita excepciones
	mtc0	$v0, $sr	# inicializa el registro de estado
Retorno desde la excepción			
Restaurar $sp y el resto de GPR utilizados como registros temporales	lw	$sp, XCT_SP($at)	# restaurar $sp desde la pila
	lw	$v0, XCT_V0($at)	# restaurar $v0 desde la pila
	lw	$v1, XCT_V1($at)	# restaurar $v1 desde la pila
	lw	$k1, XCT_EPC($at)	# copiar el antiguo $epc desde la pila
	lw	$at, XCT_AT($at)	# restaurar $at desde la pila
Restaurar ERC y retornar	mtc0	$k1, $epc	# restaurar $epc
	eret	$ra	# volver a la instrucción interrumpida

FIGURA 5.28 Código MIPS que guarda y restaura el estado de una excepción.

Extensión: Para procesadores con instrucciones más complejas que pueden incluir accesos a varias posiciones de memoria y escritura de varios datos, es mucho más difícil hacer que las instrucciones sean de ejecución repetida. El procesamiento de una instrucción puede generar varios fallos de página en el medio de la instrucción. Por ejemplo, los procesadores x86 tiene instrucciones de mover bloques de datos que implican accesos a miles de palabras de datos. En estos procesadores, las instrucciones a menudo no pueden ser de ejecución repetida desde el principio, como las instrucciones MIPS. En lugar de esto, la instrucción debe interrumpirse y continuar más tarde

la ejecución desde el centro de la secuencia de accesos. Reanudar una instrucción en medio de su ejecución requiere que se guarde algún estado especial, procesar la excepción y restaurar el estado especial. Para que esto funcione adecuadamente se necesita una coordinación detallada y cuidadosa entre el código de manejo de la excepción del sistema operativo y el hardware.

Resumen

Memoria virtual es el nombre que se da al nivel de jerarquía de memoria que gestiona el alojamiento temporal de información entre la memoria principal y el disco. La memoria virtual permite que un único programa extienda su espacio de direccionamiento más allá de los límites de la memoria principal. Más importante, la memoria virtual permite que la memoria principal se comparta entre varios procesos activos simultáneamente con protección de los espacios de memoria física.

La gestión de la parte de la jerarquía de memoria situada entre la memoria principal y el disco supone un reto debido al alto coste de los fallos de página. Se utilizan distintas técnicas para reducir la frecuencia de fallos:

1. Las páginas se hacen grandes para aprovechar la localidad espacial y reducir la frecuencia de fallos.
2. La correspondencia entre direcciones virtuales y direcciones físicas, que se implementa con una tabla de páginas, se hace completamente asociativa de forma tal que una página virtual puede ser colocada en cualquier lugar de la memoria principal.
3. El sistema operativo utiliza técnicas, como LRU o un bit de acceso, para elegir qué páginas debe reemplazar.

Las escrituras a disco son costosas, así que la memoria virtual utiliza técnicas de escritura retardada y también realiza un seguimiento de la página para saber si no ha sido modificada (utilizando un bit de consistencia) y evitar escribir en disco las páginas que no han cambiado.

El mecanismo de memoria virtual convierte una dirección virtual utilizada por el programa a otra del espacio de direccionamiento físico que se utiliza para acceder a memoria. Esta conversión de direcciones permite proteger el uso compartido de la memoria principal y proporciona varios beneficios adicionales, como la simplificación de la reserva de memoria. Para asegurar que los procesos están protegidos unos de otros, se requiere que sólo el sistema operativo pueda modificar las conversiones de dirección, que se implementan evitando que los programas de usuarios modifiquen la tabla de páginas. El control del uso compartido de las páginas entre los distintos procesos puede implementarse con la ayuda del sistema operativo y de bits de acceso que se incluyen en la tabla de páginas que indican si el programa de usuario tiene acceso de lectura o de escritura a esa página.

Si un procesador tuviera que acceder a una tabla de páginas residente en memoria principal para convertir cada referencia a memoria, la memoria virtual añadiría mucho retardo temporal adicional y ¡las caches serían inútiles! En su lugar, una estructura TLB actúa como una cache para las conversiones realizadas desde la tabla de páginas. Entonces, las direcciones son convertidas de virtual a física utilizando las conversiones almacenadas en la TLB.

Las caches, la memoria virtual y las TLB se fundamentan todas ellas en un mismo conjunto de principios y políticas. La próxima sección describe este marco común.

Comprender las prestaciones de los programas

Aunque la memoria virtual fue inventada para posibilitar que una memoria pequeña actúe como una grande, la diferencia en prestaciones que existe entre el disco y la memoria principal ocasiona que si un programa accede de forma rutinaria a más cantidad de memoria virtual de la que tiene en la memoria física, se ejecuta muy lentamente. Este programa estaría continuamente intercambiando páginas entre la memoria y el disco, lo cual se denomina *hiperpaginación (thrashing)*. Aunque es rara, la hiperpaginación es desastrosa cuando ocurre. Si el programa entra en hiperpaginación, la solución más sencilla es ejecutarlo en un computador con más memoria o ampliarle la memoria al computador. Una solución más compleja consiste en volver a examinar el algoritmo y las estructuras de datos para ver si se puede modificar la localidad, y así reducir el número de páginas que el programa usa simultáneamente. Este conjunto de páginas se denomina informalmente *conjunto de trabajo* o *conjunto local* (*working set*).

Un problema de prestaciones común corresponde al de los fallos de TLB. Ya que una cache TLB puede manejar sólo 32-64 entradas de página a la vez, el programa podría experimentar fácilmente una alta frecuencia de fallos, ya que el procesador puede acceder sólo a menos que un cuarto de megabyte: 64 × 4 KB = 0.25 MB. Por ejemplo, los fallos de TLB constituyen a menudo un reto para el algoritmo Radix Sort. Para intentar aliviar este problema, la mayoría de las arquitecturas de los computadores actuales soportan tamaños variables de página. Por ejemplo, además del tamaño de página de 4 KB, el hardware de MIPS soporta páginas de 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB y 256 MB. Por lo tanto, si un programa utiliza tamaños de página grandes, puede acceder directamente a más memoria sin sufrir fallos de TLB.

En la práctica, el reto consiste en que el sistema operativo permita a los programas que seleccionen estos tamaños de página mayores. Alternativamente, la solución más compleja a la reducción de los fallos de TLB consiste, una vez más, en volver a analizar el algoritmo y las estructuras de datos para reducir el conjunto local de páginas; dada la importancia de los accesos a memoria en las prestaciones y la frecuencia de los fallos de TLB, algunos programas con conjuntos de trabajo grandes se han rediseñados con este objetivo.

Autoevaluación

Asigne cada uno de los elementos de la jerarquía de memoria de la izquierda a la frase de la derecha que más se le parezca:

1. Cache L1	a. Una cache para una cache
2. Cache L2	b. Una cache para discos
3. Memoria Principal	c. Una cache para la memoria principal
4. Cache TLB	d. Una cache para las entradas de la tabla de páginas

5.5 Un marco común para las jerarquías de memoria

Ya hemos reconocido que los distintos tipos de jerarquías de memoria comparten un gran sustrato común. Aunque muchos de los aspectos de las jerarquías de memoria difieran cuantitativamente, muchas de las políticas y características que determinan cómo funciona una jerarquía son cuantitativamente similares. La figura 5.29 muestra cómo algunas de las características cuantitativas de las jerarquías de memoria pueden diferir. En el resto de esta sección analizaremos los aspectos comunes del funcionamiento de las jerarquías de memoria, y cómo éstos determinan su comportamiento. Analizaremos estas políticas como una serie de cuatro cuestiones que se aplican a cualquier pareja de niveles de la jerarquía de memoria, aunque por simplicidad usaremos principalmente la terminología de las caches.

Característica	Valores típicos para caches L1	Valores típicos para caches L2	Valores típicos para la memoria paginada	Valores típicos para una cache TLB
Tamaño total en bloques	250–2000	15 000–50 000	16 000–250 000	40–1024
Tamaño total en kilobytes	16–64	500–4000	1 000 000–1 000 000 000	0.25–16
Tamaño del bloque en bytes	32–64	64–128	4000–64 000	4–32
Penalización por fallo en ciclos	10–25	100–1000	10 000 000–100 000 000	10–1000
Frecuencias de fallos (global para L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURA 5.29 Parámetros cuantitativos clave del diseño que caracterizan a los elementos más importantes de la jerarquía de memoria de un computador. Los valores típicos para estos niveles corresponden al año 2008. Aunque el intervalo de valores es amplio, se debe parcialmente a que muchos de los valores que han variando a lo largo del tiempo están relacionados entre sí. Por ejemplo, a medida que las caches se han hecho más grandes para superar las mayores penalizaciones por fallo, los tamaños de los bloques también han aumentado.

Cuestión 1: ¿Dónde se puede situar un bloque?

Hemos visto que el emplazamiento de bloques en el nivel más alto de la jerarquía de memoria puede utilizar un conjunto de técnicas, desde correspondencia directa, pasando por asociativa por conjuntos, a completamente asociativa. Como se mencionó anteriormente, todas estas técnicas pueden ser consideradas como variaciones de la técnica asociativa por conjuntos, donde el número de conjuntos y el número de bloques por conjunto varía:

Nombre de la técnica	Número de conjuntos	Bloques por conjunto
Correspondencia directa	Número de bloques en la cache	1
Asociativa por conjuntos	$\frac{\text{Número de bloques en la cache}}{\text{Asociatividad}}$	Asociatividad (normalmente) 2–16)
Completamente asociativa	1	Número de bloques en la cache

La ventaja de incrementar el grado de asociatividad consiste en que normalmente se disminuye la frecuencia de fallos. La mejora en la frecuencia de fallos se origina por la reducción de los fallos que compiten por la misma posición. Próxi-

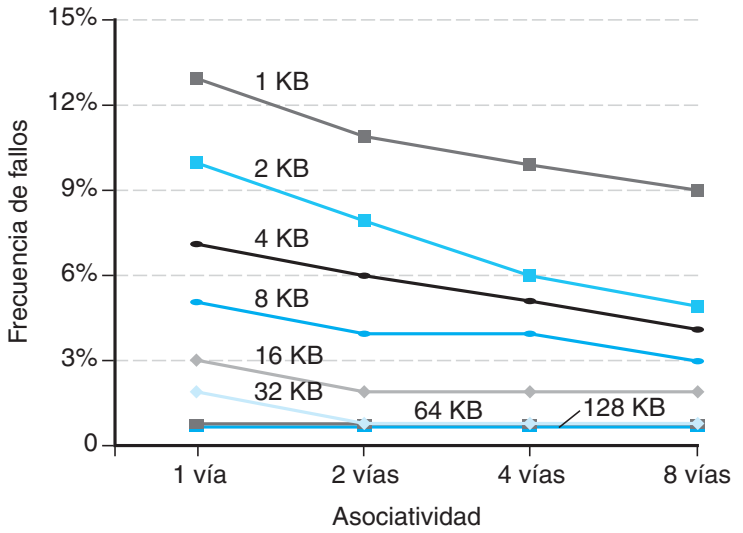


FIGURA 5.30 Frecuencias de fallos de la cache de datos para ocho tamaños de cache mejora a medida que aumenta la asociatividad. Mientras el beneficio de pasar de una asociatividad por conjuntos de una vía (correspondencia directa) a una de dos vías resulta ser significativo, los beneficios de una mayor asociatividad son inferiores (p. ej., 1%-10% cuando se pasa de dos vías a cuatro vías frente a una mejora de 20%-30% cuando se va de una vía a dos vías). Existe incluso una mejora todavía inferior cuando la asociatividad por conjuntos va de cuatro vías a ocho vías, lo cual a su vez, se aproxima mucho a las frecuencias de fallos de un cache completamente asociativa. Las caches más pequeñas obtienen un beneficio absoluto significativamente mayor de la asociatividad, debido a que la frecuencia de fallos base de una cache pequeña es mayor. La figura 5.15 explica cómo fueron obtenidos estos datos.

mamente analizaremos esto con más detalle. Veamos primero cuánta mejora se consigue. La figura 5.30 muestra la frecuencia de fallos para caches de varios tamaños con la asociatividad variando desde correspondencia directa a asociativa por conjuntos de ocho vías. Las mayores ganancias se obtiene al pasar de correspondencia directa a asociativa por conjuntos de dos vías, con reducciones del 20% al 30% en la frecuencia de fallos. A medida que la capacidad de la cache es mayor, la mejora relativa que proporciona la asociatividad aumenta sólo un poco. Ya que la frecuencia global de fallos de una cache más grande es menor, la oportunidad de mejorar la frecuencia de fallos disminuye, y por ello, la mejora absoluta de la frecuencia de fallos que proporciona la asociatividad se reduce significativamente. Las desventajas que potencialmente puede presentar la asociatividad, como se mencionó anteriormente, corresponden al incremento del coste y a un tiempo de acceso más largo.

Cuestión 2: ¿Cómo se encuentra un bloque?

La forma de localizar un bloque depende del método de emplazamiento de los bloques, ya que determina el número de posibles localizaciones. A continuación, se resumen los métodos:

Asociatividad	Método de localización	Comparaciones necesarias
Correspondencia directa	índice	1
Asociativa por conjuntos	Indexa el conjunto, busca entre los elementos	Grado de asociatividad
Completamente asociativa	Inspecciona todas las entradas de la cache	Capacidad de la cache
	Tabla de consulta separada	0

La elección entre correspondencia directa, asociativa por conjuntos o completamente asociativa en cualquier jerarquía de memoria dependerá del coste de un fallo en relación con el coste de implementar la asociatividad, tanto en tiempo como en hardware adicional. La inclusión de la cache L2 en el chip permite mucha más asociatividad, debido a que el tiempo de acierto no es tan crítico y el diseñador no está obligado a utilizar chips SRAM estándar como bloques básicos para la construcción del computador. Las caches completamente asociativas son prohibitivas excepto para tamaños pequeños, donde el coste de los comparadores no es exagerado y las mejoras de la frecuencia de fallos absoluta son las más importantes.

En los sistemas de memoria virtual, se emplea una tabla de correspondencias separada (la tabla de páginas) para indexar la memoria. Además del almacenamiento necesario para la tabla, el uso de una tabla de índices implica un acceso extra a memoria principal. La elección de la asociatividad completa para el emplazamiento de páginas, así como la tabla adicional están motivadas por cuatro hechos:

1. La asociatividad completa es beneficiosa, ya que los fallos son muy costosos.
2. La asociatividad completa permite que el software utilice métodos sofisticados de reemplazos diseñados para reducir la frecuencia de fallos.
3. El mapa de correspondencias completo puede indexarse fácilmente sin necesidad de hardware adicional ni que se aplique ningún método de búsqueda.

Por lo tanto, los sistemas de memoria virtual casi siempre utilizan emplazamiento completamente asociativo.

El emplazamiento asociativo por conjuntos es a menudo utilizado por las caches y las TLBs, donde el acceso combina una indexación y una búsqueda dentro de un pequeño conjunto. Unos pocos sistemas han utilizado caches de correspondencia directa debido a sus ventajes en el tiempo de acceso y a su simplicidad. La ventaja en el tiempo de acceso se debe a que la búsqueda del bloque solicitado no depende de una comparación. Tales opciones de diseño dependen de muchos detalles de la implementación, tales como si la cache está en el chip o no, la tecnología utilizada para implementar la cache, y el papel que juega el tiempo de acceso a la cache en la determinación del tiempo de ciclo del procesador.

Cuestión 3: ¿Qué bloque se reemplaza en un fallo de cache?

Cuando se produce un fallo en una cache asociativa, debemos decidir qué bloque reemplazar. En una cache completamente asociativa, todos los bloques son candidatos a ser reemplazados. Si la cache es asociativa por conjuntos, debemos escoger

entre los bloques de un conjunto. Evidentemente, el reemplazo es sencillo en una cache de correspondencia directa debido a que existe un solo candidato.

Ya hemos mencionado las dos estrategias de reemplazos más importantes para caches asociativas por conjuntos o completamente asociativas:

- *Aleatorio:* Los bloques candidatos son seleccionados aleatoriamente, posiblemente utilizando alguna ayuda hardware. Por ejemplo, MIPS utiliza reemplazos aleatorios para los fallos de TLB.
- *Menos recientemente usado* (LRU): El bloque reemplazado corresponde al que no se ha usado durante el tiempo más largo.

En la práctica, LRU es demasiado costoso de implementar para jerarquías con más de un pequeño grado de asociatividad (normalmente, de dos a cuatro), ya que el seguimiento de la información relacionada con el uso es costoso. Incluso para asociatividades por conjuntos de cuatro vías, se utiliza con frecuencia una aproximación a LRU; por ejemplo, guardando qué par de bloques es LRU (lo cual requiere 1 bit), y luego guardando qué bloque en cada par es LRU (lo cual requiere 1 bit por cada par).

Para asociatividades mayores se utiliza, o bien una aproximación a LRU, o bien reemplazos aleatorios. En las caches, el algoritmo de reemplazos está en hardware, lo cual significa que el método debe ser fácil de implementar. El reemplazo aleatorio es sencillo de construir en hardware y, para una cache asociativa por conjuntos de dos vías, el reemplazo aleatorio tiene una frecuencia de fallos de alrededor 1.1 veces mayor que el reemplazo LRU. A medida que las caches se hacen más grandes, la frecuencia de fallos para ambas estrategias de reemplazos baja, y la diferencia absoluta se convierte pequeña. De hecho, el reemplazo aleatorio puede a veces ser mejor que las simples aproximaciones a LRU que son fácilmente implementadas en hardware.

En la memoria virtual, siempre se utiliza alguna forma aproximada de LRU, ya que incluso una minúscula reducción de la frecuencia de fallos puede ser importante cuando el coste de un fallo es enorme. A menudo es posible utilizar los bits de acceso o su funcionalidad equivalente para hacer que el sistema operativo realice más fácilmente el seguimiento de un conjunto de páginas menos recientemente utilizadas. Como los fallos son tan costosos y relativamente infrecuentes, la aproximación a esta información realizada principalmente en software es aceptable.

Cuestión 4: ¿Qué ocurre en una escritura?

Un aspecto clave de cualquier jerarquía de memoria consiste en saber cómo se tratan las escrituras. Ya hemos visto las dos opciones básicas:

- *Escritura directa*: la información se actualiza tanto en el bloque de la cache como en el bloque del nivel inferior de la jerarquía de memoria (memoria principal para una cache). Las caches de la sección 5.2 utilizaban este esquema.
- *Escritura retardada* (denominado también *copia retardada*): la información se actualiza sólo en el bloque de la cache. El bloque modificado es actualizado en el nivel inferior de la jerarquía sólo cuando es reemplazado. Los sistemas de memoria virtual siembre utilizan escritura retardada, por las razones que se indicaron en la sección 5.4.

Tanto la escritura retardada como la escritura directa tienen sus ventajas. Las ventajas que son claves en la escritura retardada son las siguientes:

- Las palabras pueden ser individualmente almacenadas por el procesador al ritmo de la cache, en vez del que puede aceptar la memoria principal.
- Múltiples escrituras dentro del bloque requieren sólo una escritura en el nivel inferior de la jerarquía.
- Cuando los bloques se actualizan en el nivel inferior de la jerarquía, el sistema puede hacer uso de un alto ancho de banda, ya que se escribe el bloque entero.

Las escrituras directas tienen estas ventajas:

- Los fallos son más simples y menos costosos debido a que nunca requieren que un bloque sea actualizado en el nivel inferior de la jerarquía.
- La escritura directa es más sencilla de implementar que la escritura retardada, aunque para ser prácticos en un sistema de alta frecuencia, una cache de escritura directa necesitará usar un búfer de escritura.

En sistemas de memoria virtual sólo es práctica una política de escritura retardada, debido a la alta latencia de una escritura en el nivel inferior de la jerarquía (el disco). El ritmo al cual las escrituras son generadas por el procesador generalmente excede el ritmo al cual el sistema de memoria puede procesarlos, incluso con el uso de memorias que físicamente y lógicamente son más anchas y modos de acceso mejorados (*burst*) para la DRAM. En consecuencia, las caches en los niveles más bajos de la jerarquía utilizan normalmente la estrategia de escritura retardada.

IDEA clave

Mientras las caches, TLBs, y memoria virtual pueden inicialmente parecer muy distintas entre si, se basan en los mismos dos principios de localidad, y su funcionamiento puede ser entendido a través de cómo resuelven cuatro cuestiones:

Cuestión 1: ¿Dónde se puede situar a un bloque?
Respuesta: En un único lugar (correspondencia directa), unos pocos lugares (asociativa por conjuntos), o en cualquier lugar (completamente asociativa).

Cuestión 2: ¿Cómo se encuentra un bloque?
Respuesta: Existen cuatro métodos: por indexación (como en una cache de correspondencia directa), a través de una búsqueda limitada (como en una cache asociativa por conjuntos), a través de una búsqueda completa (como en una cache completamente asociativa), o a través de una tabla separada para consultas (como en una tabla de páginas).

Cuestión 3: ¿Qué bloque se reemplaza en un fallo?
Respuesta: Normalmente el menos recientemente utilizado, o un bloque escogido aleatoriamente.

Cuestión 4: ¿Cómo se manejan las escrituras?
Respuesta: Cada nivel de la jerarquía puede usar bien escritura directa o escritura retardada.

Las tres C: un modelo para entender el comportamiento de las jerarquías de memoria

En esta sección analizaremos un modelo que nos ayuda a comprender los orígenes de los fallos en una jerarquía de memoria y los efectos de los cambios en la jerarquía sobre los fallos. Explicaremos las ideas en término de caches, pero estas ideas se pueden extender directamente a cualquier otro nivel de la jerarquía. En este modelo, todos los fallos se clasifican en una de estas tres categorías (las **tres C**):

Modelo de las tres C: modelo de cache en el cual los fallos se clasifican en tres categorías: fallos obligatorios (*compulsory misses*), fallos de capacidad (*capacity misses*) y fallos de conflicto (*conflict misses*).

- **Fallos obligatorios (*compulsory misses*)**: son fallos de cache causados por el primer acceso a un bloque que nunca ha estado en la cache. También se les denomina **fallos en frío**.
- **Fallos de capacidad (*capacity misses*)**: son fallos de cache causados cuando la cache no puede contener todos los bloques que necesita un programa durante su ejecución. Los fallos de capacidad se producen cuando los bloques son reemplazados y posteriormente vuelven a la cache.
- **Fallos de conflicto (*conflict misses*)**: son fallos que se producen en caches asociativas por conjuntos o de correspondencia directa cuando varios bloques compiten por el mismo conjunto. Los fallos de conflicto son aquellos fallos que en una cache de correspondencia directa o asociativa por conjuntos son eliminados utilizando una cache completamente asociativa de la misma capacidad. Estos fallos de cache también se denominan **fallos de colisión**.

Fallo obligatorio (fallo en frío): fallo de cache causado por el primer acceso a un bloque que nunca ha estado en la cache.

Fallo de capacidad: fallo de cache que se produce porque la cache, incluso la de tipo completamente asociativa, no puede contener todos los bloques que son necesarios para satisfacer las peticiones de un programa.

Fallo de conflicto (fallo de colisión): fallo de cache que se produce en una cache asociativa por conjuntos o de correspondencia directa cuando varios bloques compiten por el mismo conjunto y que no aparece en una cache completamente asociativa de la misma capacidad.

La figura 5.31 muestra cómo la frecuencia de fallos se separa en las tres componentes. Estas componentes pueden ser directamente combatidas cambiando algunos aspectos del diseño de las caches. Ya que los fallos de conflicto surgen directamente de la competencia por el mismo bloque de cache, el aumento de la asociatividad reduce los fallos de conflicto. La asociatividad, sin embargo, puede ralentizar el tiempo de acceso, originando unas prestaciones globales inferiores.

Los fallos de capacidad pueden fácilmente ser reducidos agrandando la cache. De hecho, las caches de segundo nivel han estado creciendo en capacidad a ritmo constante durante muchos años. Evidentemente, cuando hacemos una cache más grande, debemos ser cuidadosos con el incremento del tiempo de acceso, el cual podría originar prestaciones globales inferiores. De este modo, las caches de primer nivel han estado creciendo lentamente.

Como los fallos obligatorios son generados por el primer acceso a un bloque, la forma que principalmente usa un sistema cache para reducir el número de fallos obligatorios consiste en incrementar el tamaño de bloque. Esto reducirá el número de accesos requeridos para ponerse en contacto con un bloque por primera vez, debido a que el programa estará formado por menos bloques. Como se ha mencionado anteriormente, si el tamaño de bloque aumenta demasiado, se puede tener un efecto negativo en las prestaciones debido al incremento de la penalización por fallo.

La descomposición de los fallos constituye un modelo cualitativo útil. En el diseño de caches reales, muchas de las opciones de diseño interaccionan, y el cambio de una característica de la cache afecta a menudo a varios componentes de la

frecuencia de fallos. Aparte de estos inconvenientes, este modelo es una forma útil de comprender mejor el rendimiento de los diseños de las caches.

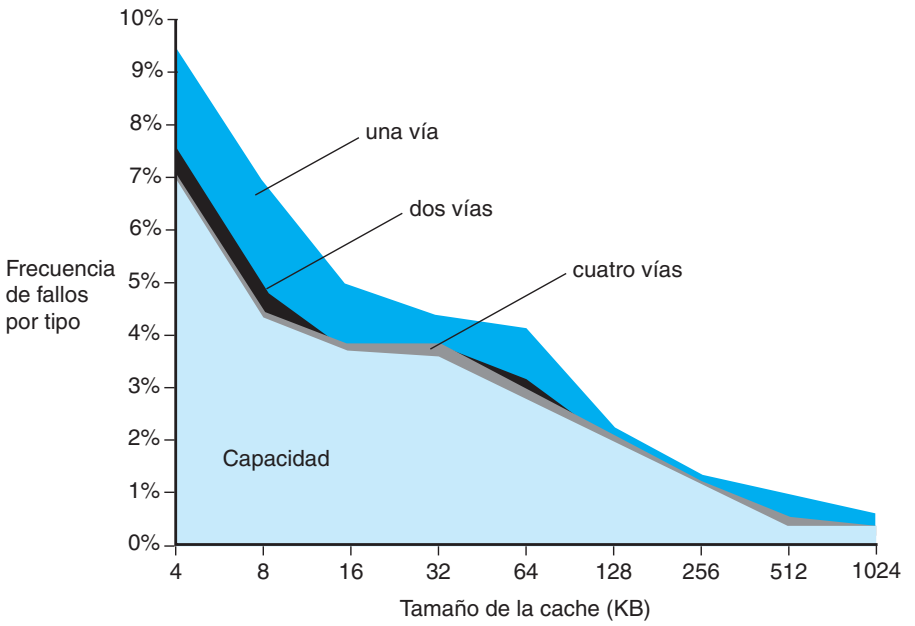


FIGURA 5.31 En la frecuencia de fallos se pueden identificar tres causas que producen fallos. Este gráfico muestra la frecuencia de fallos total y sus componentes para un rango de capacidades de la cache. Estos datos son para programas de prueba SPEC2000 de tipo entero y punto flotante, y fueron obtenidos de la misma fuente que la de los datos de la figura 5.30. La componente de los fallos obligatorios es 0.006% y no se puede observar en este gráfico. La siguiente componente es la frecuencia de fallos de capacidad, la cual depende de la capacidad de la cache. La parte debida a los conflictos, la cual depende tanto de la asociatividad como de la capacidad de la cache, se muestra para un rango de asociatividades que van desde una vía hasta ocho vías. En cada caso, las partes de distinto color corresponden al incremento de la frecuencia de fallos que se produce cuando la asociatividad cambia desde el grado de asociatividad superior más próximo al grado de asociatividad asociado al color. Por ejemplo, la parte etiquetada dos vías indica los fallos adicionales que se han producido cuando la cache tiene una asociatividad de dos en vez de cuatro. De este modo, la diferencia en la frecuencia de fallos que presenta una cache de correspondencia directa respecto a una cache completamente asociativa de la misma capacidad viene dada por la suma de las partes marcadas como *ocho vías, cuatro vías, dos vías* y *una vía*. La diferencia entre ocho vías y cuatro vías es tan pequeña que es difícil observarla en esta gráfico.

IDEA clave

El desafío involucrado en el diseño de las jerarquías de memorias consiste en que cada cambio que potencialmente mejora la frecuencia de fallos puede también afectar negativamente a las prestaciones globales, como se resume en la figura 5.32. Esta combinación de efectos positivos y negativos es lo que hace que el diseño de la jerarquía de memoria sea interesante.

Cambio en el diseño	Efecto sobre la frecuencia de fallos	Posible efecto negativo sobre las prestaciones
Aumenta la capacidad de la cache	Disminuyen los fallos de capacidad	Puede aumentar el tiempo de acceso
Aumenta la asociatividad	Disminuye la frecuencia de fallos debida a fallos de conflicto	Puede aumentar el tiempo de acceso
Aumenta el tamaño de bloque	Disminuye la frecuencia de fallos para una rango amplio de tamaños de bloques debido a la localidad espacial	Aumenta la penalización por fallo Bloques muy grandes podrían aumentar la frecuencia de fallos

FIGURA 5.32 Retos del diseño de la jerarquía de memoria.

Autoevaluación

¿Cuáles de las siguientes afirmaciones son generalmente verdad (si existiese alguna)?

1. No existe manera de reducir los fallos obligatorios.
2. Las caches completamente asociativas no tienen fallos de conflicto.
3. Para reducir los fallos, la asociatividad es más importante que la capacidad.

5.6 Máquinas virtuales

Una idea relacionada con la memoria virtual y casi tan antigua como ella son las Máquinas Virtuales (MV). Se desarrollaron a mediados de la década de los 60 y se han mantenido a lo largo de los años como una parte importante en el campo de los grandes servidores. Aunque fueron ignoradas en el dominio de los computadores de un solo usuario en las décadas de 1980 y 1990, recientemente han ganado popularidad por los siguientes motivos:

- La importancia creciente del aislamiento y seguridad en los sistemas modernos
- Los fallos en seguridad y fiabilidad de los sistemas operativos estándar
- La utilización de un único computador por varios usuarios no relacionados
- El espectacular aumento de la velocidad de los procesadores en las últimas décadas, que hace aceptable el sobrecoste introducido por las MV.

La definición más amplia de una MV incluye básicamente todos los métodos de emulación que proporcionan una interfaz software estándar, como la MV de Java. En esta sección nos centramos en las MV que proporcionan un entorno completo a nivel de sistema de la arquitectura del repertorio de instrucciones (ISA). Aunque algunas MV pueden ejecutar ISA diferentes del computador nativo, supondremos que la ISA siempre corresponde a la del computador nativo. Algunos ejemplos son IBM VM/370, Vmware ESX Server y Xen.

Las máquinas virtuales a nivel de sistema proporcionan al usuario la ilusión de que dispone de un computador completo para él solo, incluida una copia del sistema operativo. Un computador ejecuta varias MV y puede soportar varios sistemas operativos diferentes (SO). En una plataforma convencional, un único SO es el "propietario" de todos los recursos hardware, pero con una MV varios SO comparten estos recursos hardware.

El software que da soporte a las MV se denomina *monitor de máquinas virtuales* (MMV) o *hipervisor*; el MMV es el corazón de la tecnología de las máquinas virtuales. La plataforma hardware subyacente se denomina *anfitrión (host)* y sus recursos son compartidos entre las MV *invitadas*. El MMV determina la correspondencia entre recursos virtuales y recursos físicos: un recurso físico puede estar compartido en el tiempo, dividido o incluso emulado en software. El MMV es mucho más pequeño que un SO tradicional; la parte de aislamiento de una MMV puede tener unas 10 000 líneas de código.

Aunque estamos interesados en las MV para mejorar la protección, las MV proporcionan otros dos beneficios de interés comercial:

1. *Manejo de software*. Las MV proporcionan una abstracción que permite ejecutar software antiguo, incluyendo sistemas operativos como DOS. Una estructura típica podría ser disponer de algunas MV ejecutando SO antiguos, varias ejecutando versiones estables de SO actuales y unas pocas evaluando próximas versiones de los SO.

2. *Manejo de hardware*. Una de las razones de utilizar varios servidores es tener cada aplicación ejecutándose en una versión compatible del sistema operativo en computadores independientes, puesto que esta separación mejora la fiabilidad. Las MV permiten que esas versiones diferentes del software puedan ejecutarse independientemente compartiendo el mismo hardware, reduciendo así el número de servidores. Otro ejemplo es que los MMV permiten la migración de una MV a otro computador diferente, bien para equilibrar la carga de trabajo, bien para evitar hardware que no funciona correctamente.

En general, el coste de la virtualización del procesador depende de la carga de trabajo. El sobrecoste de la virtualización de programas de usuario limitados por las prestaciones del procesador es cero, puesto que se llama al SO en contadas ocasiones y todo se ejecuta a la velocidad original. Las cargas de trabajo con operaciones de E/S intensivas generalmente hacen un uso intensivo del SO, ejecutando muchas llamadas al sistema e instrucciones con privilegios modo núcleo, y esto puede suponer que el sobrecoste de la virtualización sea elevado. Por otra parte, si las cargas de trabajo intensivas en operaciones de E/S están también limitadas en sus prestaciones por las operaciones de E/S, el coste de la virtualización puede ocultarse completamente, porque el procesador está a menudo inactivo esperando por la E/S.

El sobrecoste está determinado por el número de instrucciones que debe emular el MMV y por el tiempo que se necesita para emular cada una. Así, cuando la MV invitada ejecuta la misma ISA que el anfitrión, como suponemos en esta sección, el objetivo de la arquitectura y del MMV es ejecutar la mayoría de las instrucciones directamente en el hardware nativo.

Requisitos de un Monitor de Máquina Virtual

¿Qué debe hacer un monitor de MV? Presentar una interfaz software al software invitado, debe aislar el estado de cada invitado del resto de los invitados y debe protegerse del software invitado (incluyendo SO invitados). Los requisitos cualitativos son:

- El software invitado debe comportarse en la MV exactamente igual que si estuviese ejecutándose en el hardware nativo, excepto en lo relacionado con las prestaciones y en las limitaciones derivadas de compartir los recursos con varias MV.
- El software invitado no debe poder cambiar directamente la asignación de recursos reales del sistema.

Para "virtualizar" el procesador, el MMV tiene que controlar casi todo, acceso a estados con privilegios, traducción de direcciones, excepciones de E/S e interrupciones, aunque la MV invitada y el SO actual los estén utilizando temporalmente.

Por ejemplo, en el caso de una interrupción del reloj, el MMV debería suspender la MV invitada en ejecución, guardar su estado, procesar la interrupción, determinar qué MV invitada se va a ejecutar a continuación y cargar su estado. Las MV invitadas que dependen de una interrupción del reloj disponen de un temporizador virtual y de una interrupción del reloj emulada por el MMV.

Para estar a cargo de todas estas tareas, el MMV debe tener más privilegios que las MV invitadas, que generalmente se ejecutan en modo usuario; esto también asegura que la ejecución de cualquier instrucción privilegiada será realizada por el MMV. Los requisitos básicos de un sistema de máquinas virtuales son casi idénticos a los requisitos de memoria virtual paginada que se indican a continuación:

- Como mínimo dos modos de operación, sistema y usuario
- Un subconjunto privilegiado de instrucciones disponible sólo para ejecutar en modo sistema, generando una interrupción software (*trap*) si se ejecuta en modo usuario; todos los recursos del sistema deben controlarse únicamente con estas instrucciones.

(Carencia de) Soporte de arquitectura del repertorio de instrucciones para máquinas virtuales

Si se planifican las MV durante el diseño de la ISA, es relativamente sencillo reducir el número de instrucciones que deben ejecutarse en la MV y su velocidad de emulación. Una arquitectura que permite que la MV ejecute directamente en el hardware es *virtualizable*, y la arquitectura del IBM 370 lleva orgullosamente esta etiqueta.

Desgraciadamente, y como consecuencia de que las MV han sido implementadas en computadores de sobremesa y servidores de aplicaciones basados en PC solo desde hace muy poco tiempo, la mayoría de los repertorios de instrucciones se diseñaron sin tener en cuenta la virtualización. Entre estos culpables se encuentran el x86 y la mayor parte de las arquitecturas RISC, incluyendo ARM y MIPS.

Puesto que el MMV debe asegurar que el sistema invitado sólo interactúe con recursos virtuales, se ejecuta un SO invitado convencional como un programa de usuario en la MMV. De estas forma, si el SO invitado intenta acceder o modificar información relacionada con recursos hardware utilizando instrucciones privilegiadas, por ejemplo, leyendo o escribiendo el puntero de la tabla de páginas, se produce una excepción software en el MMV. Entonces, el MMV puede hacer los cambios apropiados en el recurso real correspondiente.

Así, si cualquier instrucción que intenta leer o escribir esta información sensible genera una excepción software al ejecutarse en modo usuario, el MMV puede mantener una versión virtual de la información sensible, tal como espera el SO invitado.

En ausencia de este soporte, deben tomarse otras medidas. Un MMV debe tener especial cuidado a la hora de ubicar las instrucciones problemáticas y asegurase de que se comportan correctamente cuando las ejecuta el SO invitado, incrementando, de este modo, la complejidad del MMV y reduciendo las prestaciones de la ejecución de la MV.

Protección y arquitectura del repertorio de instrucciones

La protección es un esfuerzo conjunto de la arquitectura y el sistema operativo, pero cuando el soporte para memoria virtual se hizo popular, los diseñadores de la arquitectura tuvieron que modificar algunos detalles delicados de las arquitecturas del repertorio de instrucciones. Por ejemplo, para incluir soporte de memoria virtual en el IBM 370 fue necesario cambiar la exitosa arquitectura del repertorio de instrucciones del IBM 360, que había sido anunciado sólo seis años antes. Ajustes similares se están haciendo en la actualidad para dar cabida a las máquinas virtuales.

Por ejemplo, la instrucción POPF del x86 carga los registros indicadores de condiciones desde la pila en memoria. Uno de los indicadores es Interrupción Habilitada (*Interrupt Enable*, IE); si se ejecuta la instrucción POPF en modo usuario, en lugar de producir una excepción, se cambian todos los indicadores excepto IE. En modo sistema, se cambia también el indicador IE. Puesto que el SO invitado se está ejecutando en la MV en modo usuario, esto puede ser un problema, porque espera que se cambie el indicador IE.

Históricamente, el hardware de los grandes computadores de IBM y el MMV dieron los siguientes pasos para mejorar las prestaciones de las máquinas virtuales:

1. Reducir el coste de la virtualización del procesador
2. Reducir el sobrecoste debido a la virtualización en las interrupciones
3. Reducir el coste de las interrupciones dirigiendo las interrupciones a la MV sin intervención del MMV

En 2006, las nuevas propuestas de AMD e Intel abordaron el primer paso, reducir el coste de la virtualización del procesador. Será interesante ver cuántas generaciones de arquitecturas y modificaciones del MMV serán necesarias para abordar los tres pasos, cuánto se tardará antes de que las máquinas virtuales del siglo 21 sean tan eficientes como los computadores de IBM y los MMV de los años 70.

Extensión: Además de la virtualización del repertorio de instrucciones, otro reto es la virtualización de la memoria virtual, dado que cada SO invitado en la MV maneja un conjunto propio de tablas de páginas. Para conseguir que esto funcione, el MMV separa las nociones de *memoria real* y *memoria física* (que a menudo se consideran sinónimos) y hace que la memoria real sea un nivel intermedio entre la memoria virtual y la memoria física. (A veces se utilizan los términos *memoria virtual*, *memoria física* y *memoria de la máquina* para referirse a estos tres niveles). El SO invitado proyecta la memoria virtual en memoria física mediante las tablas de páginas, y las tablas de páginas del MMV proyectan la memoria real del invitado en memoria física. La arquitectura de la memoria virtual se especifica bien a través de tablas de páginas, como en el IBM VM/370 y el x86, bien a través de TLB como en el MIPS.

En lugar de tener un nivel extra de indirección en cada acceso a memoria, el MMV mantiene una *tabla de páginas acompañante* (shadow page table) que proyecta directamente el espacio de direcciones virtuales del invitado en el espacio de direcciones físicas del hardware. Detectando todas las modificaciones de la tabla de páginas del invitado, el MMV puede asegurar que la tabla de páginas acompañante que el hardware está usando para las traducciones, corresponde a la tabla de páginas del SO invitado, con la excepción de las páginas físicas correctas sustituidas por páginas reales en las tablas de invitados. Así, el MMV debe detectar cualquier intento del SO invitado de cambiar su tabla de páginas o de acceder al puntero de la tabla de páginas. Habitualmente esto se consigue protegiendo contra escritura la tabla de páginas del invitado y detectando cualquier intento de acceso al puntero de la tabla de páginas por parte del SO invitado. Como se ha dicho anteriormente, el acceso al puntero de la tabla de páginas se detecta de forma simple si es una operación privilegiada.

La parte final de la arquitectura que puede virtualizarse es la E/S. Esta es, con diferencia, la parte más complicada de la virtualización del sistema debido al cada vez mayor número de dispositivos de E/S conectados al computador y a la cada vez mayor variedad de dispositivos de E/S. Otras dificultades son la compartición de un dispositivo real entre varias MV y el soporte del gran número de controladores de dispositivos que se necesitan, especialmente si se mantienen varios SO invitados diferentes en el mismo sistema de MV. La ilusión de la MV se puede mantener dándole a cada MV versiones genéricas de cada tipo de controlador de dispositivo de E/S y dejando que el MMV controle el dispositivo de E/S real.

5.7 Utilización de una máquina de estados finitos para el control de una cache sencilla

Al igual que en el capítulo 4, donde se implementó el control para un camino de datos de ciclo único y para un camino de datos segmentado, ahora podemos implementar el control de la cache. Esta sección comienza con la definición de una cache sencilla y la descripción de máquinas de estados finitos (MEF), y termina con el MEF de un controlador de esta cache sencilla. La sección 5.9 en el CD va un paso más allá, mostrando la cache y el controlador en un nuevo lenguaje de descripción hardware.

Una cache sencilla

Vamos a diseñar el controlador de una cache sencilla. Las características de la cache son las siguientes:

- Cache de correspondencia directa
- Escritura retardada con reserva de escritura
- El tamaño de bloque es 4 palabras (16 bytes o 128 bits)
- El tamaño de cache es 16 KB, es decir 1024 bloques
- Direcciones de byte de 32 bits
- Incluye un bit de validez y un bit de consistencia por bloque

De acuerdo con la sección 5.2, podemos determinar los campos de una dirección de cache:

- Índice de cache de 10 bits
- Desplazamiento del bloque de 4 bits
- El tamaño de la etiqueta es $32 - (10 + 4) = 18$ bits

Las señales entre procesador y cache son:

- Señal de Lectura o Escritura (1 bit)
- Señal de Validez (1 bit), que indica si es una operación de cache
- Direcciones de 32 bits
- Datos de 32 bits del procesador a la cache
- Datos de 32 bits de la cache al procesador
- Señal de Preparado (1 bit), que indica que se ha completado la operación de la cache

Observe que ésta es una cache bloqueante, es decir, el procesador debe esperar hasta que la cache ha terminado la operación.

La interfaz entre la memoria y la cache tiene los mismos campos que entre el procesador y la cache, excepto que ahora el campo de datos es de 128 bits. Este ancho extra de memoria es habitual en los microprocesadores actuales, que utilizan palabras de 32 o 64 bits en el procesador y 128 bits en el controlador de DRAM. El diseño se simplifica haciendo que el tamaño del bloque de cache sea igual al ancho de la DRAM. Estas son las señales:

- Señal de Lectura o Escritura (1 bit)
- Señal de Validez (1 bit), que indica si es una operación de memoria
- Direcciones de 32 bits
- Datos de 128 bits de la cache a la memoria
- Datos de 32 bits de la memoria a la cache
- Señal de Preparado (1 bit), que indica que se ha completado la operación de la memoria

Observe que la interfaz con la memoria no implica un número fijo de ciclos. Supondremos que existe un controlador de memoria que notifica a la cache, mediante la señal Listo, que la operación de escritura o lectura en memoria ha finalizado.

Antes de describir el controlador de cache, repasaremos las máquinas de estados finitos, que nos permitirán controlar una operación que necesita varios ciclos para completarse.

Máquinas de estado finitos

En el diseño de la unidad de control del camino de datos de ciclo único se utilizaron tablas de verdad para especificar la activación de las señales de control en función del tipo de instrucción. En una cache el control es más complejo porque la operación puede consistir en una serie de pasos. El control de la cache debe especificar las señales que se van a activar en un paso dado y en el siguiente paso de la secuencia.

El método de control multipaso más extendido se basa en las **máquinas de estados finitos**, que generalmente se representan de forma gráfica. Una máquina de estados finitos consiste en un conjunto de estados e indicaciones de como se producen los cambios de estado. Las indicaciones se definen con una **función de estado siguiente**, que obtiene el nuevo estado a partir del estado actual y las entradas. Al utilizar una máquina de estados finitos para el control, cada estado proporciona también un conjunto de salidas que se activan cuando la máquina está en ese estado. La implementación de una máquina de estados finitos habitualmente supone que todas las salidas para las que no se indican de forma explícita sus valores, valen cero. De forma similar, la operación correcta del camino de datos depende de que las señales para las que no se indica explícitamente su valor sean cero, en lugar de una indeterminación.

Máquinas de estados finitos: Función lógica secuencial que consiste de un conjunto de entradas y salidas, una función de estado siguiente que determina el nuevo estado en función del estado actual y las entradas, y una función de salida que determina las salidas activas en función del estado actual y posiblemente las entradas.

función de estado siguiente: Una función combinacional que, dadas las entradas y el estado actual, determina el estado siguiente de una máquina de estado finitos

La señal de control de un multiplexor funciona de forma algo diferente, porque selecciona una de las entradas tanto cuando vale 0 como cuando vale 1. Por lo tanto, en la máquina de estado finitos, siempre especificaremos el valor de las señales de control de los multiplexores que intervengan en la operación del control. Al implementar una máquina de estados finitos con lógica digital, el valor por defecto de una señal de control puede ser 0, y para fijar este valor no haría falta ninguna puerta lógica. Un ejemplo sencillo de una máquina de estados finitos se muestra en el **apéndice C**, y si no está familiarizado con el concepto de máquina de estados finitos, debería consultar el **apéndice C** antes de continuar.

La máquina de estados finitos puede implementarse con un registro temporal, que mantiene el estado actual, y un bloque de lógica combinacional que determina las señales de control que deben activarse en el camino de datos y el estado siguiente. La figura 5.33 muestra el aspecto general de una implementación. En el **apéndice D** se describe en detalle como se implementa una máquina de estados finitos según este esquema. En la **sección C.3** la lógica combinacional de la máquina de estados finitos se implementa con una ROM (*read-only memory* o memoria de sólo lectura) y una PLA (*programmable logic array* o matriz lógica programable). (Véase también la descripción de estos elementos en el **apéndice C**.)

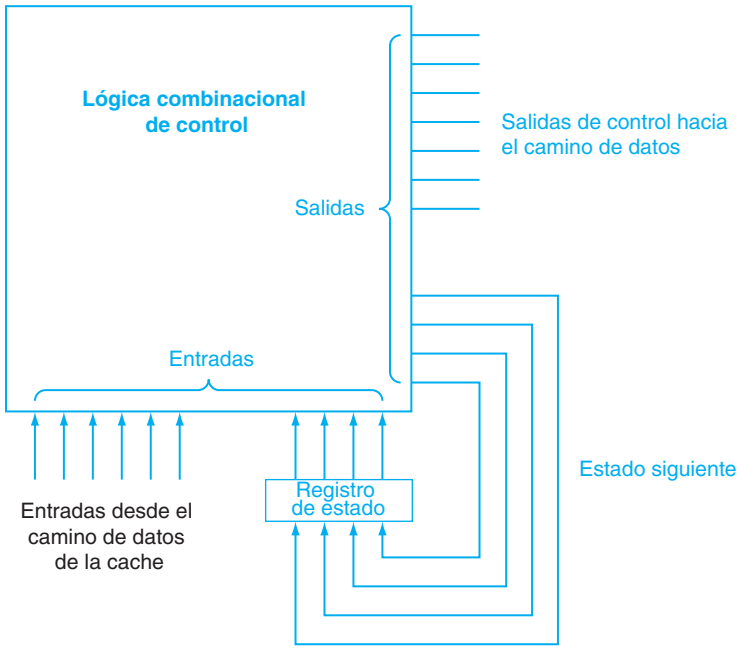


FIGURA 5.33 La máquinas de estado finito de control se implementan habitualmente con un bloque de lógica combinacional y un registro para mantener el estado actual. Las salidas del bloque de lógica combinacional son la codificación del estado siguiente y las señales de control que deben activarse durante el estado actual. Las entradas del bloque combinacional son el estado actual y cualquier señal de entrada que deba usarse para la determinación del estado siguiente. En este caso, las entradas son los bits del código de operación del registro de la instrucción. Observe que en la máquina de estados finitos que se utiliza en este capítulo, las salidas dependen únicamente del estado actual y no de las entradas. La *Extensión* explica esto con más detalle.

Extensión: El tipo de máquinas de estados finitos utilizado en este libro se llama máquinas Moore, y reciben este nombre de Edward Moore. Su característica principal es que la salida sólo depende del estado actual. En una máquina Moore, la caja etiquetada como lógica de control combinacional puede dividirse en dos partes. Una parte con la las salidas de control y el estado de entrada, y la otra parte con únicamente la salida de estado siguiente.

Un tipo alternativo es la máquina Mealy, que recibe este nombre de George Mealy. La máquina Mealy permite que tanto las entradas como el estado actual se utilicen para determinar la salida. Las ventajas de las máquinas Moore son que resultan en mejores implementaciones en cuanto a la velocidad y el tamaño de la unidad de control. Las ventajas en la velocidad se deben a que las salidas de control, que se necesitan al comienzo del ciclo de reloj, no dependen de las entradas, sólo dependen del estado actual. En el apéndice C, al implementar esta máquina de estados finitos con puertas lógicas, se ve claramente la ventaja en tamaño. La desventaja de una máquina Moore es que necesita más estados. Por ejemplo, en situaciones en las que dos secuencias de estados difieren sólo en un estado, la máquina Mealy puede unificar los estados haciendo que la salida dependa de las entradas.

MEF para un controlador de cache sencillo

La figura 5.34 muestra los estados de nuestro controlador de cache sencillo:

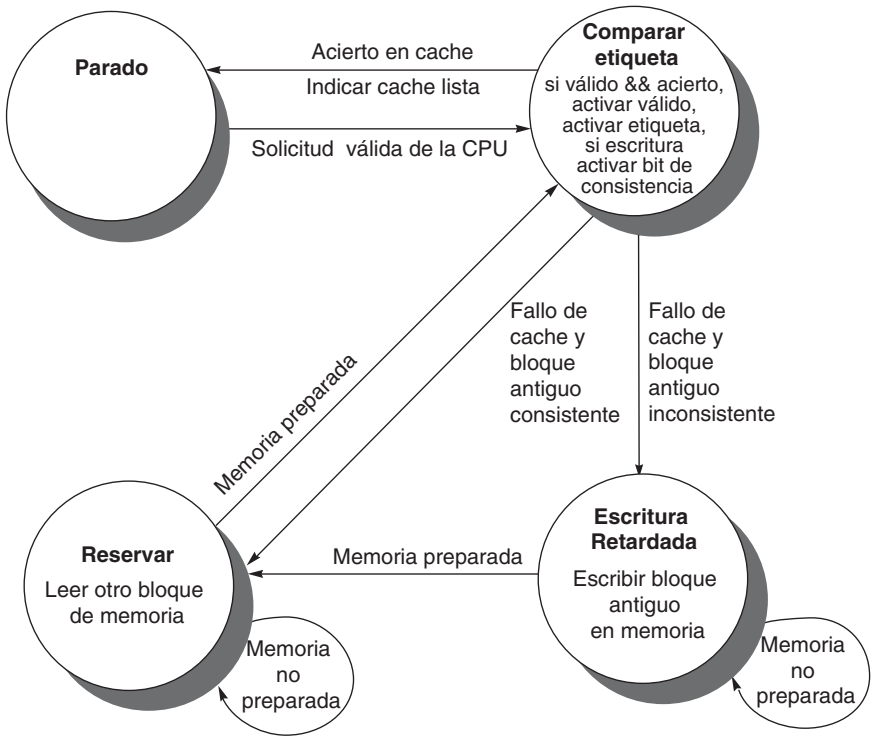


FIGURA 5.34 Los cuatro estados del controlador sencillo.

- *Parado*: Este estado está esperando por una solicitud de lectura o escritura válida por parte del procesador, que hace que la MEF cambie al estado Comparar Etiqueta.
- *Comparar Etiqueta*: Como su nombre indica, en este estado se comprueba si la lectura o escritura es un acierto o un fallo. La parte índice de la dirección selecciona la etiqueta a comparar. Si es válida y la parte de etiqueta de la dirección es igual a la etiqueta en la cache, se produce un acierto. En este caso, el dato se lee o escribe en la palabra de cache seleccionada y se activa la señal de Cache Preparada. Si es una operación de escritura se activa el bit de consistencia. Obsérvese que un acierto de escritura activa también el bit de validez y el campo de etiqueta; aunque parece innecesario, se incluye por que la etiqueta es una memoria sencilla y para cambiar el bit de consistencia es necesario cambiar el bit de validez y la etiqueta. Si es un acierto y el bloque es válido, la MEF vuelve al estado parado. En caso de fallo, primero se actualiza la etiqueta de la cache y después se pasa la estado Escritura Retardada o al estado Reservar en función del valor del bit de consistencia.

- *Escritura Retardada*: Durante este estado se escribe el bloque de 128 bits en memoria utilizando la dirección que se obtiene al componer la etiqueta y el índice de cache. Se permanece en este estado esperando por la señal de memoria preparada. Cuando se completa la escritura, la MEF pasa al estado Reservar.
- *Reservar*: El nuevo bloque se busca en la memoria. Se permanece en este estado esperando a que se active la señal de Memoria Preparada. Cuando se completa la lectura de memoria, la MEF vuelve al estado Comparar Etiqueta. Aunque se podría definir un nuevo estado para completar la operación, en lugar de reutilizar el estado Compara Etiqueta, hay mucho solape entre estados, incluyendo la actualización de la palabra apropiada en el bloque en caso de escritura.

Este sencillo modelo se puede extender fácilmente añadiendo más estados para mejorar las prestaciones. Por ejemplo, en el estado Comparar Etiqueta se hace la comparación y la lectura o escritura de la cache en un ciclo de reloj. A menudo la comparación y el acceso a la cache se hacen en estados diferentes para mejorar duración del ciclo del reloj. Otra optimización consistiría en añadir un búfer de escritura de forma que se podría guardar el bloque inconsistente y leer el nuevo bloque en primer lugar, de esta forma el procesador no tiene que esperar dos accesos a memoria en caso de un fallo con bloque inconsistente. La cache escribiría entonces el bloque inconsistente desde el búfer mientras el procesador opera con el dato solicitado.

La **sección 5.9**, en el CD, proporciona más detalles sobre la MEF, mostrando el controlador completo en un lenguaje de descripción hardware y un diagrama de bloques de esta cache sencilla.

5.8 Paralelismo y jerarquías de memoria: coherencia de cache

Dado que un multiprocesador multinúcleo tiene varios procesadores en un mismo chip, esto procesadores muy probablemente van a compartir un espacio de direcciones físicas común. El almacenamiento en cache de datos compartidos introduce un nuevo problema, porque la visión de la memoria que tienen dos procesadores diferentes es la que tienen de sus memorias caches individuales, lo que, si no se toma ninguna precaución, podría significar que ven dos valores diferentes. La figura 5.35 ilustra este problema y muestra como dos procesadores diferentes pueden tener dos valores diferentes para la misma posición de memoria principal. Este problema se conoce habitualmente como *problema de coherencia de cache*.

De forma informal, podemos decir que un sistema de memoria es coherente si cualquier lectura de un dato devuelve el valor más recientemente modificado de ese dato. Esta definición, aunque intuitiva, es vaga y simplista; la realidad es mucho más compleja. Esta definición simple contiene dos aspectos diferentes del comportamiento del sistema de memoria críticos para una escritura correcta de programas con memoria compartida. El primer aspecto, llamado *coherencia*, define *que valores* deben obtenerse con una lectura.

Instante de tiempo	Suceso	Contenido de la cache de la CPU A	Contenido de la cache de la CPU B	Contenido de la posición de memoria X
0				0
1	CPU A lee X	0		0
2	CPU B lee X	0	0	0
3	CPU A almacena 1 en X	1	0	1

FIGURA 5.35 Problema de la coherencia de cache para una única posición de memoria (X), lectura y escritura por parte de dos procesadores (A y B). Inicialmente asumimos que ninguna cache tiene la variable y que X tiene el valor 0. Suponemos una cache de escritura directa; una cache de escritura retardada añade alguna complicación, pero es similar. Después de que el procesador A escriba el valor X, la cache de A y la memoria tienen el mismo valor, pero la cache de B no, y si B lee la variable X, ¡obtendrá el valor 0!

El segundo aspecto, llamado *consistencia*, determina *cuando* un valor que se escribe será devuelto en una lectura.

Comencemos por la coherencia. Un sistema de memoria es coherente si

1. La lectura de un procesador P en una posición de memoria X después de que P haya escrito en X, sin que haya ninguna otra escritura en la posición X por parte de otro procesador entre la escritura y la lectura de P, siempre devuelve el valor escrito por P. Así, en la figura 5.35, si la CPU A leyese X después del instante de tiempo 3, debería obtener el valor 1.
2. La lectura de un procesador en la posición de memoria X después de que otro procesador haya escrito en X, devuelve el valor escrito si la lectura y la escritura están suficientemente separadas en el tiempo y no hay ninguna otra escritura en X entre estos dos accesos. De este modo, en la figura 5.35 necesitamos un mecanismo para que el valor 0 en la cache de la CPU B sea reemplazado por el valor 1 una vez que la CPU A almacena este valor en la posición de memoria X en el instante 3.
3. Las escrituras en una misma posición se hacen *en serie*; es decir, dos escrituras en la misma posición por dos procesadores cualesquiera son vistas en el mismo orden por todos los procesadores. Por ejemplo, si la CPU B almacena el valor 2 en la posición de memoria X después del instante 3, los procesadores no pueden nunca leer el valor 2 de X y más tarde leer el valor 1.

La primera propiedad simplemente mantiene el orden del programa; por supuesto, esperamos siempre que esta propiedad se mantenga, por ejemplo, en sistemas con un solo procesador. La segunda propiedad define lo que significa tener una visión coherente de la memoria: si un procesador pudiese leer continuamente un valor antiguo, claramente diríamos que la memoria es incoherente.

La necesidad de *escrituras en serie* es más sutil, pero igualmente importante. Suponer que las escrituras no se hacen en serie y que el procesador P1 escribe en la posición X, seguido de una escritura en la misma posición por parte del procesador P2. Forzar a que las escrituras sean en serie asegura que cualquier procesador verá la escritura de P2 en algún momento. Si no se hacen en serie, podría darse el caso de que un procesador vea la escritura de P2 en primer lugar y después la de

P1, manteniendo el valor escrito por P1 de forma indefinida. La forma más sencilla para evitar esta dificultad es asegurar que todas las escrituras a una misma posición se ven en el mismo orden; esta propiedad se llama *escrituras en serie*.

Esquemas básicos para forzar la coherencia

En multiprocesadores con coherencia de cache, las caches proporcionan tanto *migración* como *replicación* de datos compartidos:

- *Migración*: Un dato puede moverse a una cache local y utilizarse ahí de una forma transparente. La migración reduce la latencia de acceso a los datos compartidos que están almacenados de forma remota y la demanda de ritmo de transferencia en la memoria compartida.
- *Replicación*: Cuando los datos compartidos se leen simultáneamente por parte de varios procesadores, se hace una copia del dato en las caches locales. La replicación reduce la latencia de acceso y la disputa en la lectura de un dato compartido.

La migración y la replicación son críticas en las prestaciones del acceso a datos compartidos, por eso muchos multiprocesadores incluyen un protocolo hardware para mantener las caches coherentes. Los protocolos para mantener la coherencia entre varios procesadores se denominan *protocolos de coherencia de cache*. La clave para implementar un protocolo de coherencia de cache es rastrear si los bloques de datos están o no compartidos.

El protocolo de coherencia de cache más popular es *fisgoneo (snooping)*. Cada cache tiene una copia del bloque de memoria física y una copia de su estado de compartición, pero no se mantiene una información centralizada. Las caches son accesibles mediante algún medio de transmisión (un bus o una red), y todos los controladores de cache monitorizan o *fisgan (snoop)* el medio para determinar si tiene una copia de un bloque que está siendo solicitada en un bus o conmutador de acceso.

En la siguiente sección explicamos la implementación de la coherencia de cache basada en fisgoneo con un bus compartido, pero para la implementación de un esquema de coherencia cache basado en fisgoneo puede utilizarse cualquier medio que transmita los fallos de cache a todos los procesadores. Esta transmisión a todas las caches hace que los protocolos de fisgoneo sean fáciles de implementar pero limitan su escalabilidad.

Protocolos de fisgoneo (*snooping*)

Un método para forzar la coherencia es asegurar que un procesador tenga acceso exclusivo a los datos antes de modificarlos con una operación de escritura. Esta forma de actuar se llama *protocolo de invalidación por escritura* porque invalida las copias en otras caches cuando hay una escritura. El acceso exclusivo asegura que cuando se produce una escritura no hay otras copias del dato: todas las copias almacenadas en las caches se han invalidado.

La figura 5.36 muestra un ejemplo de protocolo de invalidación para un bus compartido con fisgoneo y caches de escritura retardada. Para comprender como se asegura la coherencia con este protocolo, consideremos una escritura seguida

de una lectura por parte de otro procesador: como la escritura requiere acceso exclusivo, cualquier copia en la cache del procesador que hace la lectura debe ser invalidada (de ahí el nombre del protocolo). De este modo, cuando se realiza la lectura, se produce un fallo en la cache y se busca una nueva copia del dato. En caso de escritura, el procesador que escribe tiene el acceso exclusivo y se evita así que otro procesador pueda escribir simultáneamente. Si dos procesadores intentan escribir el mismo dato simultáneamente, uno de los dos ganará la carrera, y la copia del otro procesador se invalidará. Para que el segundo procesador complete su escritura, debe obtener una nueva copia del dato, que ya tendrá el valor actualizado. Así, este protocolo fuerza también las escrituras en serie.

Actividad del procesador	Actividad del bus	Contenido de la cache de la CPU A	Contenido de la cache de la CPU B	Contenido de la posición de memoria X
				0
CPU A lee X	Fallo de cache para X	0		0
CPU B lee X	Fallo de cache para X	0	0	0
CPU A escribe 1 en X	Invalidación de X	1		0
CPU B lee X	Fallo de cache para X	1	1	1

FIGURA 5.36 Ejemplo de un protocolo de invalidación que funciona con un bus de fisgoneo para un bloque de cache (X) con caches de escritura retardada. Suponemos que inicialmente ninguna cache tiene el valor X y que valor de X en memoria vale 0. Se muestran los contenidos de la CPU y la memoria una vez completada la actividad del procesador y el bus. Un celda en blanco indica que no hay actividad o no hay una copia en cache. Cuando se produce el segundo fallo del procesador B, la CPU A responde proporcionando el valor solicitado y cancelando el acceso a memoria. Además, se actualizan el contenido de la cache de B y el valor de X en memoria. Esta actualización de memoria, que ocurre cuando un bloque se comparte, simplifica el protocolo, pero permite rastrear el propietario y forzar que la escritura retardada se realice solo cuando se reemplaza el bloque. Para ello, se requiere la introducción de un estado adicional, llamado "propietario", que indica que un bloque puede compartirse, pero el procesador propietario es el responsable de actualizar cualquier otro procesador y la memoria cuando cambia o reemplaza el bloque.

El tamaño del bloque juega un papel importante en la coherencia de cache. Por ejemplo, supongamos fisgoneo en una cache con bloques de ocho palabras, con escrituras y lecturas alternativas en una palabra por parte de dos procesadores. La mayoría de los protocolos intercambian bloques completos entre los procesadores, incrementando así las necesidades de ancho de banda de la coherencia.

Bloques grandes pueden también causar lo que se llama **compartición falsa**: cuando dos variables compartidas no relacionadas están en el mismo bloque de cache, se intercambia el bloque completo entre los procesadores incluso aún cuando están accediendo a variables diferentes. Los programadores y los compiladores deben colocar los datos cuidadosamente para evitar la compartición falsa.

Interfaz hardware software

Compartición falsa: cuando dos variables compartidas no relacionadas están en el mismo bloque de cache y el bloque completo se intercambia entre procesadores aún cuando los procesadores están accediendo a variables diferentes.

Extensión: Aunque las tres propiedades de la página 535 son suficientes para asegurar la coherencia, la cuestión de cuando un valor modificado será visible es también importante. Para ver por qué, observe que no podemos pretender que una lectura de X en la figura 5.35 inmediatamente vea el valor escrito en X por algún otro procesador. Por ejemplo, si se produce una escrita de X seguida de una lectura de X por otro procesador con una separación temporal muy pequeña, puede ser imposible asegurar que el valor obtenido en la lectura sea el valor escrito, porque el valor escrito puede que todavía no haya salido del procesador. La determinación de exactamente *cuando* un valor que se escribe debe ser visible para los que quieran leerlo se define mediante un *modelo de consistencia de memoria*.

Hacemos las dos siguientes suposiciones. Primero, una escritura no termina (y permite que se comience la siguiente escritura) hasta que todos los procesadores han visto los efectos de la escritura. Segundo, el procesador no cambia el orden de la escritura con respecto a cualquier otro acceso a memoria. Estas dos condiciones implican que si un procesador escribe en una posición X y después en una posición Y, cualquier otro procesador que ve el nuevo valor de Y debe ver también el nuevo valor de X. Estas restricciones permiten al procesador reordenar las lecturas, pero fuerzan a que las escrituras se terminen en el orden del programa.

Extensión: El problema de la coherencia de cache en multiprocesadores y E/S (véase capítulo 6), aunque similar en su origen, tiene diferentes características que afectan a las soluciones adecuadas. Al contrario que en la E/S, donde son raras las copias múltiples de un dato, y deben evitarse cuando sea posible, un programa que se ejecuta en varios procesadores normalmente tendrá varias copias del mismo dato en varias caches.

Extensión: Además del protocolo de coherencia de cache basado en fisgoneo, donde el estado de los bloques compartidos está distribuido, el *protocolo de coherencia de cache basado en directorio* mantiene el estado de compartición de un bloque de la memoria física en una localización, llamada *directorio*. La implementación de la coherencia basada en directorio tiene un sobrecoste ligeramente mayor que el fisgoneo, pero puede reducir el tráfico entre las caches y, por lo tanto, escalar a un mayor número de procesadores.

Material avanzado: implementación de controladores de cache

En esta sección del CD se muestra cómo implementar el control de una cache, de modo similar a como se implementó el control de los caminos de datos de ciclo único y segmentado en el capítulo 4. Esta sección comienza con una descripción de máquinas de estados finitos y la implementación del controlador de cache de una cache de datos sencilla, incluyendo la descripción del controlador en un lenguaje de descripción hardware. Después se profundiza en los detalles de un protocolo de coherencia de cache de ejemplo y en las dificultades que presenta su implementación.

5.10 Casos reales: las jerarquías de memoria del AMD Opteron X4 (Barcelona) y del Intel Nehalem

En esta sección describiremos la jerarquía de memoria de dos microprocesadores modernos: el AMD Opteron X4 (Barcelona) y el Intel Nehalem. La figura 5.37 muestra la foto del dado de silicio del Intel Nehalem, y la figura 1.9, en el capítulo 1, muestra la foto del dado de silicio del AMD Opteron X4. Ambos procesadores tienen caches de segundo y tercer nivel en el mismo procesador. Esta integración reduce el tiempo de acceso a las caches de niveles inferiores y el número de pines del chip, ya que no existe la necesidad de un bus para acceder a una cache externa de segundo nivel. Ambos procesadores tienen los controladores de memoria integrados en el chip, lo que reduce la latencia en los accesos a la memoria.

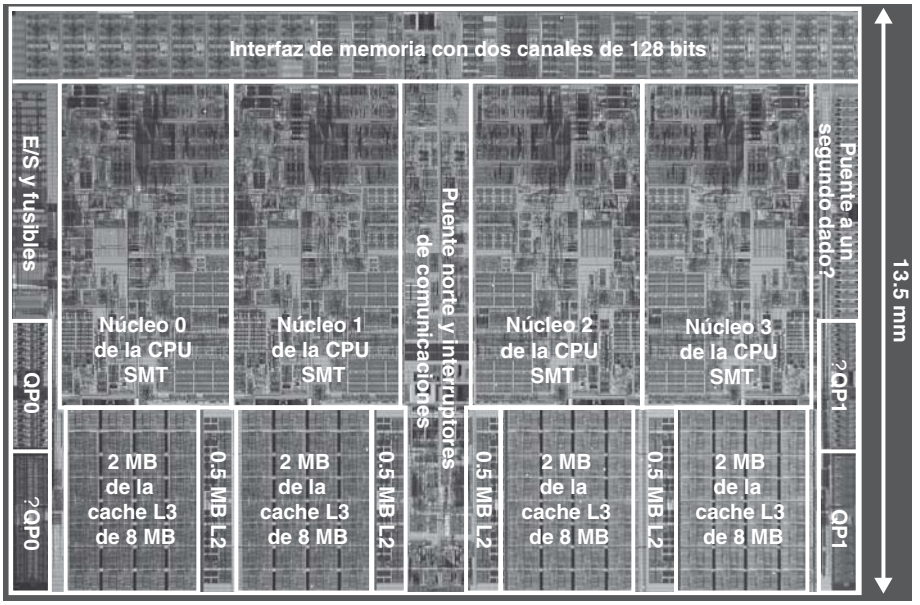


FIGURA 5.37 Foto del dado del procesador Intel Nehalem con sus distintas partes etiquetadas. Este dado de 13.5 mm x 19.6 mm tiene 731 millones de transistores. Contiene cuatro procesadores cada uno con caches privadas de instrucciones de 32 KB y de datos de 32 KB, y una cache L2 de 512 MB. Los cuatro núcleos comparten una cache L3 de 8 MB. Los dos canales de memoria de 128 bits son una interfaz con una memoria DDR DRAM. Cada núcleo tiene una TLB de dos niveles. El controlador de memoria está integrado en el dado, es decir, no hay puentes norte separados como en el procesador Intel Clovertown.

Las jerarquías de memoria del Nehalem y el Opteron

La figura 5.38 resume el tamaño de las direcciones y las TLB de los dos procesadores. Observe que el AMD Opteron X4 (Barcelona) tiene 4 TLB y las direcciones virtuales y físicas no coinciden con el tamaño de palabra. El X4 implementa sólo 48 de los 64 bits potenciales de su espacio virtual y 48 de los 64 bits potenciales de su espacio de direcciones físico. Nehalem tiene tres TLB, la dirección virtual es de 48 bits y la dirección física de 44 bits.

Característica	Intel Nehalem	AMD Opteron X4 (Barcelona)
Dirección virtual	48 bits	48 bits
Dirección física	44 bits	48 bits
Tamaño página	4 KB, 2/4 MB	4 KB, 2/4 MB
Organización del TLB	1 TLB para instrucciones y 1 TLB para datos por núcleo Ambos TLB L1 son asociativos por conjuntos de cuatro vías y con reemplazo LRU El TLB L2 es asociativo por conjuntos de cuatro vías y con reemplazo LRU El I-TLB L1 tiene 128 entradas para páginas pequeñas y 7 por hilo para páginas grandes El D-TLB L1 tiene 64 entradas para páginas pequeñas y 32 para páginas grandes El TLB L2 tiene 128 entradas Los fallos de TLB se procesan por hardware	1 TLB L1 para instrucciones y 1 TLB L1 para datos por núcleo Ambos TLB L1 son completamente asociativos y con reemplazo LRU 1 TLB L2 para instrucciones y 1 TLB L2 para datos por núcleo Ambos TLB L2 son asociativos por conjuntos de cuatro vías y con reemplazo de turno rotatorio Los dos TLB L1 tienen 48 entradas Los dos TLB L2 tienen 512 entradas Los fallos de TLB se procesan por hardware

FIGURA 5.38 Hardware para la traducción de direcciones y el TLB en el Intel Nehalem y el AMD Opteron X4. El tamaño de palabra determina el tamaño máximo de la dirección virtual, pero el procesador no utiliza todos los bits. Ambos procesadores incluyen soporte para páginas grandes, que son utilizadas, por ejemplo, por el sistema operativo o por las asignaciones de un búfer de marcos de página. Las páginas grandes evitan evitan utilizar un número elevado de entradas para asignar un único objeto que siempre está presente. El procesador Nehalem soporta dos hilos con apoyo hardware por núcleo (véase sección 7.5 en el capítulo 7).

La figura 5.39 muestra las caches de los procesadores. Cada procesador en el X4 tiene caches L1 propias de datos e instrucciones de 64 KB y su propia cache L2 de 512 KB. Los cuatro procesadores comparte una única cache L3 de 2 MB. El procesador Nehalem tiene una estructura parecida, con caches L1 propias de datos e instrucciones de 32 KB y una cache L2 propia de 512 KB, en cada procesador, y los 4 procesadores comparten una única cache L3 de 8MB.

La figura 5.40 muestra el CPI, frecuencia de fallos por mil instrucciones para las caches L1 y L2, y los accesos a la DRAM por mil instrucciones para el Opteron X4 ejecutando los programas de prueba del SPECint 2006. Observe que el CPI y la frecuencia de fallos están altamente correlacionados y el coeficiente de correlación es 0.97. Aunque no se dispone de los datos de fallos de L3, podemos inferir la efectividad de la L3 por la reducción en los accesos a la DRAM frente a los fallos de la L2. Unos cuantos programas se benefician de la cache L3 de 2 MB, h264avc, hmmer y bzip, pero la mayoría no se beneficia.

Características	Intel Nehalem	AMD Opteron X4 (Barcelona)
Organización de la cache L1	Caches de datos en instrucciones separadas	Caches de datos en instrucciones separadas
Tamaño de la cache L1	32 KB datos/instrucciones por núcleo	64 KB datos/instrucciones por núcleo
Asociatividad en cache L1	Asociativa por conjuntos de 4 vías (I), 8 vías (D)	Asociativa por conjuntos de 2 vías
Reemplazamientos en cache L1	Reemplazo LRU aproximado	Reemplazo LRU
Tamaño de bloque en la L1	64 bytes	64 bytes
Política de escritura en la L1	Escritura directa con reserva de escritura	Escritura directa con reserva de escritura
Tiempo de acierto en la L1 (load)	No disponible	3 ciclos
Organización de la cache L2	Unificada (instrucciones y datos) por núcleo	Unificada (instrucciones y datos) por núcleo
Tamaño de la cache L2	256 KB (0.25 MB)	512 KB (0.5 MB)
Asociatividad en cache L2	Asociativa por conjuntos de 8 vías	Asociativa por conjuntos de 16 vías
Reemplazamientos en cache L2	Reemplazo LRU aproximado	Reemplazo LRU aproximado
Tamaño de bloque en la L2	64 bytes	64 bytes
Política de escritura en la L2	Escritura directa con reserva de escritura	Escritura directa con reserva de escritura
Tiempo de acierto en la L2 (load)	No disponible	9 ciclos
Organización de la cache L3	Unificada (instrucciones y datos)	Unificada (instrucciones y datos)
Tamaño de la cache L3	8192 KB (8 MB), compartida	2048 KB (2 MB), compartida
Asociatividad en cache L3	Asociativa por conjuntos de 16 vías	Asociativa por conjuntos de 32 vías
Reemplazamientos en cache L3	No disponible	Reemplaza bloque compartido por menos núcleos
Tamaño de bloque en la L3	64 bytes	64 bytes
Política de escritura en la L3	Escritura directa con reserva de escritura	Escritura directa con reserva de escritura
Tiempo de acierto en la L3 (load)	No disponible	38 (?) ciclos

FIGURA 5.39 Caches de primer, segundo y tercer nivel en los procesadores Intel Nehalem y AMD Opteron X4 (Barcelona).

Técnicas para reducir la penalización por fallos

Ambos procesadores incorporan optimizaciones adicionales que les permiten reducir la penalización por fallos. La primera de estas optimizaciones es que en caso de fallo la palabra solicitada se devuelve antes, como se describe en la *Extensión* de la página 473. Ambos procesadores permiten que durante un fallo de cache se continúe ejecutando instrucciones que acceden a la cache de datos. Esta técnica, denominada **cache no bloqueante**, es ampliamente utilizada por los diseñadores que intentan esconder la latencia de los fallos de cache en procesadores con ejecución fuera de orden. Existen dos tendencias en la implementación de la cache no bloqueante. *Acierto después de fallo* permite que se produzcan aciertos adicionales en la cache durante un fallo, mientras que *fallo después de fallo* permite que varios fallos de caches estén pendientes de resolverse. El propósito del primero es ocultar la latencia del fallo haciendo alguna otra operación, mientras que el propósito del segundo es solapar la latencia de dos fallos diferentes.

Cache no bloqueante: cache que permite que el procesador continúe realizando accesos a la cache mientras se está procesando un fallo anterior.

El solapamiento de una gran parte del tiempo de fallo cuando existen muchos fallos pendientes necesita un sistema de memoria con un alto ritmo de transferen-

Nombre	CPI	Cache L1 D fallos/1000 instr.	Cache L2 D fallos/1000 instr.	DRAM accesos/1000 instr.
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
onmetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
median	1.35	13.6	7.5	5.4

FIGURA 5.40 CPI, frecuencia de fallos y accesos a la DRAM en la jerarquía de memoria del procesador Opteron modelo X4 2356 (Barcelona) ejecutando los programas de prueba SPECint2006. Desafortunadamente, el contador de fallos de L3 no funciona en este procesador, así sólo disponemos de los accesos a la DRAM para determinar la efectividad de la L3. Observe esta figura utiliza el mismo sistema y programas de prueba que la figura 1.20 en el capítulo 1.

cia capaz de procesar varios fallos en paralelo. En los sistemas de sobremesa, la memoria sólo es capaz de aprovechar de una forma limitada esta capacidad, pero los grandes servidores y los sistemas multiprocesador disponen de sistemas de memoria que pueden procesar muchos fallos en paralelo.

Ambos microprocesadores prebuscan instrucciones y disponen de un mecanismo hardware integrado para acceder a los datos. Analizan el patrón de fallos de datos y utilizan esta información para intentar predecir la siguiente dirección y comenzar la búsqueda del dato antes de que ocurra el fallo. Estas técnicas generalmente funcionan mejor cuando se accede a matrices de datos en el interior de un lazo.

Un reto importante con el que se enfrentan los diseñadores de caches es soportar procesadores como el Nehalem y el Opteron X4, que pueden ejecutar varias instrucciones de acceso a memoria por ciclo de reloj. Las peticiones múltiples pueden ser resueltas en el primer nivel de cache mediante dos técnicas diferentes. La cache puede ser multipuerto, permitiendo varios accesos simultáneos al mismo bloque de cache. Sin embrago, las caches multipuerta son a menudo demasiado caras porque las celdas de RAM en una memoria multipuerta son mucho más grandes que en una memoria de una sola puerta. Un esquema alternativo es dividir la cache en varios bancos y permitir múltiples accesos independientes, siempre que estos accesos sean a bancos diferentes. Esta técnica es similar a la memoria principal entrelazada (véase figura 5.11). La cache L1 del Opteron X4 soporta dos lecturas de 128 bits por ciclo y tiene ocho bancos.

El procesador Nehalem y la mayoría de los procesadores siguen la política de *inclusión* en su jerarquía de memoria. Esto significa que en los niveles inferiores (niveles más alejados del procesador) habrá una copia de todos los datos de los niveles superiores (niveles más próximos al procesador). Por el contrario, el procesador AMD sigue la política de *exclusión* en las cache de primero y segundo

nivel, que significa que un bloque de cache puede encontrase en el primero o en el segundo nivel, pero no en ambos. Así, cuando en un fallo de L1 un bloque se transfiere desde la L2 a la L1, el bloque reemplazado se envía de nuevo a la L2.

Las sofisticadas jerarquías de memoria de estos procesadores y el elevado porcentaje del dado que se destina a las caches y la TLB pone de manifiesto los esfuerzos que se están haciendo para reducir la diferencia entre el ciclo de reloj del procesador y la latencia de la memoria.

Extensión: La cache L3 compartida del Opteron no siempre sigue el principio de exclusión. Como en la L3 los bloques de datos pueden estar compartidos por varios procesadores, sólo se elimina un bloque de la L3 si no hay otros procesadores que lo estén utilizando. De este modo, el protocolo de la cache L3 reconoce si un bloque está compartido o esta siendo usado por un único procesador.

Extensión: Como el procesador Opteron X4 no sigue el principio de inclusión tradicional, tiene una novedosa relación entre los niveles de la jerarquía de memoria. En lugar de que la memoria alimente al nivel L2 y este al nivel L1, la cache del nivel L2 sólo mantiene bloques que se han eliminado de la cache L1. De este modo, la cache L2 puede ser llamada *cache de víctimas*, porque sólo tiene bloques desplazados desde la L1 ("víctimas"). De modo similar, la L3 es una cache de víctimas para la cache L2, sólo almacena bloques que han desbordado de la cache L2. Si en un fallo de L1, el bloque no se encuentra en la L2, pero se encuentra en la cache L3, la cache L3 envía el bloque directamente a la L1. De esta forma, en un fallo en la L1 se puede recibir el bloque desde un acierto en la L2 o en la L3 o desde memoria.

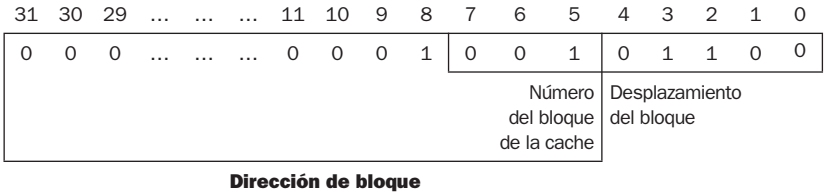
5.11 Falacias y errores habituales

Como uno de los aspectos de naturaleza más cuantitativa de la arquitectura de los computadores, podría parecer que la jerarquía de memoria es menos vulnerable a las falacias y errores. No solamente se han propagado falacias y se han encontrado errores, sino que algunos de ellos han tenido importantes consecuencias negativas. Comenzaremos con un error que cometen muchos estudiantes en los ejercicios y exámenes.

Error: olvidar que se debe tener en cuenta el direccionamiento de byte o el tamaño de bloque de la cache cuando se simula una cache.

Cuando se simula una cache (a mano o con computador), necesitamos estar seguros de que tenemos en cuenta el efecto del direccionamiento de byte y de los bloques multipalabra cuando se determina qué bloque de la cache se corresponde con una dirección determinada. Por ejemplo, si tenemos una cache de correspondencia directa de 32 bytes con un tamaño de bloque de 4 bytes, la dirección de byte 36 se corresponde con el bloque 1 de la cache, ya que la dirección de byte 36 es la dirección de bloque 9 y (9 módulo 8) = 1. Por otra parte, si la dirección 36 es una dirección de palabra, entonces se corresponde con el bloque (36 mod 8) = 4. Asegúrese de que el problema indica claramente qué es lo que referencia la dirección.

De forma parecida, debemos tomar en consideración el tamaño del bloque. Suponga que tenemos una cache con 256 bytes y un tamaño de bloque de 32 bytes. ¿En qué bloque cae la dirección de byte 300? Si dividimos la dirección 300 en campos, podemos conocer la respuesta:



La dirección de byte 300 se corresponde con la dirección de bloque

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

El número de bloques en la cache es

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

El bloque número 9 cae en el bloque de cache número (9 módulo 8) = 1.

Este error lo comete mucha gente, incluidos los autores (en los borradores previos), así como los profesores que olvidan si las direcciones hacen referencia a números de palabras, bytes o bloques. Recuerde este fallo cuando esté planteando ejercicios.

Error: ignorar el comportamiento del sistema de memoria cuando se programa o cuando se genera código en un compilador.

Lo siguiente se podría escribir fácilmente como falacia: "Los programadores pueden ignorar las jerarquías de memoria cuando están programando." Ilustramos esto con un ejemplo que utiliza la multiplicación de matrices, con el objetivo de complementar la comparación de los algoritmos de clasificación de la figura 5.18.

Este es el lazo más interno de la versión de la multiplicación de matrices del capítulo 3:

```
for (i=0; i!=500; i=i+1)
   for (j=0; j!=500; j=j+1)
      for (k=0; k!=500; k=k+1)
         x[i][j] = x[i][j] + y[i][k] * z[k][j];
```

Cuando se ejecuta con datos de entrada que corresponden a matrices de 500 × 500 en doble precisión, el tiempo de ejecución de la CPU del lazo mostrado más arriba en una CPU de tipo MIPS con una cache de segundo nivel de 1 MB fue alrededor de la mitad del tiempo cuando se compara con la situación en la que el orden del lazo cambia a `k,j,i` (tal que `i` es el más interno). La única diferencia se encuentra en cómo el programa accede a memoria y el consiguiente efecto sobre la jerarquía de memoria. Optimizaciones adicionales del compilador que usan una técnica denominada *agrupación en bloques* pueden hacer que el tiempo de ejecución de este código sea otras cuatro veces más rápido.

Error: La asociatividad de una cache compartida es menor que el número de núcleos o hilos que comparten la cache.

Si no se pone una atención especial, un programa paralelo ejecutándose en 2^n procesadores o hilos puede fácilmente situar estructuras de datos en direcciones que podrían enviarse al mismo conjunto de una cache L2 compartida. Si la asociatividad de la cache es de al menos 2^n vías, el hardware oculta estos conflicto accidentales al programador. En caso contrario, programador podría encontrar comportamientos extraños en las prestaciones, realmente debidos a fallos de conflicto en la L2, al pasar por ejemplo de un diseño con 16 núcleos a uno con 32 núcleos si ambos utilizan caches L2 asociativas por conjuntos de 16 vías.

Error: utilización del tiempo promedio de acceso a memoria para evaluar la jerarquía de memoria de un procesador fuera de orden.

Si un procesador se para durante un fallo de cache, entonces se puede calcular separadamente el tiempo de parada debido a la memoria y el tiempo de ejecución del procesador, y así evaluar independientemente la jerarquía de memoria utilizando el tiempo promedio de acceso a memoria (véase página 478).

Si el procesador continúa ejecutando instrucciones, e incluso puede soportar más fallos de cache durante un anterior fallo de cache, entonces la única valoración precisa de la jerarquía de memoria se consigue simulando el procesador fuera de orden junto con la jerarquía de memoria.

Error: ampliar un espacio de direccionamiento añadiendo segmentos encima de un espacio de direccionamiento no segmentado.

Durante los años 70, muchos programas crecían tanto que no todo el código y los datos podían ser direccionados con sólo direcciones de 16 bits. Entonces, los computadores fueron modificados para ofrecer direcciones de 32 bits, bien a través de un espacio de direccionamiento no segmentado de 32 bits (también denominado *espacio de direccionamiento plano*), bien añadiendo 16 bits de segmento a un dirección existente de 16 bits. Desde el punto de vista del marketing, añadir segmentos que fueran visibles al programador y que forzara al programador y compilador a descomponer programas en segmentos podría resolver el problema de direccionamiento. Desafortunadamente, existen dificultades cada vez que un lenguaje de programación desea una dirección que es superior a un segmento, tal como los índices de las matrices grandes, los punteros no restringidos o los parámetros de referencia. Además, añadir segmentos puede convertir cada dirección en dos palabras –una para el número de segmento y otra para el desplazamiento de segmento– causando problemas cuando se usan direcciones en registros.

Error: Implementar un monitor de máquina virtual en una arquitectura del repertorio de instrucciones que ha sido diseñada para ser virtualizada.

Mucho diseñadores en los años 70 y 80 no fueron cuidadosos para asegurarse que todas las instrucciones que leen o escriben información relacionada con los recursos hardware fueran privilegiadas. Esta actitud descuidada causa problemas ahora en los MMV para todas esas arquitecturas, incluyendo la x86, que usamos aquí como ejemplo.

La figura 5.41 describe 18 instrucciones que causan problemas en la virtualización [Robin and Irvine, 2000]. Los dos tipos más amplios son instrucciones que

- Leen registros de control en modo usuario, lo que revela que el sistema operativo invitado está ejecutándose en una máquina virtual.
- Comprueba la protección, tal como se requiere en las arquitecturas de memoria virtual segmentadas, pero asume que el sistema operativo se está ejecutando con los privilegios del nivel más alto.

Para simplificar la implementación de MMV en el x86, tanto AMD como Intel han propuesto extensiones a la arquitectura a través de nuevos modos. El VT-x de Intel proporciona un nuevo modo de ejecución para las MV, una definición en la arquitectura del estado de la MV, instrucciones para cambiar MV de forma rápida, y un extenso conjunto de parámetros para seleccionar las circunstancias dónde debe ser invocada una MMV. Además, el VT-x añade 11 nuevas instrucciones para el x86. El Pacifica de AMD propone una propuesta similar.

Una alternativa a la modificación del hardware es hacer pequeñas modificaciones en el sistema operativo para evitar las partes molestas de la arquitectura. Esta técnica, se llama *paravirtualización*, y código abierto Xen VMM es un buen ejemplo. El Xen VMM proporciona un SO invitado con un abstracción de la máquina virtual que usa únicamente las partes fáciles de virtualizar del hardware x86 en el que se ejecuta la MV.

Categoría del problema	Instrucciones x86 con problemas
Acceso a registros sensibles sin utilizar excepciones cuando se está ejecutando en modo usuario	Almacenar el registro de tabla de descriptores global (SGDT) Almacenar el registro de tabla de descriptores local (SLDT) Almacenar el registro de tabla de descriptores de interrupción (SIDT) Almacenar palabra de estado de la máquina (SMSW) Introducir indicadores (*flags*) (PUSHF, PUSHFD) Extraer indicadores (*flags*) (POPF, POPFD)
Cuando se accede a los mecanismos de memoria virtual en modo usuario, fallan las comprobaciones de protección de las instrucciones x86	Cargar derechos de acceso del descriptor del segmento (LAR) Cargar límites del segmento del descriptor del segmento (LSL) Verificar si se puede leer el descriptor del segmento (VERR) Verificar si se puede modificar el descriptor del segmento (VERW) Introducir en el registro del segmento (POP CS, POP SS, ...) Extraer el registro del segmento (PUSH CS, PUSH SS, ...) Llamar función para cambiar el nivel de privilegio (CALL) Retornar de función de cambio de nivel de privilegio (RET) Saltar a un nivel de privilegio diferente (JMP) Interrupción software (INT) Almacenar registro selector de segmento (STR) Mover a/desde registro de segmento (MOVE)

FIGURA 5.41 Resumen de 18 instrucciones x86 que causan problemas en la virtualización [Robin and Irvine, 2000]. Las cinco primeras instrucciones en grupo superior permiten que un programa en modo usuario lea un registro de control, tal como un registro de tabla de descriptores, sin dar lugar a una excepción. La instrucción de introducir indicadores (*flags*) modifica un registro de control con información sensible pero falla silenciosamente en modo usuario. La comprobación de la protección de una arquitectura la memoria virtual segmentada del x86 es la ruina del grupo inferior, por que cada una de esas instrucciones comprueba implícitamente el nivel de privilegios como parte de la ejecución de la instrucción al leer un registro de control. La comprobación supone que el SO debe estar en el nivel de privilegios más elevado, lo que no ocurre en una MV invitada. Solamente la instrucción mover a registro del segmento intenta modificar el estado de control, y la comprobación de protección la hace fracasar también.

5.12 Conclusiones finales

La dificultad de construir un sistema de memoria que mantenga el ritmo de los procesadores más rápidos es acetuada por el hecho de que la base del material de la memoria principal, las DRAMs, es esencialmente el mismo tanto en los computadores más rápidos como en los más lentos y baratos.

Es el principio de localidad el que nos da una oportunidad para superar la alta latencia del acceso a la memoria, y la solidez de esta estrategia se demuestra en todos los niveles de la jerarquía de memoria. Aunque estos niveles de la jerarquía parezcan bastante diferentes en términos cuantitativos, siguen estrategias similares en sus operaciones y explotan las mismas propiedades de la localidad.

Las caches multinivel hacen posible utilizar otras optimizaciones más fácilmente por dos razones. En primer lugar, los parámetros de diseño de una cache de segundo o tercer nivel son distintos de los de la cache de primer nivel. Por ejemplo, como una cache de bajo nivel será mucho más grande, es posible utilizar tamaños de bloque mayores. En segundo lugar, una cache de segundo o tercer nivel no está siendo usada constantemente por el procesador, como lo está la cache de primer nivel. Esto nos permite tener a la cache de segundo o tercer nivel trabajando en algo cuando se encuentra ociosa, lo cual puede ser útil en la prevención de futuros fallos.

Otra posible dirección consiste en buscar ayuda desde el software. La gestión eficiente de la jerarquía de memoria utilizando una variedad de transformaciones de programa y de facilidades hardware constituye un punto de concentración en la mejora de los compiladores. Dos ideas diferentes están siendo exploradas. Una idea consiste en reorganizar el programa para mejorar su localidad espacial y temporal. Esta técnica se centra en los programas basados en lazos que utilizan grandes matrices como la principal estructura de datos; los problemas grandes del álgebra lineal constituyen un ejemplo típico. Reestructurando los lazos que acceden a las matrices, se puede obtener una sustancial mejora en la localidad, y por lo tanto en las prestaciones de la cache. El ejemplo de la página 544 mostró cómo de efectivo podía ser incluso un pequeño cambio en la estructura del lazo.

Otra dirección consiste en intentar usar **prebúsquedas** dirigidas por el compilador. En la prebúsqueda, un bloque de datos se lleva a la cache antes de que realmente sea referenciado. Muchos microprocesadores utilizan prebúsqueda hardware para intentar predecir accesos que pueden difícilmente predecibles por el software.

Prebúsqueda: técnica en la cual bloques de datos que se necesitan en el futuro son llevados con anticipación a la cache por medio de la utilización de instrucciones especiales que especifican la dirección del bloque.

Una tercera alternativa son las instrucciones conscientes de la cache que optimizan la transferencia de memoria. Por ejemplo, los microprocesadores de la sección 7.10 del capítulo 7 utilizan una optimización que, en caso de fallo de escritura, no busca los contenidos del bloque en memoria, porque el programa va a modificar el bloque completo. Esta optimización reduce de forma significativa el tráfico de memoria.

Como se verá en el capítulo 7, los sistemas de memoria constituyen también un aspecto central en el diseño de los procesadores paralelos. La creciente importancia

de la jerarquía de memoria en la determinación de las prestaciones del sistema significa que esta importante área continuará siendo por algunos años un punto de concentración tanto para diseñadores como para investigadores.

5.13 Perspectiva histórica y lecturas recomendadas

La sección de historia proporciona una visión de conjunto de las tecnologías de las memorias, desde las líneas de retardo de mercurio a las DRAM, pasando por la invención de la jerarquía de memoria y los mecanismos de protección, y memoria virtual, y concluye con una breve historia de los sistemas operativos, incluyendo CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows y Linux.

5.14 Ejercicios

Contribución de Jichuan Chang, Jacob Leverich, Kevin Lim and Parthasarathy Ranganathan (todos de Hewlett-Packard)

Ejercicio 5.1

En este ejercicio se considera la jerarquía de memoria en las aplicaciones mostradas en la siguiente tabla.

a.	Búsquedas en la web
b.	Banca electrónica

5.1.1 [10]<5.1> Suponiendo que tanto los clientes como los servidores están involucrados en el proceso, nombre en primer lugar los sistemas cliente y servidor. ¿Dónde se pueden poner caches para acelerar el proceso?

5.1.2 [10]<5.1> Diseñe una jerarquía de memoria para el sistema. Muestre la capacidad y la latencia típica en varios niveles de la cache. ¿Qué relación hay entre la capacidad de la cache y la latencia de acceso?

5.1.3 [15]<5.1> ¿Cuál es la unidad de transferencia de datos entre niveles de la jerarquía? ¿Cuál es la relación entre posición del dato, tamaño del dato y latencia de la transferencia?

5.1.4 [10]<5.1, 5.2> El ancho de banda en la comunicación y en el procesamiento del servidor son dos factores importantes a tener en cuenta en el diseño de una jerarquía de memoria. ¿Qué anchos de banda pueden ser un factor limitante en este ejemplo? ¿Cómo se pueden mejorar y a qué coste?

5.1.5 [5]<5.1, 5.8> Considere ahora que hay varios clientes accediendo simultáneamente al servidor, ¿se mejorará en este escenario la localidad espacial y temporal?

5.1.6 [10]<5.1, 5.8> Dé un ejemplo en que la cache proporcione datos anticuados. ¿Cómo se puede reducir o evitar este problema?

Ejercicio 5.2

En este ejercicio se explora la localidad de memoria del cálculo con matrices. En el siguiente código C, los elementos de la misma fila se almacenan de forma contigua.

a.
```
for (I=0; I<8000; I++)
  for (J=0; J<8; J++)
    A[I][J]=B[J][0]+A[J][I];
```

b.
```
for (J=0; J<8; I++)
  for (I=0; I<8; I++)
    A[I][J]=B[J][0]+A[J][I];
```

5.2.1 [5]<5.1> ¿Cuántos enteros de 32 bits pueden almacenarse en una línea de cache de 16 bytes?

5.2.2 [5]<5.1> ¿Cuáles de las referencias a variables del código C anterior muestran localidad temporal?

5.2.3 [10]<5.1> ¿Cuáles de las referencias a variables del código C anterior muestran localidad espacial?

La localidad se ve afectada por el orden de las referencias a memoria y por la distribución de los datos. Este mismo cálculo puede escribirse en Matlab, y la diferencia con C es que los elementos de una misma columna se almacenan de forma contigua en memoria.

a.
```
for I=1:8000
  for J=1:8
    A(I,J)=B(J,0)+A(J,I);
  end
end
```

b.
```
for J=1:8
  for I=1:8
    A(I,J)=B(J,0)+A(J,I);
  end
end
```

5.2.4 [10]<5.1> ¿Cuántas líneas de cache de 16 bytes son necesarias para almacenar todos los elementos de 32 bits que se referencian de la matriz?

5.2.5 [5]<5.1> ¿Qué referencias muestran localidad temporal?

5.2.6 [10]<5.1> ¿Qué referencias muestran localidad espacial?

Ejercicios 5.3

Las caches son importantes para conseguir una jerarquía de memoria de altas prestaciones en los procesadores. En la tabla se muestra una lista de referencias a direcciones de memoria de 32 bits

a.	1, 134, 212, 1, 135, 213, 162, 161, 2, 44, 41, 221
b.	6, 214, 175, 214, 6, 84, 65, 174, 64, 105, 85, 215

5.3.1 [10]<5.2> Dada una cache de correspondencia directa con 16 bloques de una palabra indique, para cada una de estas referencias, la dirección binaria, la etiqueta y el índice. Indique también si es un fallo o un acierto, suponiendo que inicialmente la cache está vacía.

5.3.2 [10]<5.2> Dada una cache de correspondencia directa con bloques de dos palabras y un total de ocho bloques indique, para cada una de estas referencias, la dirección binaria, la etiqueta y el índice. Indique también si es un fallo o un acierto, suponiendo que inicialmente la cache está vacía.

5.3.3 [20]<5.2, 5.3> Optimice el diseño de la cache para las referencias anteriores. Hay tres diseños posibles de una cache de correspondencia directa y una capacidad de ocho palabras: C1 tiene bloques de una palabra, C2 bloques de dos palabras y C3 bloques de cuatro palabras. ¿Cuál es el mejor diseño en términos de frecuencia de fallos? Si la parada por fallo es de 25 ciclos y C1 tiene un tiempo de acceso de 2 ciclos, C2 de 3 ciclos y C3 de 5 ciclos, ¿cuál es el mejor diseño?

Hay muchos parámetros diferentes que son importantes en las prestaciones finales de una cache. En la tabla se muestran algunos parámetros para diferentes caches de correspondencia directa.

	Tamaño de la cache de datos	Tamaño del bloque de cache	Tiempo de acceso a la cache
a.	64 KB	1 palabra	1 ciclo
b.	64 KB	2 palabras	2 ciclos

5.3.4 [15]<5.2> Calcule el número total de bits de la cache suponiendo direcciones de 32 bits. Dado este tamaño total, determine el tamaño más próximo al

tamaño dado de la cache de correspondencia directa con bloques de 16 palabras, siendo este tamaño igual o mayor al dado. Explique por qué la segunda cache, a pesar de tener bloques de mayor tamaño, puede proporcionar peores prestaciones que la primera.

5.3.5 [20]<5.2, 5.3> Genere una secuencia de accesos de lectura con una frecuencia de fallos en una cache de 2 KB asociativa por conjuntos de dos vías menor que en la cache de la tabla. Identifique una posible solución para que la frecuencia de fallos de la cache de la tabla sea menor o igual que la de la cache de 2 KB. Discuta las ventajas y desventajas de esta solución.

5.3.6 [15]<5.2> La fórmula de la página 457 muestra la forma típica para indexar una cache de correspondencia directa, específicamente (dirección del bloque) módulo (número de bloques en la cache). Suponiendo direcciones de 32 bits y 1024 bloques en la cache, considere una función diferente, específicamente (dirección del bloque[31:27]) XOR (dirección del bloque[26:22]). ¿Es posible usar esta función para indexar una cache de correspondencia directa? En caso afirmativo, explicar por qué y discuta cualquier cambio que pueda ser necesario hacer en la cache. En caso negativo, explicar por qué.

Ejercicio 5.4

En una cache de correspondencia directa con direcciones de 32 bits, los bits de la dirección se usan como se indica en la tabla.

	Etiqueta	Índice	Desplazamiento
a.	31-10	9-4	3-0
b.	31-12	11-15	4-0

5.4.1 [5]<5.2> ¿Qué tamaño (en palabras) tiene la línea de cache?

5.4.2 [5]<5.2> ¿Cuántas entradas tiene la cache?

5.4.3 [5]<5.2> ¿Cuál es la relación entre el número total de bits de la cache y el número de bits de almacenamiento?

Inmediatamente después de encender el computador, se producen las siguientes referencias a la cache, expresadas como direcciones de byte

Dirección	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.4.4 [10]<5.2> ¿Cuántos bloques se reemplazan?

5.4.5 [10]<5.2> ¿Cuál es la razón de aciertos?

5.4.6 [20]<5.2> Muestre el estado final de la cache, representando cada entrada válida con <índice, etiqueta, dato>.

Ejercicio 5.5

Recuerde que hay dos estrategias de escritura y dos estrategias de reserva de escritura, y que pueden implementarse varias combinaciones tanto en la L1 como en la L2.

	L1	L2
a.	Escritura retardada con reserva de escritura	Escritura directa sin reserva de escritura
b.	Escritura retardada con reserva de escritura	Escritura directa con reserva de escritura

5.5.1 [5]<5.2, 5.5> Para reducir la latencia de acceso se sitúan búferes entre diferentes niveles de la jerarquía de memoria. Para la configuración de la tabla, indique los búferes necesarios entre las caches L1 y L2, y entre la cache L2 y la memoria.

5.5.2 [20]<5.2, 5.5> Describa el procedimiento para procesar un fallo de escritura en L1, considerando los componentes involucrados y la posibilidad de reemplazar un bloque no consistente.

5.5.3 [20]<5.2, 5.5> Describa el procedimiento para procesar un fallo de escritura en L1, considerando los componentes involucrados y la posibilidad de reemplazar un bloque no consistente, para una cache multinivel exclusiva (un bloque puede residir en la L1 o en el L2, pero no en ambas).

Considere las conductas de programa y cache de la tabla.

	Lectura de datos cada 1000 instrucciones	Escritura de datos cada 1000 instrucciones	Frecuencia de fallos de instrucciones	Frecuencia de fallos de datos	Tamaño de bloque (byte)
a.	200	160	0.20%	2%	8
b.	180	120	0.20%	2%	16

5.5.4 [5]<5.2, 5.5> ¿Cuáles son los anchos de banda de lectura y escritura mínimos (medido en bytes por ciclo) necesarios para obtener un CPI = 2 en una cache de escritura directa con reserva de memoria?

5.5.5 [5]<5.2, 5.5> ¿Cuáles son los anchos de banda de lectura y escritura mínimos (medido en bytes por ciclo) necesarios para obtener un CPI = 2 en una cache de escritura retardada con reserva de memoria, suponiendo que el 30% de los bloques de datos de cache reemplazados son no consistentes?

5.5.6 [5]<5.2, 5.5> ¿Cuáles son los anchos de banda de lectura y escritura mínimos necesarios para obtener un CPI = 1.5?

Ejercicio 5.6

Las aplicaciones multimedia que reproducen ficheros de audio y vídeo son parte de un tipo de carga de trabajo que se denominan cargas de trabajo de *streaming*; es decir, utilizan grandes cantidades de datos pero con poca reutilización. Considere una carga de trabajo de este tipo que accede a un conjunto de trabajo de 512 KB secuencialmente con las siguientes direcciones:

```
0, 4, 8, 12, 16, 20, 24, 28, 32, ...
```

5.6.1 [5]<5.5, 5.3> Suponga una cache de 64 KB de correspondencia directa con líneas de 32 bytes. ¿Cuál es la frecuencia de fallos con las direcciones de la tabla? Indique si esta frecuencia de fallos es muy sensible al tamaño de la cache o del conjunto de trabajo. ¿Cómo se clasificarían los fallos con esta carga de trabajo según el modelo de las tres C?

5.6.2 [5]<5.5, 5.1> Recalcule la frecuencia de fallos para líneas de 16, 64 y 128 bytes. ¿Qué tipo de localidad se está explotando?

5.6.3 [10]<5.5, 5.1> La prebúsqueda es una técnica que carga de manera especulativa, y basándose en patrones de direcciones predecibles, líneas adicionales en la cache cuando se accede a una línea de cache particular. Un ejemplo de prebúsqueda sería un búfer de *streaming*, que prebusca secuencialmente líneas de cache adyacentes y las almacena en un búfer separado cuando se trae una línea particular a la cache. Si el dato está en el búfer de prebúsqueda, se considera un acierto, se carga en la cache y se prebusca la siguiente línea. Suponga un búfer de *streaming* de dos entradas y suponga que la latencia de la cache es tal que puede cargarse una línea de cache antes de que se complete el cálculo sobre la línea de cache anterior. ¿Qué frecuencia de fallos se produce con la secuencia de la tabla?

El tamaño del bloque de cache (B) puede afectar a la frecuencia de fallos y a la latencia de los fallos. Tomando las frecuencias de fallos para varios tamaños de bloque de la tabla, y suponiendo un CPI = 1 con una media de 1.35 referencias a memoria (instrucciones y datos) por instrucción, ayude a encontrar el tamaño de bloque óptimo dadas las siguientes tasas de fallos para distintos tamaños de bloque.

	8	16	32	64	128
a.	8%	3%	1.8%	1.5%	2%
b.	4%	4%	3%	1.5%	2%

5.6.4 [10]<5.2> ¿Cuál es el tamaño de bloque óptimo para una latencia de fallos de 20 × B ciclos?

5.6.5 [10]<5.2> ¿Cuál es el tamaño de bloque óptimo para una latencia de fallos de 24 + B ciclos?

5.6.6 [10]<5.2> ¿Cuál es el tamaño de bloque óptimo para una latencia de fallos constante?

Ejercicio 5.7

En este ejercicio se explora cómo la capacidad afecta a las prestaciones globales. En general, el tiempo de acceso a la cache es proporcional a la capacidad. Suponga que el tiempo de acceso a la memoria principal es 70 ns y que el 36% de las instrucciones son accesos a memoria. La siguiente tabla muestra datos de caches L1 de dos procesadores P1 y P2.

		Tamaño de la L1	Frecuencia de fallos de la L1	Tiempo de acierto en la L1
a.	P1	1 KB	11.4%	0.62 ns
	P2	2 KB	8.0%	0.66 ns
b.	P1	8 KB	4.3%	0.96 ns
	P2	16 KB	3.4%	1.08 ns

5.7.1 [5]<5.3> Suponiendo que el tiempo de acierto de la L1 determina la duración del ciclo de P1 y P2, ¿cuáles son la frecuencias de reloj?

5.7.2 [5]<5.3> ¿Cuál es el AMAT de P1 y P2?

5.7.3 [5]<5.3> Suponiendo un CPI base igual a 1, ¿cuál es el CPI de P1 y P2? ¿Qué procesador es más rápido?

En los tres problemas siguientes se añade una cache a P1, presumiblemente para completar la capacidad limitada de la L1, con las siguientes características.

	Tamaño de la L2	Frecuencia de fallos de la L2	Tiempo de acierto de la L2
a.	512 KB	98%	3.22 ns
b.	4 MB	73%	11.48 ns

5.7.4 [10]<5.3> ¿Cuál es el AMAT de P1 con la cache L2?

5.7.5 [10]<5.3> Suponiendo un CPI base igual a 1, ¿cuál es el CPI de P1 con la cache L2?

5.7.6 [10]<5.3> ¿Qué procesador es más rápido ahora que P1 tiene una cache L2? Si P1 es más rápido, ¿cuál debería ser la frecuencia de fallos en la L1 de P2 para igualar las prestaciones de P1? Si P2 es más rápido, ¿cuál debería ser la frecuencia de fallos en la L1 de P1 para igualar las prestaciones de P2?

Ejercicio 5.8

En este ejercicio se analiza el impacto de los diferentes diseños de cache, específicamente se compara las caches asociativas con las de correspondencia directa de la sección 5.2. Para este ejercicio se utiliza la tabla con direcciones de referencias a memoria del ejercicio 5.3.

5.8.1 [10]<5.3> Utilizando las referencias del ejercicio 5.3, muestre el contenido final de una cache asociativa por conjuntos de tres vías, con bloques de dos palabras y un tamaño total de 24 palabras. Utilice la estrategia de reemplazo LRU. Para cada referencia indicar los bits del campo índice, del campo de etiqueta, el desplazamiento y si es un fallo o un acierto.

5.8.2 [10]<5.3> Utilizando las referencias del ejercicio 5.3, muestre el contenido final de una cache totalmente asociativa, con bloques de una palabra y un tamaño total de ocho palabras. Utilice la estrategia de reemplazo LRU. Para cada referencia, indique los bits del campo índice, del campo de etiqueta, el desplazamiento y si es un fallo o un acierto.

5.8.3 [15]<5.3> Utilizando las referencias del ejercicio 5.3, ¿cuál es la frecuencia de fallos en una cache totalmente asociativa, con bloques de dos palabras y un tamaño total de ocho palabras, utilizando la estrategia de reemplazo LRU? ¿Cuál es la frecuencia de fallos utilizando una estrategia de reemplazo MRU (usada más recientemente)? ¿Cuál es la mejor frecuencia de fallos posible para esta cache, con cualquier estrategia de reemplazo?

Las cache multinivel es una técnica que permite superar las limitaciones en espacio de las caches de primer nivel manteniendo su velocidad. Consideremos un procesador con los siguientes parámetros:

	CPI base, sin paradas de memoria	Velocidad del procesador	Tiempo de acceso a memoria principal	Frecuencia de fallos de la cache L1 por instrucción	Cache de segundo nivel de correspondencia directa, velocidad	Frecuencia de fallos global con una cache de segundo nivel de correspondencia directa	Cache de segundo nivel asociativa por conjuntos de ocho vías, velocidad	Frecuencia de fallos global con una cache de segundo nivel asociativa por conjuntos de ocho vías
a.	2.0	3 GHz	125 ns	5%	15 ciclos	3.0%	25 ciclos	1.8%
b.	2.0	1 GHz	100 ns	4%	10 ciclos	4.0%	20 ciclos	1.6%

5.8.4 [10]<5.3> Calcule el CPI del procesador de la tabla usando: 1) sólo una cache de primer nivel, 2) una cache de segundo nivel de correspondencia directa, 3) una cache de segundo nivel asociativa por conjuntos de ocho vías. ¿Cómo varían estos resultados si se dobla el tiempo de acceso a memoria principal? ¿Y si se reduce a la mitad?

5.8.5 [10]<5.3> Es posible tener una jerarquía de memoria con más de dos niveles de cache. Dado el procesador anterior con una cache de segundo nivel de correspondencia directa, se desea añadir un tercer nivel de cache con un tiempo de acceso de 50 ciclos y que reduce la frecuencia de fallos global al 1.3%. ¿Se mejorarían las prestaciones? En general, ¿cuáles son las ventajas y desventajas de añadir un tercer nivel de cache?

5.8.6 [20]<5.3> En procesadores antiguos, como el Pentium o el Alpha 21264, el segundo nivel cache era externo (situado en un chip diferente) al procesador principal y al primer nivel de cache. Esto permitía disponer de caches de segundo nivel de gran capacidad pero la latencia de acceso era mucho mayor y el ancho de banda típicamente era menor debido a que su frecuencia de reloj era más baja. Suponga una cache de segundo nivel de 512 KB externa y con una frecuencia de fallos global del 4%. Si cada 512 KB adicionales hiciesen disminuir la frecuencia de fallos global en un 0.7% y la cache tuviese un tiempo de acceso total de 50 ciclos, ¿qué tamaño debería tener la cache para igualar las prestaciones de la cache de segundo nivel de correspondencia directa con las características de la tabla? ¿Y de la cache asociativa con conjuntos de ocho vías?

Ejercicio 5.9

En sistemas de altas prestaciones, como por ejemplo el índice de árboles-B de una base de datos, el tamaño de página se determina principalmente en función del tamaño de los datos y las prestaciones del disco. Suponga que, en promedio, la página de índice de árboles-B está llena al 70% con entradas de tamaño fijo. La utilidad de la página es su profundidad de árbol-B, definida como el $\log_2$ (entradas). La siguiente tabla muestra, para entradas de 16 bytes y un disco de hace 10 años con una latencia de 10 ms y un ritmo de transferencia de 10 MB/s, que el tamaño de página óptimo es 16K

Tamaño de página (KB)	Utilidad de la página o profundidad del árbol-B (número de accesos a disco guardados)	Coste de acceso del índice de la página	Utilidad/coste
2	6.49 (o log2(2048/16x0.7))	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

5.9.1 [10]<5.4> ¿Cuál es el mejor tamaño de página si las entradas son de 128 bytes?

5.9.2 [10]<5.4> Basándose en el problema 5.9.1, ¿cuál es el mejor tamaño de página si las páginas están llenas al 50%?

5.9.3 [20]<5.4> Basándose en el problema 5.9.2, ¿cuál es el mejor tamaño de página si se utiliza un disco moderno con una latencia de 3 ms y un ritmo de trans-

ferencia de 100 MB/s? Explique por qué los servidores futuros probablemente tendrán páginas de mayor tamaño.

El número de accesos al disco se puede reducir si se guarda en DRAM las páginas "usadas frecuentemente" (páginas "calientes"), pero ¿cómo se determina el significado exacto de páginas "usadas frecuentemente" en un sistema? Normalmente se utiliza la relación de coste entre accesos a DRAM y disco para cuantificar el umbral de tiempo de reuso de las páginas calientes. El coste de un acceso a disco es $Disco/accesos_por_segundo mientras que el coste de mantener una página en DRAM es $DRAM_MB/tamaño_de_página. Los costes típicos de DRAM y disco y los tamaños de páginas de bases de datos típicos en varios años se muestran en la tabla:

Año	Coste de DRAM ($/MB)	Tamaño de página (KB)	Coste de disco ($/disco)	Frecuencia de acceso a disco (accesos/seg)
1987	5000	1	15 000	15
1997	15	8	2000	64
2007	0.05	64	80	83

5.9.4 [10]<5.1, 5.4> ¿Cuáles son los umbrales de tiempo de reutilización para esas tres generaciones de la tecnología?

5.9.5 [10]<5.4> ¿Cuáles son los umbrales de tiempo de reutilización si se hubiese mantenido un tamaño de página de 4K? ¿Cuál es la tendencia?

5.9.6 [20]<5.4> ¿Qué otros factores pueden cambiarse para seguir utilizando el tamaño de página de 4K (evitando así reescrituras software)? Discuta sus opciones de uso con las tendencias actuales de tecnología y costes.

Ejercicio 5.10

Como se ha descrito en la sección 5.4, la memoria virtual utiliza una tabla de páginas para rastrear la traducción de direcciones virtuales a direcciones físicas. Este ejercicio muestra cómo actualizar esta tabla a medida que se accede a la direcciones. La siguiente tabla contiene una secuencia de direcciones virtuales. Suponga páginas de 4 KB, una TLB totalmente asociativa de cuatro entradas y una estrategia de reemplazo LRU verdadero. Incremente el siguiente mayor número de página, en caso de que la página se tenga que traer desde el disco.

a.	4095, 31272, 15789, 15000, 7193, 4096, 8912
b.	9452, 30964, 19136, 46502, 38110, 16653, 48480

TLB

Válido	Etiqueta	Número de página física
1	11	12
1	7	4
1	3	6
0	4	9

Tabla de páginas

Válido	Página física o en disco
1	5
0	Disco
0	Disco
1	6
1	9
1	11
0	Disco
1	4
0	Disco
0	Disco
1	3
1	12

5.10.1 [10]<5.4> Dadas las referencia de la tabla y el estado inicial de la TLB y la tabla de páginas mostrados, determine el estado final del sistema. Indique además para cada referencia de memoria si es un acierto de TLB, un acierto en la tabla de páginas o un fallo de página.

5.10.2 [15]<5.4> Repita el problema 5.10.1 utilizando páginas de 16 KB en lugar de 4KB. ¿Cuáles son las ventajas de tener un mayor tamaño de página? ¿Cuáles son las desventajas?

5.10.3 [15]<5.3, 5.4> Muestre los contenidos finales de la TLB si fuese una TLB asociativa por conjuntos de dos vías. Muestre los contenidos también en caso de una TLB de correspondencia directa. Discuta la importancia de disponer de una TLB en aplicaciones de altas prestaciones. ¿Cómo se manejarían los accesos a memoria virtual si no hubiese TLB?

Hay varios parámetros que influyen en el tamaño de la tabla de páginas. En la tabla se muestran varios parámetros clave para la tabla de páginas:

	Tamaño de la dirección virtual	Tamaño de página	Tamaño de las entradas de la tabla de páginas
a.	32 bits	4 KB	4 bytes
b.	64 bits	16 KB	8 bytes

5.10.4 [5]<5.4> Determine el tamaño total de la tabla de páginas para un sistema que ejecuta cinco aplicaciones que utilizan la mitad de la memoria disponible.

5.10.5 [10]<5.4> Determine el tamaño total de la tabla de páginas para un sistema que ejecuta cinco aplicaciones que utilizan la mitad de la memoria disponible, suponiendo que se dispone de una tabla de páginas de dos niveles con 256 entradas. Suponga que cada entrada de la tabla de páginas principal es de 6 bytes. Calcule la cantidad de memoria mínima y máxima necesaria.

5.10.6 [10]<5.4> Un diseñador de caches quiere aumentar el tamaño de una cache de 4 KB indexada virtualmente y etiquetada físicamente. Dados los tamaños de página de la tabla, ¿es posible construir una cache de correspondencia directa de 16 KB, suponiendo dos palabras por bloque? ¿Cómo debería aumentarse el tamaño de los datos de la cache?

Ejercicio 5.11

En este ejercicio analizamos las optimizaciones espacio/tiempo de las tablas de páginas. La siguiente tabla muestra varios parámetros de un sistema de memoria virtual.

	Dirección virtual (bits)	Tamaño de la DRAM física	Tamaño de página	Tamaño PTE (byte)
a.	32	4 GB	8 KB	4
b.	64	16 GB	4 KB	8

5.11.1 [10]<5.4> Para una tabla de páginas en un único nivel, ¿cuántas entradas de la tabla de páginas (PTE) se necesitan? ¿Cuánta memoria física es necesaria para almacenar la tabla de páginas?

5.11.2 [10]<5.4> La utilización de una página de tablas en varios niveles permite mantener en memoria física sólo las PTE activas, reduciendo así las necesidades de memoria de la tabla de páginas. ¿Cuántos niveles de tablas de páginas se necesitarán en este caso? ¿Cuántos accesos a memoria serán necesarios para la traducción de la dirección si se produce un fallo de TLB?

5.11.3 [15]<5.4> Se puede utilizar una tabla invertida de páginas para optimizar el espacio y el tiempo. ¿Cuántas PTE son necesarias para almacenar la tabla de páginas? Suponiendo una implementación con funciones almohadilla, ¿cuáles son el caso común y el peor caso del número de referencias a memoria necesarios para procesar un fallo de TLB?

La siguiente tabla muestra los contenidos de una TLB de cuatro entradas.

Entrada	Válida	Dirección virtual de la página	Modificada	Protección	Dirección física de la página
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.11.4 [5]<5.4> ¿En qué casos el bit de validez de la entrada 2 será 0?

5.11.5 [5]<5.4> ¿Qué ocurre cuando una instrucción escribe en la dirección virtual 30? ¿En qué caso una TLB manejada por software es más rápida que una TLB en hardware?

5.11.6 [5]<5.4> ¿Qué ocurre cuando una instrucción escribe en la dirección virtual xxx?

Ejercicio 5.12

En este ejercicio analizaremos cómo influyen las estrategias de reemplazo en frecuencia de fallos. Suponga una cache asociativa por conjuntos de dos vías con cuatro bloques. Podría serle de utilidad hacer una tabla como la de la página 483 para resolver los problemas de este ejercicio, como se muestra a continuación para la secuencia de direcciones "0, 1, 2, 3, 4".

Dirección de memoria del bloque accedido	Fallo/ acierto	Bloque expulsado	Contenido de los bloques de cache después de la referencia			
			Conjunto 0	Conjunto 0	Conjunto 1	Conjunto 1
0	fallo		Mem[0]			
1	fallo		Mem[0]		Mem[1]	
2	fallo		Mem[0]	Mem[2]	Mem[1]	
3	fallo		Mem[0]	Mem[2]	Mem[1]	Mem[3]
4	fallo	0	Mem[4]	Mem[2]	Mem[1]	Mem[3]
...						

La siguiente tabla muestra la secuencia de direcciones:

	Secuencia de direcciones
a.	0, 2, 4, 0, 2, 4, 0, 2, 4
b.	0, 2, 4, 2, 0, 2, 4, 0, 2

5.12.1 [5]<5.3, 5.5> Suponiendo una estrategia de reemplazo LRU, ¿cuántos aciertos se producirán en esta secuencia de direcciones?

5.12.2 [5]<5.3, 5.5> Suponiendo una estrategia de reemplazo MRU (más recientemente usado), ¿cuántos aciertos se producirán en esta secuencia de direcciones?

5.12.3 [5]<5.3, 5.5> Simule una estrategia de reemplazo aleatorio echando una moneda al aire. Por ejemplo, "cara" significa expulsar el primer bloque del conjunto y "cruz" significa expulsar el segundo bloque del conjunto. ¿Cuántos aciertos se producirán en esta secuencia de direcciones?

5.12.4 [10]<5.3, 5.5> ¿Qué dirección debería ser expulsada en cada reemplazo para maximizar el número de aciertos? ¿Cuántos aciertos se producirán en esta secuencia de direcciones con esta estrategia "óptima"?

5.12.5 [10]<5.3, 5.5> Describa por qué es difícil implementar una estrategia de reemplazo que sea óptima para cualquier secuencia de direcciones.

5.12.6 [10]<5.3, 5.5> Suponga que para cada referencia de memoria se puede decidir si la dirección solicitada se lleva o no a la cache. ¿Qué impacto podría tener en la frecuencia de fallos?

Ejercicios 5.13

Para soportar varias máquinas virtuales se requieren dos niveles de virtualización. Cada máquina virtual controla la traducción de dirección virtual (VA) a dirección física (PA), y el hipervisor traduce la dirección física (PA) de cada máquina virtual a dirección de la máquina (MA). Para acelerar las traducciones se utiliza una técnica software denominada "tabla de páginas acompañante (shadow paging)" que duplica cada tabla de páginas de la máquina virtual en el hipervisor, e intercepta cambios en las traducciones VA a PA para mantener copias consistentes. Para eliminar la complejidad de las tablas de páginas acompañantes, una estrategia hardware, denominada tabla de páginas anidada (o tabla de páginas extendida), soporta de forma explícita dos tipos de tablas de páginas (VA => PA y PA => MA) y puede procesar estas tablas puramente en hardware.

Consideremos la siguiente secuencia de operaciones:

(1) crear proceso; (2) fallo de TLB; (3) fallo de página; (4) cambio de contexto

5.13.1 [10]<5.4, 5.6> Para esta secuencia de operaciones, indique qué ocurriría al utilizar la estrategia de tablas de páginas sombreadas y la estrategia de tabla de páginas anidadas.

5.13.2 [10]<5.4, 5.6> Suponiendo una tabla de páginas en cuatro niveles y basada en el x86 tanto en la tabla de páginas invitada como en la tabla de páginas anidada, ¿cuántas referencias de memoria son necesarias para procesar un fallo de TLB en la tabla de páginas nativa y en la tabla de páginas anidada?

5.13.3 [10]<5.4, 5.6> Entre frecuencia de fallos de TLB, latencia de fallo de TLB, frecuencia de fallos de páginas y latencia del manejador del fallo de página, ¿qué métrica es más importante para una tabla de páginas sombreada? ¿Cuáles son importantes para una tabla de páginas anidada?

La siguiente tabla muestra algunos parámetros de un sistema de paginado sombreado.

Fallos de TLB por cada 1000 instrucciones	Latencia de fallo de TLB de tabla de páginas anidada	Fallos de página por cada 1000 instrucciones	Sobrecoste de fallos de página sombreado
0.2	200 ciclos	0.001	30 000 ciclos

5.13.4 [10]<5.6> Para un programa de prueba con una ejecución nativa con CPI = 1, ¿cuál es el CPI con tablas de páginas sombreadas con respecto al CPI con tablas de páginas anidadas (suponiendo sólo sobrecoste de virtualización de tablas de páginas)?

5.13.5 [10]<5.6> ¿Qué técnicas pueden utilizarse para reducir el sobrecoste introducido por el sombreado de la tabla de páginas?

5.13.6 [10]<5.6> ¿Qué técnicas pueden utilizarse para reducir el sobrecoste introducido por la tabla de páginas anidada?

Ejercicio 5.14

Una de los mayores impedimentos para la difusión de la máquinas virtuales es el sobrecoste introducido por su ejecución en el sistema nativo. En la tabla se muestran varios parámetros de las prestaciones y del comportamiento.

	CPI base	Accesos privilegiados del SO por cada 1000 instrucciones	Impacto de las llamadas al SO invitado sobre las prestaciones	Impacto de las excepciones al MMV sobre las prestaciones	Accesos de E/S por cada 1000 instrucciones	Tiempo de acceso a E/S (incluye llamada al SO invitado)
a.	2	100	20 ciclos	150 ciclos	20	1000 ciclos
b.	1.5	110	25 ciclos	160 ciclos	10	1000 ciclos

5.14.1 [10]<5.6> Calcule el CPI suponiendo que no hay accesos a E/S. ¿Cuál sería el CPI si se duplica el impacto de la MMV? ¿Y si se reduce a la mitad? Si un desarrollador de software de máquina virtual quiere obtener una degradación en las prestaciones del 10%, ¿cuál sería la mayor penalización posible de las excepciones al MMV?

5.14.2 [10]<5.6> Los accesos de E/S tienen un impacto elevado sobre las prestaciones del sistema. Calcule el CPI de un máquina con las características de la tabla, suponiendo un sistema no virtualizado. Calcule el CPI otra vez suponiendo un sistema virtualizado. ¿Cómo cambiaría el CPI si el número de accesos de E/S se redujese a la mitad? Explique por qué las aplicaciones limitadas por E/S son menos afectadas por la virtualización.

5.14.3 [30]<5.6> Compare y contraste las ideas de memoria virtual y máquina virtual. ¿Cuáles son sus objetivo? ¿Cuáles son los pros y contras de cada una? Señale casos en los que es deseable tener memoria virtual y casos en los que es deseable tener una máquina virtual.

5.14.4 [20]<5.6> En la sección 5.6 se ha discutido la virtualización bajo la suposición de que el sistema virtualizado tiene la misma ISA que el hardware subyacente. Sin embargo, un posible uso de la virtualización es la emulación de una ISA no nativa. Un ejemplo es QEMU, que emula varias ISA, por ejemplo, MIPS, SPARC y PowerPC. Indique algunas de las dificultades de esta clase de virtualización. ¿Es posible que el sistema emulado sea más rápido que la ejecución en la ISA nativa?

Ejercicio 5.15

En este ejercicio se analiza la unidad de control de un controlador de cache para un procesador con un búfer de escritura. Como punto de partida para el diseño de la máquina de estados finitos, se utiliza la máquina de estados finitos de la figura 5.34. Suponga que el controlador es para una cache de correspondencia directa sencilla, como la descrita en la página 530, pero con un búfer de escritura añadido, con capacidad de un bloque.

Recuerde que el objetivo del búfer de escritura es servir como un almacenamiento temporal para que el procesador no tenga que esperar por dos accesos a memoria en el procesamiento de un fallo en un bloque inconsistente. En lugar de escribir en memoria el bloque antes de leer el nuevo bloque, se almacena el bloque inconsistente en el búfer y se comienza de forma inmediata la lectura del nuevo bloque. De este modo, el bloque inconsistente se escribe en memoria mientras el procesador está trabajando.

5.15.1 [10]<5.5, 5.7> ¿Qué ocurriría si el procesador hace una referencia que produce un acierto en la cache mientras se está escribiendo un bloque del búfer de escritura en la memoria?

5.15.2 [10]<5.5, 5.7> ¿Qué ocurriría si el procesador hace una referencia que produce un fallo en la cache mientras se está escribiendo un bloque del búfer de escritura en la memoria?

5.15.3 [10]<5.5, 5.7> Diseñe una máquina de estados finitos que permita el uso del búfer de escritura.

Ejercicio 5.16

La coherencia cache incumbe a cómo acceden varios procesadores a un bloque de cache. La siguiente tabla muestra dos procesadores y sus operaciones de lectura/escritura en palabras diferentes de un bloque de cache X (inicialmente X[0] = X[1] = 0).

	P1	P2
a.	`X[0] ++; X[1] =4;`	`X[0] = 2; X[1] ++;`
b.	`X[0] ++; X[1] += 3;`	`X[0] = 5; X[1] = 2;`

5.16.1 [15]<5.8> Indique los posibles valores del bloque X con una implementación correcta de un protocolo de coherencia de cache. Indique al menos un valor posible si el protocolo no asegura la coherencia de cache.

5.16.2 [15]<5.8> Indique la secuencia de operaciones válidas de cada procesador/cache con un protocolo de fisgoneo para terminar las operaciones de lectura/escritura especificadas en la tabla.

5.16.3 [10]<5.8> Determine el mejor caso y el peor caso en el número de fallos de cache que se producirían para terminar las operaciones de lectura/escritura especificadas en la tabla.

La consistencia de memoria se refiere a cómo se ven varios datos. La siguiente tabla muestra dos procesadores y sus operaciones de lectura/escritura en diferentes bloques de cache (A y B inicialmente son 0).

	P1	P2
a.	`A = 1; B = 2; A ++; B ++;`	`C = B; D = A;`
b.	`A = 1; B += 2; A ++; B = 4;`	`C = B; D = A;`

5.16.4 [15]<5.8> Indique los posibles valores de C y D para una implementación que asegure las propiedades de la consistencia de la página 535.

5.16.5 [15]<5.8> Indique al menos un posible par de valores para C y D si esas propiedades no se cumplen.

5.16.6 [15]<5.2, 5.8> ¿Qué combinación de las estrategias de escritura y de reserva de escritura resulta en una implementación más sencilla del protocolo?

Ejercicio 5.17

Tanto el Nehalem como el Barcelona son multiprocesadores (CMP) en un chip, que tienen varios núcleos y sus caches en un único chip. El diseño de caches L2 integradas en el mismo chip para CMP tiene varias alternativas interesantes. La siguiente tabla muestra las frecuencias de fallos y las latencias de aciertos para dos programas de prueba con cache L2 privada y con cache L2 compartida. Suponga que se produce un fallo de cache L1 cada 32 instrucciones.

	Privada	Compartida
Fallos por instrucción del programa A	0.30%	0.12%
Fallos por instrucción del programa B	0.06%	0.03%

La siguiente tabla muestra las latencias de aciertos

	Cache privada	Cache compartida	Memoria
a.	6	12	120
b.	8	20	120

5.17.1 [15]<5.10> ¿Qué diseño de cache es mejor para cada unos de los dos programas de prueba? Razone la respuesta con datos.

5.17.2 [15]<5.10> La latencia de la cache compartida aumenta con el tamaño del CMP. Determine el mejor diseño si se duplica la latencia de la cache compartida. Al aumentar el número de núcleos del CMP, el ancho de banda a la memoria fuera del chip se convierte en el cuello de botella. Determine el mejor diseño si se duplica la latencia de la memoria fuera del chip.

5.17.3 [10]<5.10> Discuta los pros y contras de sistemas con cache L2 privada y compartida para cargas de trabajo de un solo hilo, multihilo y multiprogramada. Reconsidere los pros y contras si se dispone de una cache L3 integrada en el chip.

5.17.4 [15]<5.10> Suponga que ambos programas de prueba tienen un CPI base igual a 1 (cache L2 ideal). Si una cache no bloqueante aumenta el número de fallos procesados concurrentemente de 1 a 2, ¿cuánto se mejorarían las prestaciones respecto a un sistema con cache L2 compartida? ¿Y respecto a un sistema con cache L2 privada?

5.17.5 [10]<5.10> Suponiendo que una nueva generación de procesadores duplica el número de núcleos cada 18 meses, ¿cuál debería ser el ancho de banda de la memoria fuera del chip en el año 2012 para mantener las prestaciones por cada núcleo?

5.17.6 [15]<5.10> Considerando la jerarquía de memoria completa, ¿qué optimizaciones podrían mejorar el número de fallos procesados concurrentemente?

Ejercicio 5.18

En este ejercicio mostramos la definición de un informe de un servidor web y examinamos las optimizaciones de código que permiten mejorar la velocidad de procesamiento del informe. La estructura del datos del informe es la siguiente:

```
struct entry {
    int srcIP;          // dirección IP remota
    char URL[128];   // solicita URL (ejemplo “GET index. Html”)
    long long refTime;   // tiempo de referencia
    int status;   //estado de la conexión
    char brower[64]; // nombre del buscador cliente
} log [NUM_ENTRIES];
```

Algunas de las funciones de procesamiento en un informe son:

a.	`topK_sourceIP( );`
b.	`peak_hour (int status); // horas de mayor demanda de un estado`

5.18.1 [5]<5.11> ¿A qué campos del informe se accederá con la función de procesamiento indicada? Suponiendo bloques de cache de 64 bytes y que no hay prebúsqueda, ¿cuántos de fallos por entrada se producirán de media en la función indicada?

5.18.2 [10]<5.11> ¿Cómo se podría reorganizar la estructura de datos para mejorar la utilización de la cache y la localidad de los accesos? Muestre la estructura diseñada.

5.18.3 [10]<5.11> Dé un ejemplo de otra función de procesamiento del informe que pudiese necesitar una estructura de datos diferente. Si ambas funciones son importantes, ¿cómo se podría reescribir el programa para mejorar las prestaciones? Complete la discusión con datos y fragmentos de código.

Para los problemas siguientes utilizaremos los datos de "Cache Performance for SPEC CPU2000 Benchmarks" (www.cs.wisc.edu/multifacet/misc/spec2000cachedata/) correspondientes a la pareja de programas de la tabla.

a.	apsi/facerec
b.	perlbmk/ammp

5.18.4 [10]<5.11> Para caches de datos de 64 B con varios conjuntos de asociatividad, clasifique los fallos según el modelo de las tres C y determinar la frecuencia de cada tipo de fallo para cada programa de prueba.

5.18.5 [10]<5.11> Seleccione el conjunto de asociatividad que debería utilizarse en una cache L1 de 64 KB compartida por ambos programas. Si la cache L1 fuese de correspondencia directa, seleccione el conjunto de asociatividad que debería utilizarse en una cache L2 de 1 MB.

5.18.6 [20]<5.11> Dé un ejemplo en el que un mayor conjunto de asociatividad incrementaría la frecuencia de fallos. Diseñe la configuración de la cache y una secuencia de referencias para demostrar este incremento.

Respuestas a las autoevaluaciones

\$5.1, página 457: 1 y 4. (3 es falso porque el coste de la jerarquía de memoria cambia con el computador, pero en 2008 el mayor coste habitualmente corresponde a la DRAM).
\$5.2, página 475: 1 y 4: una penalización por fallo menor puede favorecer a bloques más pequeños, ya que no hay latencias elevadas, pero anchos de banda de memoria mayores habitualmente llevan a bloques más grandes, por que la penalización por fallo es solo ligeramente superior.
\$5.3, página 491: 1.
\$5.4, página 517: 1-a, 2-c, 3-b, 4-d.
\$5.5, página 525: 2. (Tanto tamaños de bloque grandes como prebúsqueda pueden reducir los fallos obligatorios, por lo tanto 1 es falso).

6

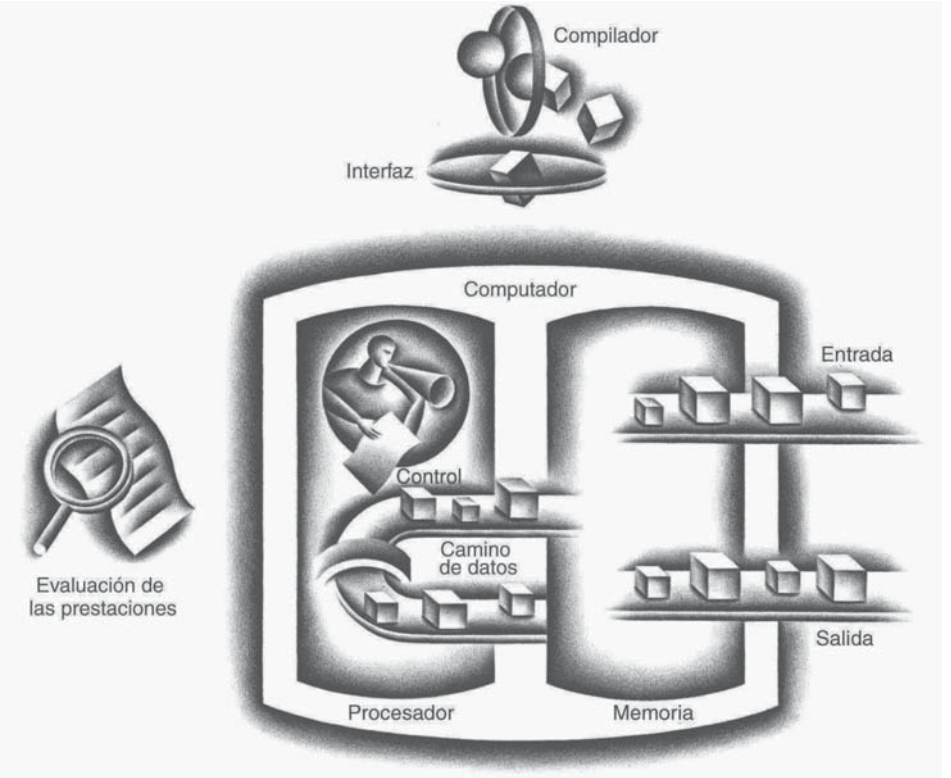
Almacenamiento y otros aspectos de la E/S

Combinando ancho de banda y almacenamiento … se permite un acceso rápido y fiable a los ricos contenidos que se expanden sobre los proliferantes discos y … depósitos de Internet.

George Gilder
The End Is Drawing Nigh, 2000

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco componentes clásicos del computador



6.1 Introducción

Aunque los usuarios pueden enfadarse cuando su computador se bloquea y deben arrancarlo de nuevo, se enfurecerían si su sistema de almacenamiento se bloqueara y perdieran toda la información. Por ello, el listón de la confiabilidad se pone mucho más alto para el almacenamiento que para la computación. Las redes también planifican los posibles fallos en la comunicación, e incluyen varios mecanismos para detectar y recuperarse de tales fallos. Por lo tanto, los sistemas E/S ponen generalmente mucho énfasis en la confiabilidad y el coste, mientras que los procesadores y la memoria se centran en las prestaciones y el coste.

Los sistemas E/S deben ser también planificados teniendo en cuenta su capacidad de ampliación y la diversidad de los dispositivos, lo cual no es un asunto importante en los procesadores. La capacidad de ampliación está relacionada con la capacidad de almacenamiento, la cual corresponde a otro parámetro de diseño en los sistemas E/S; los sistemas pueden necesitar un límite inferior de capacidad de almacenamiento para cumplir con su papel.

Aunque las prestaciones tienen una importancia menor para la E/S, es más complejo. Por ejemplo, en algunos dispositivos preocupa principalmente su latencia de acceso,

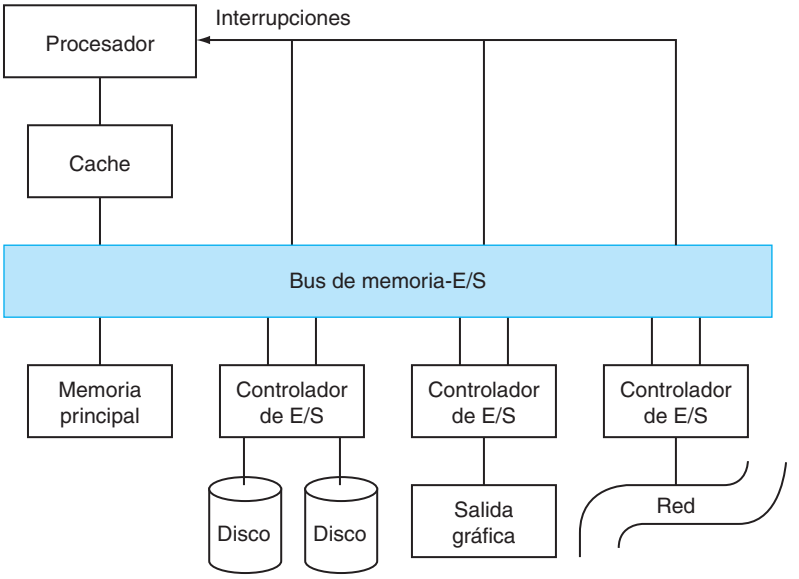


FIGURA 6.1 Un conjunto típico de dispositivos E/S. Las conexiones entre los dispositivos E/S, procesador y memoria normalmente se denominan buses. Aunque este término se asocia a conexiones paralelas compartidas y la mayoría de las conexiones E/S actuales están más cerca de ser líneas serie dedicadas. En la comunicación entre los dispositivos y el procesador se utilizan tanto interrupciones como protocolos sobre el bus, como veremos en este capítulo. La figura 6.9 muestra la organización para un PC de sobremesa.

mientras que en otros la productividad es crucial. Además, las prestaciones dependen de muchos aspectos del sistema: las características del dispositivo, la conexión entre el dispositivo y el resto del sistema, la jerarquía de memoria y el sistema operativo. Todos los componentes, desde los dispositivos E/S hasta el procesador y el software del sistema, afectarán a la confiabilidad, la capacidad de ampliación y las prestaciones de las tareas que incluyen E/S. La figura 6.1 muestra la estructura de un sistema sencillo con su E/S.

Los dispositivos E/S son increíblemente variados. Tres características son útiles para organizar esta amplia variedad:

- *Comportamiento:* entrada (se lee un sola vez), salida (solo escritura, no puede ser leída), o almacenamiento (puede ser releída y rescrita muchas veces).
- *Interlocutor:* persona o máquina que está al otro lado del dispositivo de E/S y que le envía datos a su entrada o lee datos de su salida.
- *Ritmo de transferencia de datos:* ritmo máximo al cual los datos pueden ser transferidos entre el dispositivo E/S y la memoria principal o el procesador. Es útil conocer, al diseñar un sistema de E/S, cuál es la demanda máxima que el dispositivo puede generar.

Por ejemplo, un teclado es un dispositivo de *entrada* utilizado por un *humano* con un *ritmo de transferencia de datos máximo* de alrededor de 10 bytes por segundo. La figura 6.2 muestra algunos de los dispositivos E/S conectados a los computadores.

En el capítulo 1 vimos brevemente cuatro dispositivos E/S importantes y característicos: ratón, pantallas gráficas, discos y redes. En este capítulo profundizamos en el

Dispositivo	Comportamiento	Interlocutor	Ritmo de transferencia de datos (Mbit/seg)
Teclado	entrada	humano	0.0001
Ratón	entrada	humano	0.0038
Entrada de voz	entrada	humano	0.2640
Entrada de sonido	entrada	máquina	3.0000
Escáner	entrada	humano	3.2000
Salida de voz	salida	humano	0.2640
Salida de sonido	salida	humano	8.0000
Impresora láser	salida	humano	3.2000
Pantalla gráfica	salida	humano	800.0000–8000.0000
Módem	entrada o salida	máquina	0.0160–0.0640
Red/LAN	entrada o salida	máquina	100.0000–1000.0000
Red/inalámbrica LAN	entrada o salida	máquina	11.0000–54.0000
Disco óptico	almacenamiento	máquina	80.0000
Cinta magnética	almacenamiento	máquina	32.0000
Disco magnético	almacenamiento	máquina	240.0000–2560.0000

FIGURA 6.2 Diversidad de los dispositivos de E/S. Los dispositivos de E/S pueden distinguirse porque se utilizan como dispositivo de entrada, de salida o de almacenamiento; por sus interlocutores de comunicación (persona u otro computador); y por sus anchos de banda. Los anchos de banda de datos abarcan ocho órdenes de magnitud. Observe que una red puede ser un dispositivo de entrada o de salida, pero no puede ser utilizada para almacenamiento. Los ritmos de transferencia para los dispositivos siempre se indican en base 10, así que 10 Mbit/seg = 10 000 000 bits/seg.

almacenamiento en disco y otros temas relacionados. En el CD hay una sección de temas avanzados en redes, temas que también están bien cubiertos en otros libros.

¿Cómo debemos valorar las prestaciones de la E/S cuando a menudo depende de la aplicación? En algunos entornos, podríamos preocuparnos principalmente de la productividad del sistema. En estos casos, el ancho de banda de E/S será lo más importante. Incluso el ancho de banda de E/S puede ser medido de dos formas distintas:

1. ¿Cuántos datos pueden ser transferidos a través del sistema en un cierto tiempo?
2. ¿Cuántas operaciones de E/S podemos realizar por unidad de tiempo?

Determinar cuál es la mejor medida de las prestaciones puede depender del entorno. Por ejemplo, en muchas aplicaciones multimedia, la mayoría de las peticiones de E/S se refieren a una gran cantidad de datos, y la característica más importante es el ancho de banda. En otro entorno, podríamos desear que se procesen un número grande de accesos pequeños e independientes a un dispositivo E/S. Un ejemplo de estos entornos podría ser una oficina de recaudación de impuestos. Por lo general, estas oficinas se preocupan de procesar un número grande de formularios en un tiempo dado; cada formulario de impuesto se almacena de forma independiente y suelen ser bastante pequeños. Un sistema orientado a la transferencia de grandes ficheros podría ser satisfactorio, pero un sistema de E/S que pueda soportar la transferencia simultánea de muchos ficheros pequeños puede ser más barato y rápido para procesar millones de formularios de impuestos.

Peticiones E/S: lecturas o escrituras en dispositivos E/S

En otras aplicaciones nos interesa principalmente del tiempo de respuesta, que, como se recordará, es el tiempo transcurrido hasta que se lleva a cabo una determinada tarea. Si la **petición de E/S** es extremadamente grande, el tiempo de respuesta dependerá en gran medida del ancho de banda, pero en muchos sistemas la mayoría de los accesos serán pequeños, y el sistema de E/S con la menor latencia por acceso proporcionará el mejor tiempo de respuesta. Para máquinas de un solo usuario, como los computadores de sobremesa y los portátiles, el tiempo de respuesta es la característica clave de las prestaciones.

Un gran número de aplicaciones, especialmente las del extenso mercado comercial de la informática, requieren tanto alta productividad como pequeños tiempos de respuesta. Como ejemplo podemos incluir los cajeros automáticos, los sistemas de facturación y de control de almacén, los servidores de ficheros y los servidores Web. En tales entornos, nos importa tanto el tiempo que requiere cada tarea como el número de tareas que se pueden realizar en un segundo. El número de peticiones desde distintos cajeros automáticos que se puede procesar en una hora no importa si cada una de ellas tarda 15 minutos: ¡no quedará ningún cliente que atender! De forma parecida, si cada petición se puede procesar rápidamente pero sólo se puede manejar un número pequeño de ellas a la vez, el computador central no será capaz de dar soporte a muchos cajeros automáticos, o el coste del procesamiento por cajero automático será muy elevado.

En resumen, las tres clases de computadores: de sobremesa, servidores y empotrados, son sensibles a la confiabilidad y al coste de la E/S. Los sistemas de sobremesa y empotrados están más enfocados al tiempo de respuesta y a la diversidad de los dispositivos de E/S, mientras que los servidores ponen mayor énfasis en la productividad y la capacidad de ampliación de los dispositivos de E/S.

6.2 Confiabilidad, fiabilidad y disponibilidad

Los usuarios requieren almacenamiento que sea confiable, pero ¿cómo definiría usted la confiabilidad? En la industria de la informática, esto es más complicado que la mera consulta en un diccionario. Después de un considerable debate, la definición estándar es la siguiente (Laprie 1985):

> *La confiabilidad de los sistemas informáticos se define como la calidad del servicio suministrado de forma tal que se pueda uno fiar justificadamente de ese servicio. El servicio que suministra un sistema es el comportamiento real observado tal como lo perciben otros sistemas que a su vez interaccionan con los usuarios. Cada módulo tiene también especificado un comportamiento ideal, donde una especificación del servicio es una descripción convenida del comportamiento esperado. Un fallo del sistema se produce cuando el comportamiento real se desvía del comportamiento especificado.*

Por ello, se necesita una especificación de referencia que indique el comportamiento esperado para poder determinar la confiabilidad. Los usuarios pueden entonces observar que el servicio proporcionado por un sistema puede ir alternando entre dos estados con respecto a las especificaciones del servicio:

1. *Servicio cumplimentado*, donde el servicio que se proporciona es como el especificado.
2. *Interrupción del servicio*, donde el servicio proporcionado es distinto del servicio especificado.

Las transiciones desde el estado 1 al estado 2 son causadas por fallos, y las transiciones desde el estado 2 al estado 1 se denominan *restablecimientos*. Los fallos pueden ser permanentes o intermitentes. El segundo tipo es el más difícil de diagnosticar porque el sistema está oscilando entre dos estados; los fallos permanentes son mucho más fáciles de diagnosticar. Esta definición lleva a dos términos relacionados: fiabilidad y disponibilidad.

Fiabilidad es una medida del cumplimiento continuado del servicio, equivalentemente, del tiempo hasta que se produce un fallo, desde un punto de referencia. Por ello, el tiempo medio hasta que se produce un fallo (*mean time to failure,* MTTF) de los discos que aparecen en la figura 6.5 es una medida de la fiabilidad. Un término relacionado es la frecuencia de fallos anual (*annual failure rate, AFR*), que se define, para un MTTF dado, como el porcentaje de dispositivos que se espera que fallen en un año. La interrupción del servicio se mide como el tiempo medio hasta que se realiza la reparación (*mean time to repair,* MTTR). El *tiempo medio entre fallos* (*mean time between failures,* MTBF) es simplemente la suma MTTF + MTTR. Aunque MTBF es ampliamente usado, MTTF es a menudo un término más apropiado.

Disponibilidad es una medida del cumplimiento del servicio con respecto a la alternancia entre los dos estados de cumplimiento e interrupción. La disponibilidad se cuantifica estadísticamente como

$$\text{Disponibilidad} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Observe que fiabilidad y disponibilidad son medidas cuantificables, en vez de ser meros sinónimos del concepto de confiabilidad.

¿Cuál es la causa de los fallos? La figura 6.3 resume muchos artículos que recogen datos sobre las razones por las que fallan los computadores y sistemas de telecomunicaciones. Claramente, los informáticos o personas que interactúan con las máquinas originan una buena parte de los fallos.

Informático	Software	Hardware	Sistema	Año en que se obtuvieron los datos
42%	25%	18%	Centro de cálculo (Tandem)	1985
15%	55%	14%	Centro de cálculo (Tandem)	1989
18%	44%	39%	Centro de cálculo (DEC VAX)	1985
50%	20%	30%	Centro de cálculo (DEC VAX)	1993
50%	14%	19%	Red americana de teléfonos	1996
54%	7%	30%	Red americana de teléfonos	2000
60%	25%	15%	Servicios de Internet	2002

FIGURA 6.3 Resumen de los estudios realizados sobre las razones que originan fallos. Aunque es difícil obtener datos para determinar si los informáticos se encuentran en el origen de los errores ya que estos trabajadores son a menudo quienes registran las razones de los fallos, estos estudios sí que recogieron estos datos. A menudo se establecieron otras categorías, como las condiciones del entorno que producen corte de suministro eléctrico, pero en líneas generales fueron de poca importancia. Las dos filas superiores se obtuvieron de un trabajo clásico de Jim Gray [1990], que aún es ampliamente referenciado aunque han pasado casi 20 años desde que se recogieron los datos. Las siguientes dos filas se obtuvieron del trabajo de Murphy y Gent, quienes estudiaron a lo largo del tiempo las causas de los cortes de suministros eléctricos en los sistemas VAX ("Measuring system and software reliability using an automated data collection process," *Quality and Reliability Engineering International* 11:5, September–October 1995, 341–53). Las filas quinta y sexta corresponden a estudios de datos de fallos de FCC sobre la red pública conmutada de teléfonos de Estados Unidos realizado por Kuhn ("Sources of failure in the public switched telephone network," *IEEE Computer* 30:4, April 1997, 31–36) y por Patty Enriquez. El estudio más reciente de tres servicios de Internet es el realizado por Oppenheimer, Ganapath y Patterson [2003].

Para aumentar el MTTF, podemos mejorar la calidad de los componentes o diseñar sistemas que continúen funcionando cuando existen componentes que han fallado. Por ello, los fallos necesitan ser definidos con respecto a un contexto. Un fallo en un componente puede que no origine un fallo del sistema. Para resaltar esta distinción, el término *falta* se utiliza para representar fallos de un componente. Ahora describimos tres formas de mejorar el MTTF:

1. *Evitar fallos:* prevenir la existencia de faltas por defectos de construcción.
2. *Tolerancia a fallos:* utilizar redundancia para permitir que el servicio no se interrumpa con las especificaciones estipuladas a pesar de que ocurran faltas, lo cual se debe principalmente a defectos del hardware. La sección 6.9 describe las estrategias de RAID para hacer que el almacenamiento sea confiable a través de la tolerancia a fallos.
3. *Previsión de fallos:* predecir la presencia y creación de faltas; predicciones que se aplican a faltas del hardware y del software, permitiendo reemplazar el componente antes de que falle.

La disminución del MTTR puede mejorar la disponibilidad tanto como cuando se incrementa el MTTF. Por ejemplo, las herramientas para la detección, diagnóstico y reparación de faltas puede ayudar a reducir el tiempo necesario para reparar las faltas originadas por las personas, software y hardware.

Autoevaluación

¿Cuáles de la siguientes afirmaciones sobre la confiabilidad son ciertas?

1. Si el sistema está funcionando, todos sus componentes están llevando a cabo el servicio que de ellos se espera.
2. La disponibilidad es una medida cuantitativa sobre el porcentaje de tiempo que un sistema está llevando a cabo el servicio esperado.
3. La fiabilidad es una medida cuantitativa del servicio continuado llevado a cabo por un sistema.
4. Actualmente la principal fuente de interrupciones de servicio es el software.

6.3 Almacenamiento en disco

Como se mencionó en el capítulo 1, los discos magnéticos se basan en un plato giratorio recubierto de una superficie magnética y utilizan un cabezal móvil de lectura/escritura para acceder al disco. El almacenamiento en disco es **no volátil**: los datos permanecen incluso cuando está desenchufado. Un disco magnético consiste en una colección de platos (1-4), cada uno de los cuales tiene dos superficies grabables. El conjunto de platos gira de 5400 a 15 000 rpm y tiene un diámetro que va desde una pulgada a alrededor de 3.5 pulgadas. Cada superficie del disco se divide en círculos concéntricos, denominados **pistas**. Normalmente existen de 10 000 a 50 000 pistas por superficie. A su vez, cada pista se divide en **sectores** que es donde se almacena la información; cada pista puede tener de 100 a 500 sectores. Los sectores tienen normalmente una capacidad de 512 bytes, aunque existe una iniciativa para incrementar el tamaño del sector a 4096 bytes. La secuencia almacenada en el medio magnético está constituida por un número de sector, un hueco, la información asignada a ese sector que incluye un código de corrección de errores (véase el **apéndice C**, página C66), un hueco, el número del sector siguiente, y así sucesivamente.

No volátil: dispositivo de almacenamiento donde los datos mantienen su valor incluso cuando está desenchufado de la corriente eléc-

Pista: cada uno de los miles de círculos concéntricos que constituyen la superficie de un disco magnético.

Sector: cada uno de los segmentos que constituyen una pista de un disco magnético; un sector es la cantidad más pequeña de información que es leída o escrita en un disco.

Originalmente, todas las pistas tenían el mismo número de sectores y por ello el mismo número de bits, pero con la introducción de la grabación de bits por zonas (ZBR) a principios de los 90, los discos se modificaron para disponer de un número variable de sectores (y por lo tanto de bits) por pista, manteniendo constante el espaciado entre bits. ZBR incrementa el número de bits en las pistas exteriores y por ello incrementa la capacidad de almacenamiento del periférico.

Como vimos en el capítulo 1, para leer y escribir información, los cabezales de lectura/escritura deben moverse hasta situarse sobre la posición correcta. Los cabezales de disco para cada superficie están unidos y se mueven al unísono, de forma tal que cada cabezal se posiciona sobre la misma pista en cada superficie. El termino *cilindro* se utiliza para referirse a todas las pistas que se encuentran debajo de los cabezales en un determinado punto sobre todas las superficies.

Para acceder a los datos, el sistema operativo debe dirigir el disco mediante un proceso de tres etapas. La primera consiste en posicionar el cabezal sobre la pista correcta. Esta operación se denomina **búsqueda**, y el tiempo necesario para mover el cabezal a la pista deseada se denomina *tiempo de búsqueda*.

Búsqueda: proceso de posicionamiento de un cabezal de lectura/escritura sobre la pista correcta de un disco.

Los fabricantes de discos indican en los manuales correspondientes cuáles son los tiempos mínimos de búsqueda, los tiempos máximos de búsqueda y los tiempos promedio de búsqueda. Los dos primeros son fáciles de medir, pero el promedio está abierto a distintas interpretaciones debido a que depende de la distancia de búsqueda. La industria ha decidido calcular el tiempo promedio de búsqueda como la suma de los tiempos de todas las posibles búsquedas divididas por el número de posibles búsquedas. Los tiempos promedio de búsqueda que normalmente se publican varían desde 3 ms hasta 13 ms, pero, dependiendo de la aplicación y de la planificación de las peticiones de disco, el tiempo promedio de búsqueda real puede ser sólo del 25% al 33% del valor publicado debido a la localidad de las referencias a disco. Esta localidad se debe tanto a los accesos sucesivos a un mismo fichero como al sistema operativo que trata de planificar tales accesos de forma conjunta.

Una vez que el cabezal ha alcanzado la pista correcta, debemos esperar a que el sector deseado pase debajo del cabezal de lectura/escritura. Este tiempo se denomina **latencia rotacional** o **retardo rotacional**. La latencia promedio a la información deseada se encuentra a mitad de camino alrededor del disco. Como los discos rotan entre 5400 rpm y 15 000 rpm, la latencia rotacional promedio se encuentra entre

Latencia rotacional (retardo rotacional): tiempo requerido para que el sector deseado de un disco pase debajo de la cabeza de lectura/escritura; normalmente se supone que es la mitad del tiempo de rotación.

$$\text{Latencia rotacional promedio} = \frac{0.5 \text{ rotación}}{5400 \text{ rpm}} = \frac{0.5 \text{ rotación}}{5\,400 \text{ rpm} / \left(60 \frac{\text{segundos}}{\text{minuto}}\right)}$$

$$= 0.0056 \text{ segundos} = 5.6 \text{ ms}$$

y

$$\text{Latencia rotacional promedio} = \frac{0.5 \text{ rotación}}{15\,000 \text{ rpm}} = \frac{0.5 \text{ rotación}}{15\,000 \text{ rpm} / \left(60 \frac{\text{segundos}}{\text{minuto}}\right)}$$

$$= 0.00200 \text{ segundos} = 2.0 \text{ ms}$$

La última componente del acceso a un disco, el *tiempo de transferencia*, es el tiempo necesario para transferir un bloque de bits. El tiempo de transferencia es una función del tamaño del sector, la velocidad de rotación y la densidad de grabación de una pista. Los ritmos de transferencia en el año 2008 estaban entre 70 y 125 MB/seg. La única complicación se encuentra en que la mayoría de los controladores de disco tienen una memoria cache integrada que almacena sectores a medida que son transferidos; los ritmos de transferencia desde la memoria cache son normalmente mayores y pueden llegar a 375 MB/seg (3 Gbit) en 2008. Hoy en día, muchas de las transferencias de disco tienen un tamaño que incluye múltiples sectores.

Un *controlador de disco* normalmente gestiona todo el control del disco, así como la transferencia entre el disco y la memoria. El controlador añade una componente final al tiempo de acceso al disco, el *tiempo del controlador*, que es el tiempo adicional que el controlador necesita para realizar un acceso de E/S. El tiempo promedio para realizar una operación de E/S estará formado por estos cuatro tiempos más los tiempos de espera debido a que otros procesos están utilizando el disco.

Tiempo de lectura del disco

EJEMPLO

¿Cuál es el tiempo promedio para leer o escribir un sector de 512 bytes en un disco típico que gira a 15 000 rpm? El tiempo promedio de búsqueda que indica el fabricante es de 4 ms, el ritmo de transferencia es de 100 MB/seg, y el tiempo adicional del controlador es de 0.2 ms. Suponga que el disco está inactivo, así que no existe tiempo de espera.

RESPUESTA

El tiempo de acceso al disco es igual al Tiempo promedio de búsqueda + Retardo promedio rotacional + Tiempo de transferencia + Tiempo adicional del controlador. Utilizando el tiempo promedio de búsqueda publicado por el fabricante, la respuesta es

$$4.0 \text{ ms} + \frac{0.5 \text{ rotación}}{15\,000 \text{ rpm}} + \frac{0.5\text{KB}}{100 \text{ MB/seg}} + 0.2 \text{ ms} = 4.0 + 2.0 + 0.005 + 0.2 = 6.2 \text{ ms}$$

Si el tiempo promedio de búsqueda que se ha medido es un 25% del valor promedio publicado por el fabricante, la respuesta es

$$1.0 \text{ ms} + 2.0 \text{ ms} + 0.005 \text{ ms} + 0.2 \text{ ms} = 3.2 \text{ ms}$$

Observe que cuando consideramos el tiempo promedio de búsqueda medido, a diferencia de tiempo promedio de búsqueda publicado por el fabricante, la latencia rotacional puede ser la mayor componente del tiempo de acceso.

Las densidades de los discos han aumentado continuamente durante más de 50 años. El impacto de la combinación de la mejora en densidad y la reducción del tamaño físico de un periférico basado en discos magnéticos ha sido asombroso, como se muestra en la figura 6.4. El esfuerzo de distintos diseñadores de discos ha llevado a la amplia variedad de periféricos que han estado disponibles en cada momento. La figura 6.5 muestra las características de cuatro discos magnéticos. En el año 2008, estos discos de un mismo fabricante costaban entre 0.2 dólares y 2 dólares por gigabyte, dependiendo de la capacidad, la interfaz y las prestaciones.

Así como los discos seguirán siendo viables en el futuro inmediato, la disposición convencional de los bloques, no. Las suposiciones del modelo sector-pista-cilindro son que los bloques cercanos están en la misma pista, los bloques en el mismo cilindro tienen un tiempo de acceso menor porque no hay tiempo de posicionamiento, y algunas pistas están más próximas que otras. La razón de esta ruptura ha sido el aumento del nivel de las interfaces. Las interfaces inteligentes de los niveles más altos como **ATA** y **SCSI** necesitan un microprocesador dentro del disco, lo que conduce a optimizaciones de las prestaciones.

Para acelerar la transferencia secuencial, estas interfaces de los niveles más altos organizan los discos de una forma más parecida a las cintas magnéticas que a los dispositivos de acceso aleatorio. Los bloques lógicos se ordenan como una serpentina sobre una única superficie, intentando incluir a todos los sectores que se graban con la misma densidad de bits. De esta forma, bloques consecutivos pueden estar en

Advanced Technology Attachment (ATA): conjunto de órdenes utilizado como un estándar para dispositivos de E/S que es habitual en el PC.

Small Computer System Interface (SCSI): conjunto de órdenes utilizado como un estándar para dispositivos de E/S.

diferentes pistas. Más delante, en la figura 6.19, veremos un ejemplo de un error habitual cometido al asumir el modelo sector-pista-cilindro convencional.

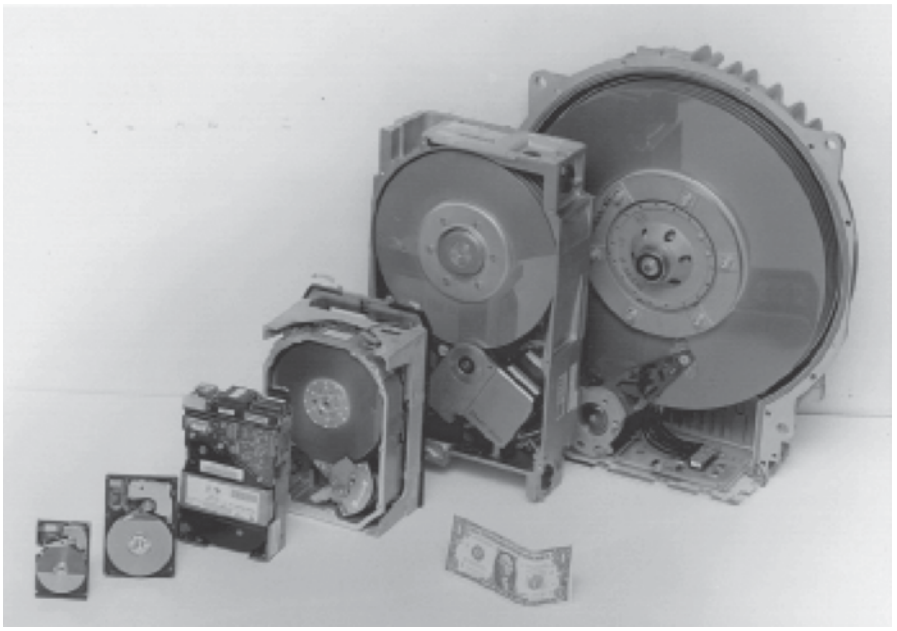


FIGURA 6.4 Seis discos magnéticos, cuyos diámetros varían desde 14 pulgadas a 1.8 pulgadas. Los discos mostrados fueron introducidos hace más de 15 años y por lo tanto no pretenden ser representativos de la mejor capacidad de almacenamiento de los discos modernos con estos diámetros. Sin embargo, esta fotografía representa los tamaños físicos relativos. El disco más grande es el DEC R81, que contiene cuatro platos de 14 pulgadas de diámetro y almacenan 456 MB. Fue fabricado en 1985. El disco de 8 pulgadas de diámetro viene de Fujitsu, fue fabricado en 1984 y almacena 130 MB en seis platos. El RD53 de Micropolis tiene cinco platos de 5.25 pulgadas y almacena 85 MB. El 0361 de IBM tiene cinco platos, pero de 3.5 pulgadas de diámetro cada uno. Este disco fabricado en 1988 almacena 320 MB. En 2008, el disco más denso de 3.5 pulgadas tiene 2 platos y almacena 1 TB en el mismo espacio, ¡proporcionando un incremento de densidad en un factor alrededor de 3000! El CP 2045 de Conner tiene dos platos de 2.5 pulgadas, contiene 40 MB, y fue fabricado en 1990. El disco más pequeño de esta fotografía es el 1820 de Integral. Este disco de plato único de 1.8 pulgadas contiene 20 MB y fue construido en 1992.

Extensión: La mayoría de los controladores de discos incluyen memorias cache. Tales caches permiten accesos rápidos a los datos que fueron leídos recientemente por parte de la CPU. Utilizan escritura directa y no se actualizan en un fallo de escritura. A menudo también utilizan algoritmos de prebúsqueda para intentar anticiparse a las peticiones. Los controladores utilizan también una cola de órdenes que permiten que el disco decida el orden en que a ejecutar estas órdenes para maximizar las prestaciones manteniendo el comportamiento correcto de la secuencia de órdenes. Obviamente, estas funciones complican la medida de las prestaciones de los discos e incrementan la importancia de la elección de la carga de trabajo.

Características	Seagate ST3300655SS	Seagate ST31000340NS	Seagate ST973451SS	Seagate ST9160821AS
Diámetro del disco (pulgadas)	3.50	3.50	2.50	2.50
Capacidad formateada de datos (GB)	147	1000	73	160
Número de superficies de disco (cabezas)	2	4	2	2
Velocidad de rotación (RPM)	15 000	7200	15 000	5400
Capacidad de la cache interna del disco (MB)	16	32	16	8
Interfaz externa, ancho de banda (MB/seg)	SAS, 375	SATA, 375	SAS, 375	SATA, 150
Ancho de banda sostenido (MB/seg)	73–125	105	79–112	44
Tiempo mínimo de búsqueda (lectura/escritura) (ms)	0.2/0.4	0.8/1.0	0.2/0.4	1.5/2.0
Tiempo medio de búsqueda (lectura/escritura) (ms)	3.5/4.0	8.5/9.5	2.9/3.3	12.5/13.0
Tiempo medio hasta que se produce un fallo (MTTF) (horas)	1 400 000 @ 25˚C	1 200 000 @ 25˚C	1 600 000 @ 25˚C	—
Frecuencia de fallos anual (AFR)	0.62%	0.73%	0.55%	—
Ciclos start/stop con contacto	—	50 000	—	>600 000
Garantía (años)	5	5	5	5
Errores de lectura no recuperables por bits leídos	<1 sector por 10^{16}	<1 sector por 10^{15}	<1 sector por 10^{16}	<1 sector por 10^{14}
Temperatura, límite de vibración (en funcionamiento)	5˚–55˚C, 60 G	5˚–55˚C, 63 G	5˚–55˚C, 60 G	0˚–60˚C, 350G
Tamaño: dimensiones (pulgadas), peso (libras)	1.0" × 4.0" × 5.8", 1.5 lbs	1.0" × 4.0" × 5.8", 1.4 lbs	0.6" × 2.8" × 3.9", 0.5 lbs	0.4" × 2.8" × 3.9", 0.2 lbs
Potencia: en funcionamiento/inactivo/en espera (vatios)	15/11/—	11/8/1	8/5.8/—	1.9/0.6/0.2
GB/pulgada cúbica, GB/vatio	6 GB/pulg^3, 10 GB/W	43 GB/pulg^3, 91 GB/W	11 GB/pulg^3, 9 GB/W	37 GB/pulg^3, 84 GB/W
Precio en 2008, $/GB	~$250, ~$1.7/GB	~$275, ~$0.3/GB	~$350, ~$5.0/GB	~$100, ~$0.6/GB

FIGURA 6.5 Características de cuatro discos magnéticos de un mismo fabricante en 2008. Los discos de las tres columnas más a la izquierda se utilizan en servidores y computadores de sobremesa, mientras que el disco de la columna de la derecha se utiliza en portátiles. Observe que el tercer disco tiene un diámetro de 2.5 pulgadas, pero es un disco de altas prestaciones con la mayor fiabilidad y el menor tiempo de búsqueda. Los discos mostrados aquí son versiones serie de la interfaz SCSI (SAS), un bus estándar de E/S para muchos sistemas, o versiones serie de ATA (SATA), un bus estándar de E/S para PC. La frecuencia de transferencia desde la cache es 3-5 veces mayor que la frecuencia de transferencia desde la superficie del disco. El coste por gigabyte tan reducido del SATA de 3.5 pulgas es debido principalmente al hiper-competitivo mercado de los PC, aunque hay diferencias en las prestaciones en E/S por segundo debido a que el SAS tiene velocidades de rotación y búsqueda más elevadas. La vida útil de estos discos es de cinco años. Observe que el MTTF especificado supone potencia y temperatura nominales. La vida útil pued ser mucho más corta si la temperatura y las vibraciones no están controladas. Para más información sobre estos dispositivos, véase el enlace *www.seagate.com*.

Autoevaluación

¿Cuáles de la siguientes afirmaciones sobre discos son ciertas?

1. Los discos de 3.5 pulgadas hacen más operaciones de E/S por segundo que los discos de 2.5 pulgadas.
2. Los discos de 3.5 pulgadas proporcionan el mayor número de gigabytes por vatio.
3. Se necesitan horas para leer secuencialmente los contenidos de un disco de alta capacidad.
4. Se necesitan meses para leer los contenidos de un disco de alta capacidad utilizando sectores aleatorios de 512 bytes.

6.4 Almacenamiento Flash

Se ha intentado muchas veces inventar una tecnología para reemplazar los discos magnéticos, y la mayoría han fallado: memorias CCD, memorias de burbujas magnéticas y memorias holográficas no dieron la talla. En el momento en que una tecnología estaba disponible, se producían avances en la tecnología de los discos como se ha dicho anteriormente, el coste disminuía y el producto competidor perdía su atractivo en el mercado.

El primer desafío creíble es la memoria flash. Esta memoria semiconductora es no volátil como los discos, pero su latencia es 100-1000 veces más rápida que la de los discos, es más pequeña, más eficiente energéticamente y más resistente a las descargas eléctricas. Igualmente importante, el hecho de que sea popular en teléfonos portátiles, cámaras digitales y reproductores de MP3 hace que haya un amplio mercado para las memorias flash dispuesto a invertir para mejorar su tecnología. Recientemente, el coste por gigabyte de la memoria flash ha estado cayendo un 50% cada año y en 2008 el precio era de $4 a $10 por gigabyte; es decir, entre 2 y 4 veces mayor que en los discos y entre 5 y 10 veces menor que en las DRAM. La figura 6.6 compara tres productos basados en memorias flash.

Características	Kingston Secure Digital (SD) SD4/8 GB	Transend Type I CompactFlash TS16GCF133	RiDATA Solid state disk 2.5 inch SATA
Capacidad de datos formateado (GB)	8	16	32
Bytes por sector	512	512	512
Ritmo de transferencia de datos (lectura/escritura en MB/seg)	4	20/18	68/50
Potencia en funcionamiento/parado (W)	0.66/0.15	0.66/0.15	2.1/—
Tamaño: alto × ancho × profundidad	0.94 ×1.26 ×0.08	1.43 ×1.68 ×0.13	0.35 ×2.75 ×4.00
Peso en gramos	2.5	11.4	52
Tiempo medio entre fallos (horas)	>1 000 000	>1 000 000	>4 000 000
GB/pulgadas^3, GB/vatio	84 GB/pul^3, 12 GB/W	51 GB/pul^3, 24 GB/W	8 GB/pul^3, 16 GB/W
Mejor precio (2008)	~$30	~$70	~$300

FIGURA 6.6 Características de tres productos de almacenamiento flash. El encapsulado estándar CompactFlash fue propuesto en 1994 por Sandisk Corporation para las tarjetas PCMCIA-ATA de los PC portátiles. Como usa la interfaz ATA, simula una interfaz de disco incluyendo órdenes de búsqueda, pistas lógicas, etc. El producto RiDATA imita una interfaz SATA de disco de 2.5 pulgadas.

Aunque el coste por gigabyte es mayor que en los discos magnéticos, la memoria flash se utiliza habitualmente en dispositivos móviles, en parte porque se presenta con capacidades menores. Como resultado, los discos de diámetro 1 pulgada están desapareciendo del mercado de los dispositivos empotrados. Por ejemplo, en 2008 el reproductor MP3 Apple iPod Shuffle se vendía por $50 y tenía 1 GB de capacidad, mientras que el disco más pequeño tenía una capacidad de 4 GB y su precio era mayor que el reproductor MP3 completo.

La memoria flash es un tipo de memoria de sólo lectura programable que se puede borrar eléctricamente (*EEPROM*). La primera memoria flash, que se llamó *flash NOR* por la similitud de la celda de almacenamiento con la puerta NOR estándar, fue un competidor directo de otras EEPROM y era direccionable de forma aleatoria como cualquier otra memoria. Unos pocos años más tarde, la memoria *flash NAND* proporcionaba mayor densidad de almacenamiento, pero como se eliminó la circuitería para el acceso aleatorio, sólo se podía leer y escribir en bloques. La NAND flash es mucho más barata por gigabyte y mucho más popular que la NOR flash; todos los productos de la figura 6.6 utilizan flash NAND. La figura 6.7 compara las características clave de las memorias flash NOR y flash NAND.

A diferencia de los discos y la memoria DRAM, pero de manera similar a otras tecnologías de EEPROM, en las memorias flash se produce desgaste de los bits (véase figura 6.7). Para sobrellevar estas limitaciones, la mayoría de los productos flash NAND incluyen un controlador para distribuir las escrituras, recolocando los bloques que han sido modificados muchas veces en bloques menos utilizados. Esta técnica se llama *nivelación del desgaste (wear leveling)*. Con la nivelación de desgaste, es muy improbable que se exceda el límite de escrituras en productos de consumo tales como teléfonos móviles, cámaras digitales, reproductores de MP3 y llaves de memoria. Estos controladores reducen las prestaciones potenciales de las memorias flash, pero son necesarios a no ser que se disponga de monitores software de alto nivel que limiten el desgaste. Sin embargo, los controladores también pueden mejorar el rendimiento *(yield)* descartando las celdas que se fabricaron de modo incorrecto.

La limitación en las escrituras es una de las razones por las que las memorias flash no son populares en computadores de sobremesa y servidores. Sin embargo, en 2008 se han vendido los primeros computadores portátiles con memoria flash en lugar de discos duros con un sobreprecio importante para ofrecer un arranque más rápido, menor tamaño y una vida útil de las baterías más larga. Hay también disponibles memorias flash con tamaños similares a la de los discos estándar, como se muestra en la figura 6.6. Combinando estas ideas, los *discos duros híbridos* incluyen, por ejemplo, una memoria flash de un gigabyte para que los portátiles puedan arrancar más rápidamente y ahorra energía permitiendo que los discos estén inactivos más frecuentemente.

Parece que en los próximos años las memorias flash podrán competir con éxito con los discos duros en el segmento de dispositivos con baterías. Como la capacidad aumenta y el coste por gigabyte sigue disminuyendo, será interesante comprobar si las

Características	Memoria flash NOR	Memoria flash NAND
Uso habitual	Memoria BIOS	Llave USB
Tamaño mínimo de acceso (bytes)	512 bytes	2048 bytes
Tiempo de lectura (microseg)	0.008	25
Tiempo de escritura (microseg)	10.00	1500 para borrar + 250
Ancho de banda de lectura (Mbytes/seg)	10	40
Ancha de banda de escritura (Mbytes/seg)	0.4	8
Desgaste (escrituras por celda)	100 000	10 000 a 100 000
Mejor precio por GB (2008)	$65	$4

FIGURA 6.7 Características de las memorias flash NOR frente a las flash NAND en 2008. Estos dispositivos pueden leer bytes y palabras de 16 bits, a pesar de elevados tamaños de acceso.

mayores prestaciones y la mejor eficiencia energética de las memorias flash le da oportunidades también en el segmento de los computadores de sobremesa y los servidores.

Autoevaluación

¿Cuáles de las siguientes afirmaciones sobre memorias flash son ciertas?

1. Como las DRAM, la memoria flash es una memoria semiconductora.
2. Como los discos, en la memoria flash no se pierde la información si se desconecta la alimentación.
3. El tiempo de acceso en lectura de la flash NOR es similar a las DRAM.
4. El ancho de banda de lectura de la flash NAND es similar a los discos.

6.5 Conexión entre procesadores, memoria y dispositivos de E/S

Entre los distintos subsistemas de un computador deben existir interfaces. Por ejemplo, la memoria y el procesador necesitan comunicarse, como lo hacen el procesador y los dispositivos de E/S. Durante muchos años, esto ha sido implementado con un *bus*. Un bus es una conexión compartida de comunicación que utiliza un conjunto de conductores para conectar muchos subsistemas. Las dos ventajas más importantes de la organización bus son la versatilidad y el bajo coste. Definiendo un único esquema de conexión, los nuevos dispositivos pueden añadirse fácilmente, y los periféricos pueden incluso ser instalados en distintos computadores que utilizan el mismo tipo de bus. Además, los buses son eficientes en coste debido a que un único conjunto de conductores se comparte de varias formas.

La principal desventaja de un bus consiste en que genera un cuello de botella en las comunicaciones, posiblemente limitando el número máximo de operaciones de E/S. Cuando la E/S debe atravesar un único bus, su ancho de banda limita el número máximo de operaciones de E/S. El diseño de un sistema bus capaz de satisfacer los requerimientos del procesador, así como de conectar un número elevado de dispositivos de E/S a la máquina, presenta un desafío muy importante.

Bus procesador-memoria: bus de pequeña longitud que conecta el procesador y la memoria; generalmente es de alta frecuencia y se diseña con el objetivo de maximizar el ancho de banda memoria-procesador.

Bus de placa base: bus diseñado para permitir que los procesadores, la memoria y los dispositivos de E/S coexistan en un único bus.

Los buses se clasifican tradicionalmente como **buses procesador-memoria** o *buses de E/S*. Los buses procesador-memoria son de pequeña longitud, generalmente de alta frecuencia y diseñados para maximizar el ancho de banda memoria-procesador. Por el contrario, los buses de E/S pueden tener una longitud grande, pueden disponer de muchos tipos de dispositivos conectados a ellos, y a menudo permiten conectar dispositivos con un amplio rango de ancho de banda de datos. Los buses de E/S normalmente no se conectan directamente a la memoria, sino que utilizan un bus procesador-memoria o un **bus de placa base** para conectarse a memoria. Otros buses con características diferentes han aparecido para realizar funciones especiales, como los buses gráficos.

Una razón por la que el diseño de un bus es tan difícil es que su velocidad máxima está en gran medida limitada por factores físicos: la longitud del bus y el número de dispositivos. Estos límites físicos evitan que la frecuencia del bus sea arbitrariamente elevada. Además, la necesidad de soportar una gama de dispositivos con latencias y ritmos de transferencia de datos que varían ampliamente también hace que el diseño de un bus sea un reto.

Como es muy difícil proporcionar una alta frecuencia utilizando muchos conductores en paralelo debido al desplazamiento y reflexión de la señal de reloj (véase

apéndice C), la industria está sustituyendo los buses paralelos compartidos por las interconexiones serie punto a punto de alta velocidad utilizando conmutadores. Por ello, estas redes están reemplazando gradualmente a los buses en nuestros sistemas.

Como resultado de esta transición, esta sección ha sido revisada en esta edición para enfatizar el problema que genera la conexión de los dispositivos E/S, procesadores y memoria, en vez de concentrarse exclusivamente en los buses.

Fundamentos de las conexiones

Consideremos una típica **transacción de bus**. Ésta incluye dos partes: el envío de la dirección y la recepción o envío de los datos. Las transacciones de bus normalmente se definen por la operación que realizan en memoria. Una transacción de *lectura* transfiere datos *desde* la memoria (bien al procesador o a un dispositivo E/S), y una transacción de escritura *escribe* datos *en* memoria. Obviamente, esta terminología es confusa. Para evitarlo, intentaremos usar los términos *entrada* y *salida*, que siempre se definen desde la perspectiva del procesador: una operación de entrada está recibiendo datos desde el dispositivo a memoria, donde el procesador puede leerlos, y una operación de salida está proporcionando datos a un dispositivo desde memoria donde el procesador los escribió.

Transacción de bus: secuencia de operaciones de bus que incluyen una petición y pueden incluir una respuesta, donde cualquiera de ellas puede transportar datos. Una transacción es iniciada por una única solicitud y puede tener asociada muchas operaciones individuales de bus.

El bus de E/S sirve como medio para ampliar la máquina y conectarle nuevos periféricos. Para facilitar esta tarea, la industria de los computadores ha desarrollado varios estándares que sirven como especificación técnica para el fabricante de computadores y para los fabricantes de periféricos. Un estándar asegura al diseñador del computador que los periféricos estarán disponibles para una nueva máquina, y asegura al fabricante de periféricos que los usuarios serán capaces de integrarlos en sus nuevos equipos. La figura 6.8 resume las características clave de los cinco estándares dominantes de buses de E/S: Firewire, USB, PCI Express (PCIe), Serial ATA (SATA) y Serial Attached SCSI (SAS). Estos buses conectan una variedad de dispositivos al computador de sobremesa, desde teclados a cámaras y discos.

Los buses tradicionales son **síncronos**. Si un bus es síncrono, incluye un reloj en las líneas de control y un protocolo de comunicaciones que se referencia a la señal de reloj. Por ejemplo, para un bus procesador-memoria que realiza una lectura desde memoria, podríamos tener un protocolo que transmite la dirección y el comando de lectura en el primer ciclo de reloj, utilizando las líneas de control para indicar el tipo de petición. Entonces, la memoria podría tener que responder con la palabra de datos en el quinto ciclo de reloj. Este tipo de protocolo puede ser implementado fácilmente con una pequeña máquina de estados finitos. Como el protocolo está predeterminado e implica poca lógica, el bus puede funcionar a alta frecuencia y la lógica de la interfaz será pequeña. Sin embargo, los buses síncronos tienen dos grandes desventajas. En primer lugar, cada dispositivo conectado al bus debe funcionar a la misma frecuencia de reloj. En segundo lugar, debido a problemas de desalineamiento del reloj, los buses síncronos no pueden ser largos si se quiere que funcionen a alta frecuencia (véase el apéndice C). Los buses procesador-memoria a menudo son síncronos debido a que los dispositivos que se intercomunican están cerca, son pocos y están preparados para funcionar a altas frecuencias de reloj.

Bus síncrono: bus que incluye un reloj en las líneas de control y un protocolo de comunicaciones establecido que se referencia a la señal de reloj.

Estos problemas llevan a conexiones **asíncronas**. Una conexión asíncrona no tiene reloj, por lo que puede adaptarse a una amplia variedad de dispositivos, y el bus puede

Interconexión asíncrona: utiliza para coordinar su uso un protocolo con acuse de recibo en vez de utilizar un reloj; se pueden adaptar una amplia variedad de dispositivos de distintas frecuencias.

Características	Firewire (1394)	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Tipo de bus	External	External	Internal	Internal	External
Dispositivos por canal	63	127	1	1	4
Ancho básico del bus de datos (señales)	4	2	2 por línea	4	4
Ancho de banda pico teórico	50 MB/seg (Firewire 400) o 100 MB/seg (Firewire 800)	0.2 MB/seg (velocidad baja), 1.5 MB/seg (velocidad plena), 60 MB /seg (velocidad alta)	250 MB/seg por línea (1x); las tarjetas PCIe disponibles en 1x, 2x, 4x, 8x, 16x o 32x	300 MB/seg	300 MB/seg
Conectable en caliente	Sí	Sí	Depende del tamaño	Sí	Sí
Longitud máxima del bus (cable de cobre)	4.5 metros	5 metros	0.5 metros	1 metro	8 metros
Nombre del estándar	IEEE 1394, 1394b	USB Implementors Forum	PCI-SG	SATA-IO	T10 Committee

FIGURA 6.8 Características claves de cinco estándares de E/S dominantes. La columna tipo de bus indica si el bus ha sido diseñado para ser usado con cables externos al PC o dentro del computador con cables cortos o cableado en tarjetas de circuito impreso. PCIe soporta lecturas y escrituras simultáneas, por eso en algunas publicaciones se supone que las lecturas y escrituras se reparten al 50% y se duplica el ancho de banda por línea.

ser todo lo largo que se quiera sin preocuparse del desalineamiento del reloj o de problemas de sincronización. Todos los ejemplos de la figura 6.8 son asíncronos.

Para coordinar la transmisión de datos entre emisor y receptor, un bus asíncrono utiliza un **protocolo con acuse de recibo**, que consiste en una serie de pasos en los cuales el emisor y receptor pasan al siguiente paso sólo cuando ambas partes están de acuerdo. El protocolo se implementa utilizando un conjunto adicional de líneas de control.

Protocolo con acuse de recibo: serie de pasos utilizados para coordinar las transferencias en un bus asíncrono, en los cuales el emisor y receptor sólo pasan al siguiente paso cuando ambas partes acuerdan que el paso actual se ha completado.

Las interconexiones de E/S de los procesadores x86

La figura 6.9 muestra el sistema de E/S de un PC tradicional. El procesador se conecta a los periféricos a través de dos chips principales. El chip que está próximo al procesador es el controlador de memoria, frecuentemente denominado *puente norte*, y el que está conectado a él es el controlador de E/S, llamado *puente sur*.

El puente norte es básicamente un controlador DMA que conecta el procesador con la memoria, el bus gráfico AGP y el chip del puente sur. El puente sur conecta el puente norte a un montón de buses de E/S. Intel, AMD, NVIDIA y otros fabricantes ofrecen una amplia variedad de estos conjuntos de chips para conectar el procesador al mundo exterior.

La figura 6.10 muestra tres ejemplos del conjunto de chips. Observe que AMD eliminó el puente norte en el Opteron y productos posteriores, reduciendo de este modo en número de chips y la latencia de memoria y tarjetas gráficas.

Debido a la Ley de Moore, un número cada vez mayor de controladores de E/S que han estado antiguamente disponibles como placas opcionales de ampliación que se conectaban a buses de E/S han sido integradas en estos conjuntos de chips. Por ejemplo, el AMD Opteron X4 y el Intel Nehalem incluyen puente norte en el

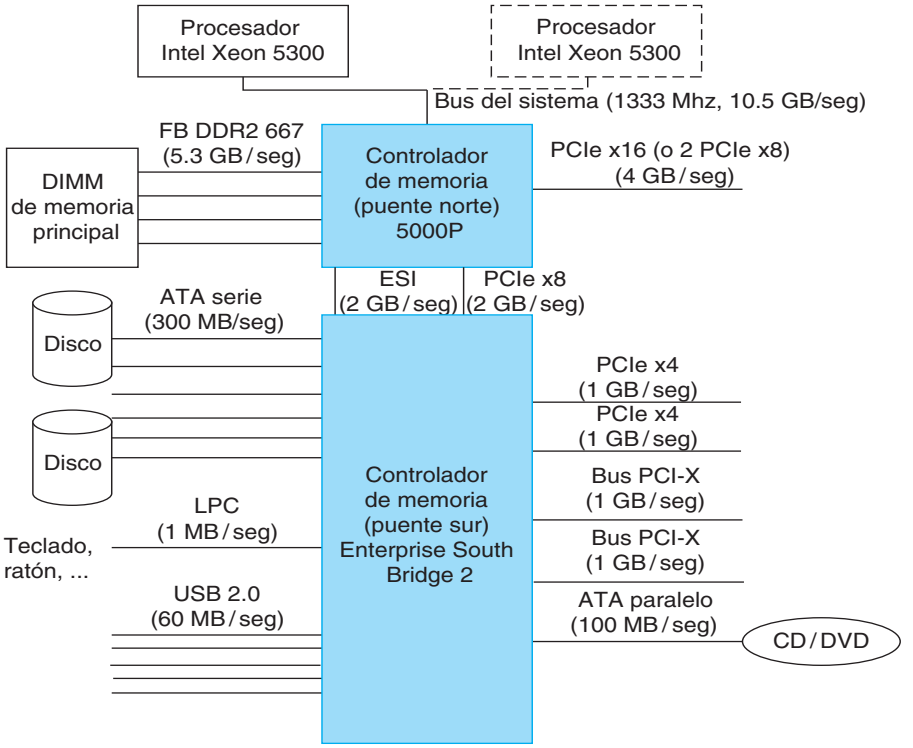


FIGURA 6.9 Organización del sistema de E/S en un servidor basado en Intel que utiliza el conjunto de chips 5000P. Si se supone que le tráfico se reparte a la mitad entre lectura y escrituras, se puede duplicar en ancho de banda por enlace del PCIe.

microprocesador, y en el chip del puente sur del Intel 975 se incorpora un controlador de RAID (véase sección 6.9).

Las interconexiones de E/S proporcionan interconexión eléctrica entre dispositivos E/S, procesadores y memoria, y también definen el protocolo de más bajo nivel para las comunicaciones. Por encima de este nivel básico, debemos definir protocolos hardware y software para controlar las transferencias de datos entre los dispositivos de E/S y la memoria, y especificar comandos del procesador para los dispositivos de E/S. Estos temas se tratan en la próxima sección.

Autoevaluación

Tanto redes como buses conectan componentes entre sí. ¿Cuáles de las siguientes frases son ciertas?

1. Las redes y los buses de E/S están casi siempre estandarizados.
2. Las redes de área local y los buses procesador-memoria son casi siempre síncronos.

	Conjunto de chips Intel 5000P	Conjunto de chips Intel 975X	AMD 580X CrossFiret
Segmento de mercado	Servidores	PC de altas prestaciones	Servidores/PC de altas prestaciones
Bus del sistema (64 bits)	1066/1333 MHz	800/1066 MHz	—
Controlador de memoria (puente norte)			
Nombre	Blackbird 5000P MCH	975X MCH	
Número de pines	1432	1202	
Tipo de memoria, frecuencia	DDR2 FBDIMM 667/533	DDR2 800/667/533	
Buses de memoria, anchura	4 x 72	1 x 72	
Número de DIMM, DRAM/DIMM	16, 1 GB/2 GB/4 GB	4, 1 GB/2 GB	
Capacidad máxima de la memoria	64 GB	8 GB	
¿Dispone de corrección de errores de memoria?	Si	No	
PCIe/ Interfaz gráfica externa	1 PCIe x16 o 2PCIe x	1 PCIe x16 0 2 PCIe x8	
Interfaz a puente sur	PCIe x8, ESI	PCIe x8	
Controlador de E/S (puente sur)			
Nombre	6321 ESB	ICH7	580X CrossFire
Empaquetado, pines	1284	652	549
Bus PCI: anchura, frecuencia	Dos de 62 bits, 133 MHz	32 bits, 33 MHz, 6 maestros	—
Puertos PCI Express	Tres PCIe x4		Dos PCIe x16, cuatro PCIe x1
Controlador MAC Ethernet, interfaz	—	1000/100/10 bit	—
Puertos USB 2.0, controladores	6	8	10
Puertos ATA, velocidad	Uno 100	Dos 100	Uno 133
Puertos ATA serie	6	2	4
Controlador de audio AC97, interfaz	—	Si	Si
Gestión de la E/S	SMbus 2.0, GPIO	Smbus 2.0, GPIO	ASF 2.0, GPIO

FIGURA 6.10 Dos conjuntos de chips de Intel y uno de AMD. Observe que las funciones del puente norte se incluyen en el microprocesador AMD, como en el más reciente Intel Nehalem.

6.6 Interfaz de los dispositivos de E/S al procesador, memoria y sistema operativo

Un protocolo de red o de bus define cómo se debe transmitir una palabra o bloque de datos mediante un conjunto de conductores. Además, aún existen otras tareas que se deben realizar para que realmente los datos sean transferidos desde un dispositivo al espacio de direccionamiento de memoria de un programa de usuario. Esta sección se centra en estas tareas y contestará a preguntas como las siguientes:

- ¿Cómo se transforma una petición de E/S del usuario en un comando de dispositivo y dicha petición se comunica al dispositivo?
- ¿Cómo se transfieren realmente los datos a o desde una posición de memoria?
- ¿Cuál es el papel del sistema operativo?

Tal como veremos al contestar a estas preguntas, el sistema operativo desempeña un papel importante en la gestión de la entrada-salida, actuando como interfaz entre el hardware y el programa que solicita la operación de E/S.

Las responsabilidades del sistema operativo surgen de tres características de los sistemas de E/S:

1. El sistema de E/S es compartido por los diferentes programas que usan el procesador.
2. Los sistemas de E/S a menudo usan interrupciones (las excepciones generadas externamente) para comunicar información sobre operaciones de E/S. Dado que las interrupciones provocan el paso al modo núcleo o supervisor, deben ser manejadas por el sistema operativo (SO).
3. El control de bajo nivel de un dispositivo es complejo porque requiere gestionar un conjunto de sucesos concurrentes, y porque los requisitos para el correcto control del dispositivo son frecuentemente muy minuciosos.

Interfaz hardware software

Las tres características de los sistemas de E/S mencionadas anteriormente conducen a varias de las funciones que el SO debe proporcionar:

- El SO garantiza que un programa de usuario tenga acceso solamente a las partes de un dispositivo de E/S a las cuales el usuario tiene derecho. Por ejemplo, el SO no debe permitir que un programa lea o escriba un fichero en disco si el propietario del fichero no ha concedido el acceso a este programa. En un sistema con dispositivos compartidos de E/S, no se podría proporcionar protección si los programas de usuario pudiesen realizar las operaciones de E/S directamente.
- El SO proporciona abstracciones para el acceso a los dispositivos suministrando rutinas que manejan las operaciones de nivel bajo con el dispositivo.
- El SO gestiona las interrupciones generadas por los dispositivos de E/S, de la misma forma que gestiona las excepciones generadas por un programa.
- El SO intenta proporcionar el acceso equitativo a los recursos compartidos de E/S, e intenta planificar los accesos de forma que aumente la productividad del sistema.

Para realizar estas funciones de parte de los programas de usuario, el sistema operativo debe poder comunicarse con los dispositivos de E/S y evitar que el programa del usuario pueda comunicarse directamente con estos dispositivos. Son necesarios tres tipos de comunicación:

1. El SO debe poder dar órdenes a los dispositivos de E/S. Estos comandos del SO incluyen no sólo operaciones como leer y escribir, sino también otras operaciones que llevan a cabo en el dispositivo, tal como la búsqueda en un disco.

2. El dispositivo de E/S debe poder notificar al SO cuando ha completado una operación o ha encontrado un error. Por ejemplo, cuando un disco termina una búsqueda, lo notificará al SO.
3. Los datos se deben transferir entre la memoria y un dispositivo de E/S. Por ejemplo, el bloque que es leído en una lectura de disco se debe mover desde disco a la memoria.

En las siguientes subsecciones veremos cómo se realizan estas comunicaciones.

Envío de comandos a los dispositivos de E/S

Para enviar un comando a un dispositivo de E/S, el procesador debe poder seleccionar o direccionar el dispositivo y proveer comandos de una o más palabras. Para direccionar un dispositivo se utilizan dos métodos: E/S asignada al espacio de memoria, e instrucciones especiales de E/S. En la **E/S asignada al espacio de memoria (*memory-mapped*)**, se asignan porciones del espacio de direccionamiento a los dispositivos de E/S. Las lecturas y escrituras a esas direcciones se interpretan como comandos para el dispositivo de E/S.

E/S asignada al espacio de memoria: esquema en el cual se asigna porciones del espacio de direccionamiento a los dispositivos de E/S, y las lecturas y escrituras a esas direcciones son interpretadas como comandos para el dispositivo de E/S.

Por ejemplo, una operación de escritura se puede utilizar para enviar datos a un dispositivo de E/S, donde los datos serán interpretados como un comando. Cuando el procesador pone la dirección y los datos en el bus de memoria, el sistema de memoria ignora la operación porque la dirección pertenece a una parte del espacio de memoria usado para E/S. El controlador del dispositivo, sin embargo, detecta la operación, recoge el dato y lo transmite al dispositivo como un comando. Los programas del usuario no pueden realizar operaciones de E/S directamente porque el SO no proporciona acceso al espacio de direccionamiento asignado a los dispositivos de E/S y las direcciones están protegidas por el mecanismo de conversión de direcciones. La E/S asignada al espacio de memoria se puede utilizar también para transmitir datos a través de la escritura o lectura en direcciones específicas. El dispositivo utiliza la dirección para determinar el tipo de comando, y los datos se pueden proporcionar por una escritura u obtener por una lectura. En cualquier caso, la dirección codifica tanto la identidad del dispositivo como el tipo de transmisión entre el procesador y el dispositivo.

La realización de una lectura o escritura de datos para satisfacer una petición del programa requiere realmente varias operaciones separadas de E/S. Además, el procesador puede tener que averiguar el estado del dispositivo entre comandos individuales para determinar si el comando terminó correctamente. Por ejemplo, una impresora sencilla tiene dos registros para el dispositivo de E/S: uno para la información sobre el estado y otro para que los datos sean impresos. El registro de Estado contiene un *bit de "finalización" (done bit)*, activado por la impresora cuando ha impreso un carácter, y un *bit de error*, que indica que la impresora está atascada o sin papel. Cada byte de datos que se va a imprimir se pone en el registro de Datos. El procesador debe entonces esperar hasta que la impresora active el bit de finalización antes de que pueda poner otro carácter en el búfer. El procesador debe también comprobar el bit de error para determinar si ha ocurrido algún problema. Cada una de estas operaciones requiere un acceso separado al dispositivo de E/S.

Extensión: La alternativa a la E/S asignada al espacio de memoria es utilizar instrucciones especiales de E/S del procesador. Estas instrucciones de E/S pueden especificar el número de dispositivo y la palabra del comando (o la posición de la palabra del comando en memoria). El procesador envía la dirección de dispositivo a través de una serie de líneas incluidas normalmente como parte del bus de E/S. El comando puede enviarse a través de las líneas de datos en el bus. Ejemplos de computadores con instrucciones de E/S son los Intel x86 y los IBM 370. Haciendo que la ejecución de instrucciones de E/S no esté permitida cuando no se está en modo núcleo o supervisor, se puede impedir o prevenir que los programas del usuario tengan acceso directo a los dispositivos.

Instrucciones especiales de E/S: instrucción dedicada que se utiliza para dar un comando a un dispositivo de entrada-salida y que especifica el número de dispositivo y la palabra del comando (o la posición de la palabra del comando en memoria).

Comunicación con el procesador

El proceso de comprobar periódicamente los bits del estado para ver si es el momento de realizar la siguiente operación de E/S, como en el ejemplo anterior, se llama **encuesta (*polling*)**. La encuesta es la manera más sencilla para que un dispositivo de entrada-salida se comunique con el procesador. El dispositivo de E/S simplemente pone la información en un registro de Estado, y el procesador debe obtener esta información. El procesador tiene todo el control y hace todo el trabajo.

Encuesta (polling): proceso de comprobar periódicamente el estado de un dispositivo de E/S para determinar la necesidad de dar servicio al dispositivo.

La encuesta o interrogación se puede utilizar de diferentes modos. Las aplicaciones para sistemas empotrados en tiempo real encuestan a los dispositivos de E/S puesto que las frecuencias de E/S están predeterminadas. Eso hace que el tiempo de sobrecarga de la E/S sea más predecible, lo cual es útil para tiempo real. Como veremos, esto permite que la encuesta pueda ser utilizada aun cuando la frecuencia de la E/S sea algo más alta.

La desventaja de la encuesta es que puede desperdiciar mucho tiempo de procesador porque los procesadores son mucho más rápidos que los dispositivos de E/S. Puede ocurrir que el procesador lea muchas veces el registro de Estado sólo para encontrar que el dispositivo todavía no ha terminado una operación de E/S que es comparativamente más lenta, o que el ratón no se ha movido desde la vez última que se realizó la encuesta. Cuando el dispositivo termina una operación, aún debemos leer el estado para determinar si se ejecutó con éxito.

El tiempo de sobrecarga de la E/S por encuesta fue reconocido hace tiempo y condujo a la propuesta de usar interrupciones para notificar al procesador que un dispositivo de E/S requiere la atención del procesador. La **E/S dirigida por interrupciones (*interrupt-driven I/O*)**, que es utilizada por casi todos los sistemas para al menos algunos dispositivos, emplea interrupciones de E/S para indicar al procesador que un dispositivo de E/S necesita atención. Cuando un dispositivo quiere notificar al procesador que ha terminado una cierta operación o necesita atención, provoca una interrupción al procesador.

E/S dirigida por interrupciones: esquema de E/S que emplea interrupciones para indicar al procesador que un dispositivo de E/S necesita atención.

Una interrupción de E/S es como las excepciones vistas en los capítulos 4 y 5, con dos diferencias importantes:

1. Una interrupción de E/S es asincrónica con respecto a la ejecución de la instrucción. Es decir, la interrupción no se asocia a ninguna instrucción y no impide que la instrucción en curso termine. Esto es muy diferente de las excepciones de fallo de página o de excepciones tales como el desbordamiento aritmético. Nuestra unidad de control sólo necesita comprobar si hay una interrupción de E/S pendiente cuando comienza una nueva instrucción.

2. Además del hecho de que ha ocurrido una interrupción de E/S, se quiere proporcionar información adicional tal como la identidad del dispositivo que ha generado la interrupción. Mas aún, las interrupciones representan dispositivos que pueden tener diferentes prioridades y sus peticiones de interrupción tienen diferentes niveles de urgencia asociada a ellas.

Para comunicar información al procesador, como la identidad del dispositivo que provoca la interrupción, un sistema puede utilizar interrupciones vectorizadas o un registro de Causa para la excepción. Cuando el procesador reconoce la interrupción, el dispositivo puede enviar la dirección del vector o un código de estado para que sea almacenado en el registro de Causa. En consecuencia, cuando el SO obtiene el control, sabe la identidad del dispositivo que causó la interrupción y puede interrogarlo inmediatamente. Un mecanismo de interrupción elimina la necesidad de que el procesador encueste al dispositivo y esto permite que el procesador se centre en ejecutar programas.

Niveles de prioridad de interrupción

Para tratar con las diversas prioridades de los dispositivos de E/S, la mayoría de los mecanismos de interrupción tienen varios niveles de prioridad: los sistemas operativos UNIX utilizan de cuatro a seis niveles. Estas prioridades indican el orden en el cual el procesador debe procesar interrupciones. Tanto las excepciones generadas internamente como las interrupciones externas de E/S tienen prioridades; normalmente, las interrupciones de E/S tienen una prioridad menor que las excepciones internas. Puede haber varias prioridades para las interrupciones de E/S, con los dispositivos de alta velocidad asociados a las prioridades más altas.

Para soportar los niveles de prioridad para las interrupciones, MIPS proporciona las primitivas que permiten que sea el sistema operativo quien implemente la política, similar a cómo MIPS maneja los fallos de TLB. La figura 6.11 muestra los registros claves, y la sección B.7 del apéndice B da más detalles.

El registro de Estado determina quién puede interrumpir al computador. Si el bit de habilitación (*enable*) de la interrupción es 0, entonces nada puede interrumpir. Un bloqueo más refinado de interrupciones está disponible en el campo de máscara de interrupción. Hay un bit en la máscara que corresponde a cada bit en el campo de la interrupción pendiente del registro de Causa. Para permitir la interrupción correspondiente, debe haber un 1 en el campo de la máscara en esa posición de bit. Una vez que ocurre una interrupción, el sistema operativo puede buscar la causa en el campo del código de excepción del registro de Estado: 0 significa que ocurrió una interrupción, y otros valores indican las excepciones mencionadas en el capítulo 7.

Aquí están los pasos que deben ocurrir en el manejo de una interrupción:

1. Realizar la operación Y lógica entre el campo de la interrupción pendiente y el campo de la máscara de interrupción para ver cuál de las interrupciones habilitadas podría ser la culpable. Se hace las copias de estos dos registros usando la instrucción `mfc0`.

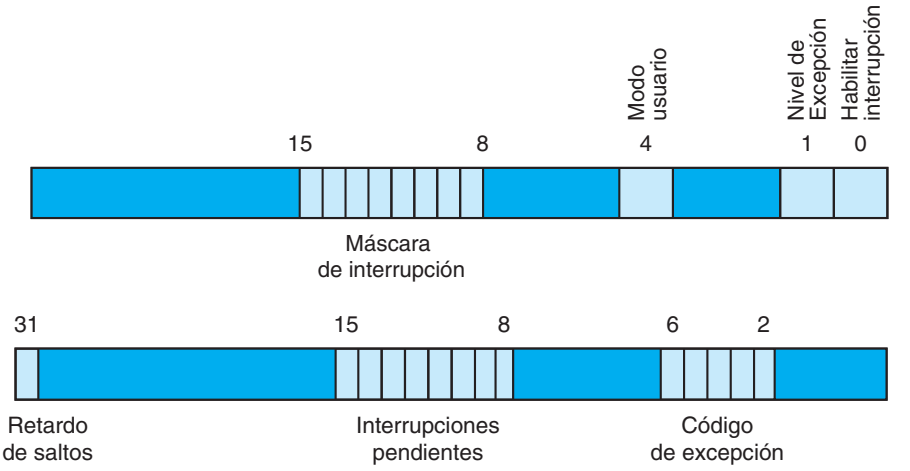


FIGURA 6.11 Los registros de Causa y Estado. Esta versión del registro de Causa corresponde a la arquitectura MIPS-32. La arquitectura MIPS 1 más antigua, para soportar anidación de interrupciones, tenía tres niveles anidados de conjuntos de bits de habilitación de interrupción, tanto para el modo núcleo como para el modo usuario. La sección B.7 en el apéndice B contiene má detalles acerca de estos registros.

2. Seleccionar de estas interrupciones la de mayor prioridad. Según la convención software, la prioridad más alta corresponde al extremo izquierdo.
3. Guardar el campo de la máscara de la interrupción del registro de Estado.
4. Cambiar el campo de la máscara de la interrupción para inhabilitar todas las interrupciones de prioridad igual o menor.
5. Guardar el estado de procesador necesario para manejar la interrupción.
6. Para permitir interrupciones de mayor prioridad, activar a 1 el bit de habilitación de la interrupción del registro de Causa.
7. Llamar a la rutina apropiada para la interrupción.
8. Antes de restaurar el estado, activar a 0 los bits de habilitación de interrupciones del registro de Causa. Esto permite restaurar el campo de máscara de la interrupción.

En las páginas de la B-36 a la B-37 del apéndice B se muestra un manejador de excepciones para una tarea de E/S simple.

¿Cómo corresponden los *niveles de prioridad de interrupción* (IPL) a estos mecanismos? El IPL es una propuesta asociada al sistema operativo. Se almacena en la memoria del proceso, y a cada proceso se da un IPL. En el IPL menor, se permiten todas las interrupciones. Inversamente, en el IPL mayor, se bloquean todas las interrupciones. Aumentar y disminuir el IPL implica realizar cambios en el campo de la máscara de interrupción del registro de Estado.

Extensión: Los dos bits menos significativos de los campos de interrupción pendiente y de máscara de interrupción se asignan a interrupciones software, que son las de menor prioridad. Estos bits son utilizados típicamente por interrupciones de mayor prioridad para asignar parte del trabajo a las interrupciones de menor prioridad una vez que el motivo inmediato que generó la interrupción de mayor prioridad se haya gestionado. Una vez que la interrupción de mayor prioridad acaba, se atienden y manejan las tareas de menor prioridad.

Transferencia de datos entre un dispositivo y memoria

Hemos visto dos métodos diferentes que permiten a un dispositivo comunicarse con el procesador. Estas dos técnicas (encuesta e interrupción de E/S) forman la base para dos métodos de implementación de la transferencia de datos entre el dispositivo de E/S y la memoria. Ambas técnicas son adecuadas para los dispositivos de menor ancho de banda, donde estamos más interesados en la reducción del coste del controlador de dispositivo y de la interfaz que en proporcionar una transferencia de alto ancho de banda. Tanto en la transferencia por encuesta como en la transferencia dirigida por interrupción es el procesador quién realiza el movimiento de datos y gestiona la transferencia. Después de revisar estos dos esquemas, examinaremos un esquema más adecuado para los dispositivos o grupos de dispositivos de más altas prestaciones.

Podemos utilizar el procesador para transferir datos entre un dispositivo y memoria basándonos en la encuesta. En aplicaciones de tiempo real, el procesador carga datos desde los registros del dispositivo de E/S y los almacena en memoria.

Un mecanismo alternativo es realizar la transferencia de los datos dirigida por interrupción. En este caso, el SO realiza la transferencia de datos en pequeñas cantidades de bytes desde o hacia el dispositivo. Sin embargo, dado que la operación de E/S se realiza por interrupción, el SO trabaja en otras tareas mientras los datos se están leyendo desde el dispositivo o se están escribiendo al dispositivo. Cuando el SO identifica una interrupción del dispositivo, lee el estado para comprobar si hay errores. Si no hay ninguno, el SO puede proporcionar los datos siguientes, por ejemplo, mediante una secuencia de escrituras en registros asignados a memoria. Cuando el último byte de una petición de E/S se ha transmitido y la operación de E/S ha finalizado, el SO puede informar al programa. El procesador y el SO hacen todo el trabajo en este proceso, teniendo acceso al dispositivo y a la memoria para cada uno de los datos transferidos.

La E/S dirigida por interrupción libera al procesador de tener que esperar cada acontecimiento de E/S, aunque si utilizamos este método para transferir datos desde o hacia un disco duro, la sobrecarga de tiempo todavía podría ser intolerable, puesto que podría consumir una fracción grande del procesador cuando el disco está realizando transferencias. Para dispositivos con ancho de banda elevado como los discos duros, las transferencias consisten básicamente en bloques de datos relativamente grandes (cientos o miles de bytes). Por ello, los diseñadores de computadores inventaron un mecanismo para descargar de trabajo al procesador y permitir que el controlador del dispositivo pueda transferir los datos directamente hacia o desde la memoria sin la participación del procesador. Este mecanismo se llama **acceso directo a memoria** (DMA). El mecanismo de interrupción todavía es utilizado por el dispositivo para comunicarse con el procesador, pero solamente al finalizar la transferencia de E/S o cuando ocurre un error.

Acceso directo a memoria: mecanismo que proporciona a un controlador de dispositivo la capacidad de transferir datos directamente hacia o desde la memoria sin la participación del procesador.

El DMA está implementado con un controlador especializado que transfiere datos entre un dispositivo de E/S y memoria independientemente del procesador.

El controlador de DMA se convierte en el **maestro del bus (*bus master*)** y dirige las lecturas o escrituras entre él mismo y la memoria. Hay tres pasos en una transferencia por DMA:

Maestro del bus: unidad en el bus que puede iniciar peticiones de bus.

1. El procesador programa el controlador de acceso directo a memoria (DMA) proporcionando la identidad del dispositivo, la operación para realizarse en el dispositivo, la dirección de memoria que es la fuente o el destino de los datos que se transferirán, y el número de bytes a transferir.
2. El DMA inicia la operación en el dispositivo y arbitra el acceso al bus. Cuando los datos están disponibles (procedentes del dispositivo o de la memoria), transfiere los datos. El controlador de DMA proporciona la dirección de memoria para las lecturas o escrituras. Si la petición requiere más de una transferencia por el bus, el controlador de DMA genera la dirección de memoria siguiente e inicia la siguiente transferencia. Usando este mecanismo, un controlador de DMA puede terminar una transferencia entera, que puede tener un tamaño de miles de bytes, sin molestar al procesador. Muchos controladores de DMA tienen algo de memoria para permitir que se adapten a los retardos producidos en la transferencia o los que se producen mientras que esperan la concesión del bus.
3. Una vez finalizada la transferencia por DMA, el controlador interrumpe al procesador, que puede entonces determinar si la operación completa terminó con éxito interrogando al controlador de DMA o examinando la memoria.

En un sistema informático puede haber varios controladores de DMA. Por ejemplo, en un sistema con un solo bus procesador-memoria y varios buses de E/S, cada controlador de bus de E/S tendrá normalmente un procesador de DMA que maneje cualquier transferencia entre un dispositivo en el bus de E/S y la memoria.

A diferencia de la E/S por encuesta o la dirigida por interrupción, el DMA se puede utilizar para interconectar o hacer la interfaz a un disco duro sin consumir todos los ciclos del procesador para las operaciones de E/S. Por supuesto, si el procesador también quiere acceder a la memoria, tendrá que esperar cuando ésta está ocupada atendiendo una transferencia por DMA. Usando caches, el procesador puede evitar tener que acceder a memoria la mayor parte del tiempo, de modo que deja la mayor parte del ancho de banda de la memoria libre, para uso de los dispositivos de E/S.

Extensión: Para reducir aún más la necesidad de interrumpir al procesador y de ocuparlo en la gestión de las peticiones de la E/S, el controlador de entrada-salida puede hacerse más inteligente. Los controladores inteligentes a menudo se llaman *procesadores de E/S* (también *controladores de E/S* o *controladores de canal*). Estos procesadores especializados ejecutan básicamente una serie de operaciones de E/S, denominada *programa de E/S*. El programa se puede almacenar en el procesador de E/S, o puede estar almacenado en memoria y ser buscado por el procesador de E/S. Al usar un procesador de E/S, el sistema operativo típicamente instala un programa de E/S que indique las operaciones de E/S a realizar, así como el tamaño y la dirección de la transferencia para las lecturas y escrituras. El procesador de E/S entonces toma las operaciones del programa de E/S e interrumpe al procesador solamente cuando se termina todo el programa. Los procesadores de DMA son esencialmente procesadores de propósito especial (generalmente un solo chip y no programable), mientras que los procesadores de E/S se implementan frecuentemente con microprocesadores de propósito general, que ejecutan un programa especializado de E/S.

El acceso directo a memoria y el sistema de memoria

Cuando se incorpora el DMA en un sistema de E/S, la relación entre el sistema de memoria y el procesador cambia. Sin el acceso directo a memoria, todos los accesos al sistema de memoria proceden del procesador y, por tanto, pasan por el mecanismo de conversión de dirección y por el acceso a la cache como si el procesador generara las referencias. Con DMA, hay otro camino para llegar al sistema de memoria, que no pasa por el mecanismo de conversión de dirección o la jerarquía de memoria. Esta diferencia genera algunos problemas tanto en los sistemas con memoria virtual como en los sistemas con cache. Estos problemas se solucionan generalmente con una combinación de técnicas hardware y de soporte software.

Las dificultades al incorporar DMA en un sistema de memoria virtual se presentan porque las páginas tienen direcciones físicas y direcciones virtuales. El acceso directo de memoria también crea problemas en los sistemas con cache, porque puede haber dos copias de un dato: una en la cache y otra en memoria. Dado que procesador de DMA envía peticiones directamente a memoria sin pasar por la cache del procesador, el valor de la posición de memoria considerado por el controlador de DMA puede ser diferente del valor visto por el procesador. Considere una lectura en el disco, que el controlador de DMA coloca directamente en memoria. Si algunas de las posiciones en las cuales el DMA escribe están en la cache, el procesador recibirá un valor obsoleto cuando realice una lectura. Asimismo, si la cache es de escritura retardada (*write-back*) el DMA puede leer un valor directamente de la memoria cuando un valor más reciente está en la cache, y dicho valor no se ha reescrito aún en la memoria (*write-back)*. A esto se le llama *problema de los datos obsoletos* o *problema de coherencia* (véase capítulo 5).

Hemos visto tres métodos diferentes para transferir datos entre un dispositivo de E/S y la memoria. Al pasar desde la E/S por encuesta a una E/S dirigida por interrupción y, finalmente, a una interfaz de E/S por acceso directo a memoria, pasamos la carga de gestionar una operación de E/S desde el procesador a controladores de E/S progresivamente más inteligentes. Estos métodos tienen la ventaja de liberar ciclos del procesador. Su desventaja es que aumentan el coste del sistema de E/S. Por este motivo, un determinado sistema informático puede elegir qué punto, a lo largo de este espectro de posibilidades, es apropiado para los dispositivos de E/S conectados con él.

Antes de tratar el diseño de los sistemas de E/S, veamos brevemente cómo se miden sus prestaciones.

Autoevaluación

En el *ranking* de las tres maneras de hacer E/S, ¿qué afirmaciones son ciertas?

1. Si queremos la menor latencia para una operación de E/S a un único dispositivo de E/S, el orden es encuesta, DMA e interrupción.
2. En términos de menor impacto en la utilización del procesador por un único dispositivo de E/S, el orden es DMA, interrupción y encuesta.

Interfaz hardware software

En un sistema con memoria virtual, ¿debería el DMA trabajar con direcciones lógicas o con direcciones físicas? La dificultad obvia con las direcciones lógicas es que el controlador de DMA necesitará traducir las direcciones lógicas a direcciones físicas. El problema principal cuando se usa una dirección física en una transferencia por DMA es que la transferencia no puede cruzar fácilmente los límites de la página. Si una operación de E/S cruzara un límite de página, entonces las posiciones de memoria involucradas en la transferencia podrían no estar necesariamente contiguas en la memoria virtual. Por lo tanto, si utilizamos direcciones físicas, debemos restringir las transferencias de DMA de forma que cada una de ellas sólo implique direcciones dentro de una página.

Un método que permite al sistema iniciar transferencias de DMA que sobrepasen los límites de la página es hacer que el DMA trabaje con direcciones virtuales. En estos sistemas, el controlador de DMA tiene unos cuantos registros que proporcionan la correspondencia entre páginas virtuales y físicas para una transferencia. El SO proporciona la información de correspondencia cuando se inicia la operación de E/S. Con información de correspondencia, el controlador de DMA no necesita preocuparse de la posición ocupada por las páginas virtuales involucradas en la transferencia.

Otra técnica consiste en hacer que el sistema operativo descomponga la transferencia de DMA en una serie de transferencias más pequeñas, cada una de ellas confinada dentro de los límites de una única página física. Las transferencias entonces son *encadenadas* y gestionadas juntas por un procesador de E/S o por un controlador inteligente de DMA que ejecuta la secuencia entera de transferencias; alternativamente, el SO puede solicitar individualmente cada una de las transferencias.

Cualquiera que sea el método utilizado, el sistema operativo aún debe cooperar para asegurar que no se cambie la ubicación de una página en memoria mientras en esa página se está haciendo una transferencia DMA.

Interfaz hardware software

El problema de la coherencia para los datos de E/S puede ser resuelto usando una de tres importantes técnicas. Una aproximación consiste en encaminar las operaciones de E/S a través de la cache. Esto asegura que las lecturas obtienen el valor más reciente mientras que las escrituras actualizan cualquier dato en la cache. Encaminar toda la E/S a través de la cache es costoso y tiene un fuerte impacto negativo en las prestaciones del procesador, puesto que los datos de E/S raramente se utilizan inmediatamente y pueden desplazar fuera de la cache datos de un programa en ejecución. Una segunda opción consiste en hacer que el SO selectivamente invalide el contenido de la cache antes de una operación de lectura de E/S, o que fuerce la actualización de la memoria principal (*write-backs*) antes de que ocurra una operación de escritura de E/S (a veces denominado *vaciado de cache* o *cache flushing)*. Esta técnica requiere un pequeño soporte por parte del hardware y es probablemente más eficiente si el software realiza la función fácil y eficientemente. Como el vaciado de grandes trozos de la cache sólo ocurre en caso de transferencias de DMA, será relativamente infrecuente. La tercera alternativa es proporcionar un mecanismo hardware para invalidar selectivamente entradas de la cache. Este mecanismo de invalidación por hardware para asegurar coherencia de la cache es típico en sistemas multiprocesador, y también se puede utilizar para E/S; este asunto se discute en detalle en el capítulo 5.

6.7 Medidas de las prestaciones de la E/S: ejemplos de discos y sistema de ficheros

¿Cómo debemos comparar sistemas de E/S? Esta pregunta es compleja porque las prestaciones de la E/S dependen de muchos aspectos del sistema y porque distintas aplicaciones ponen énfasis en distintos aspectos del sistema de E/S. Además, un diseño puede suponer complicados compromisos entre el tiempo de respuesta (latencia) y la productividad (*throughput*), haciendo imposible medir un aspecto aislado. Por ejemplo, gestionando una petición tan pronto como sea posible, generalmente se minimiza el tiempo de respuesta, aunque puede lograrse una mayor productividad si intentamos manejar juntas peticiones relacionadas. Por consiguiente, podemos aumentar la productividad en un disco agrupando peticiones que acceden a posiciones que están cerca una de otra. Tal política aumentará el tiempo de respuesta para algunas peticiones, llevando probablemente a una variación mayor en el tiempo de respuesta. Aunque la productividad será mayor, algunos *benchmarks* (programas de prueba) están restringidos por un tiempo de respuesta máximo para cualquier petición, haciendo dichas optimizaciones potencialmente problemáticas.

Aquí damos algunos ejemplos de las medidas propuestas para determinar el rendimiento de los sistemas de disco. Estos *benchmarks* (programas de prueba) están afectados por varias características del sistema, incluyendo la tecnología del disco, la conexión de los discos, el sistema de memoria, el procesador y el sistema de ficheros proporcionado por el sistema operativo.

Procesamiento de transacciones: tipo de aplicación que implica manejar pequeñas operaciones cortas (llamadas transacciones) que típicamente requieren E/S y cómputo. Las aplicaciones de procesamiento de transacciones típicamente tienen tanto requisitos de tiempo de respuesta como una medida de prestaciones basada en la productiviad de las transacciones.

Velocidad de E/S: medida del rendimiento de E/S por unidad de tiempo, tal como lecturas por segundo.

Velocidad de datos: medida del rendimiento de bytes por unidad de tiempo, tal como GB/segundo.

Antes de estudiar estos *benchmarks*, necesitamos tratar un punto confuso sobre terminología y unidades. El rendimiento de los sistemas de E/S depende del ritmo o la velocidad con que el sistema transfiere datos. La velocidad de transferencia depende de la frecuencia de reloj, que típicamente se da en GHz (10^9 ciclos por segundo). La velocidad de transferencia se da generalmente en GB/seg. En sistemas de E/S, se mide GBs usando base 10 (es decir, 1 GB = 10^9 = 1 000 000 000 bytes), a diferencia de la memoria principal donde se utiliza base 2 (es decir, 1 GB = 2^{30} = 1 073 741 824). Además de crear confusión, esta diferencia introduce la necesidad de convertir entre base 10 (1K = 1000) y base 2 (1K = 1024) porque muchos accesos de E/S se realizan con bloques de datos que tienen un tamaño que es una potencia de dos. En lugar de complicar todos los ejemplos convirtiendo exactamente una de las dos medidas, aquí hacemos esta observación e indicamos que tratando las dos medidas como si las unidades fueran idénticas se introduce un error pequeño. Este error se ilustra en la sección 6.12.

Benchmark de E/S para procesamiento de transacciones

Las aplicaciones de **procesamiento de transacciones** (*Transaction Processing, TP*) implican tanto unos requisitos de tiempo de respuesta como una medida de las prestaciones basadas en la productividad. Además, la mayor parte de los accesos de E/S son pequeños. Debido a esto, las aplicaciones TP se ocupan principalmente de la **velocidad de E/S**, medida como el número de accesos a disco por segundo, en comparación con la **velocidad de datos**, medida como bytes de datos por segundo. Las aplicaciones TP generalmente implican cambios en una base de datos grande, con un sistema que alcanza

algunos requisitos de tiempo de respuesta y manipula elegantemente ciertos tipos de fallos. Estas aplicaciones son extremadamente críticas y sensibles al coste. Por ejemplo, los bancos normalmente utilizan sistemas TP porque están interesados en una serie de características; entre ellas, asegurarse de que las transacciones no se pierdan, se procesen rápidamente y que el coste de procesamiento de cada una de ellas sea el mínimo. Aunque la confiabilidad frente a fallos es un requisito absoluto en estos sistemas, el tiempo de respuesta y la productividad son críticos para construir sistemas rentables.

Se han desarrollado *benchmarks* para el procesamiento de transacciones. El sistema más conocido es una serie desarrollada por el Consejo para el Procesamiento de Transacciones (TPC).

TPC-C, creado inicialmente en 1992, simula un ambiente complejo de consultas. TPC-H modela sistemas de apoyo para la toma de decisiones con fines específicos; las consultas no están relacionadas y el conocimiento de las consultas anteriores no se puede utilizar para optimizar las consultas futuras; el resultado es que los tiempos de ejecución de la consulta pueden ser muy largos. TPC-W es un *Benchmark* de transacciones basado en Web que simula las actividades de un servidor Web orientado a las transacciones comerciales. El *benchmark* prueba el sistema de base de datos así como el software subyacente del servidor Web. TPC-App es un benchmark para servidores de aplicaciones y servicios web. El más reciente es TPC-E que simula una carga de trabajo de procesamiento de transacciones en una firma de agentes de bolsa. Los *benchmarks* TPC se describen en *www.tpc.org*.

Todos los *benchmarks* TPC miden las prestaciones en transacciones por segundo. Además, incluyen restricciones de tiempo de respuesta, de forma que se miden prestaciones o productividad solamente cuando se cumple el límite del tiempo de respuesta. Para modelar sistemas del mundo real, también se asocian velocidades de transacciones más altas con sistemas más grandes, tanto en términos de usuarios como del tamaño de la base de datos donde se aplican las transacciones. De este modo, la capacidad de almacenamiento debe escalar con las prestaciones. Finalmente, el coste del sistema también se debe incluir para permitir comparaciones exactas del coste-prestaciones.

Benchmark de E/S para Web y Sistemas de Ficheros

Además de *benchmarks* del procesador, SPEC ofrece un *benchmark de servidor de ficheros* (SPECSFS) y un *benchmarks* servidor de Web (SPECWeb). SPECSFS es un *benchmarks* para medir prestaciones NFS (*Network File System*) que utiliza un guión (*script)* de peticiones al servidor de ficheros; comprueba las prestaciones del sistema de E/S, incluyendo disco y red de E/S, así como el procesador. SPECSFS es un *benchmark* orientado a productividad, pero con requisitos importantes del tiempo de respuesta. SPECWeb es un *benchmark* del servidor que simula varios clientes que solicitan las páginas estáticas y dinámicas de un servidor, así como clientes que envían datos al servidor, (véase capítulo 1).

El esfuerzo más reciente de SPEC se dirige hacia la medida del consumo de potencia. Así, SPECPower mide características de consumo de potencia y prestaciones en pequeños servidores.

Sun ha anunciado reciente un entorno de programas de prueba de sistemas de ficheros, *filenbench*. En lugar de una carga de trabajo estándar, proporciona un lenguaje que permite crear las cargas de trabajo que nos gustaría tener en nuestro

sistema de ficheros. Sin embargo, hay ejemplos de cargas de trabajo de ficheros que se supone que emulan aplicaciones comunes del sistema de ficheros.

Autoevaluación ¿Son las siguientes afirmaciones verdaderas o falsas? A diferencia de los *benchmarks* del procesador, los de E/S:

1. Se centran en la productividad (*throughput*) más que en la latencia.
2. Pueden requerir que el conjunto de datos escale en tamaño o número de usuarios para alcanzar hitos en prestaciones.
3. A menudo indican coste-prestaciones.

6.8 Diseño de un sistema de E/S

Los diseñadores de sistemas de E/S encuentran básicamente dos tipos de especificaciones: limitaciones de latencia y limitaciones de ancho de banda. En ambos casos, el conocimiento del patrón de tráfico afecta al diseño y al análisis.

Los requerimientos de latencia suponen asegurar un límite en la latencia para completar una operación de E/S. En el caso más sencillo, el sistema puede estar descargado, y el diseñador debe asegurar que se satisface un cierto límite para la latencia, bien porque el límite es crítico para la aplicación, bien porque el dispositivo debe tener el servicio garantizado para evitar errores. Determinar la latencia de un sistema descargado es relativamente fácil porque sólo requiere el seguimiento del camino de la operación de E/S y la suma de las latencias individuales.

Encontrar la latencia media (o distribución de latencias) bajo una cierta carga de trabajo es un problema mucho más complejo. Este tipo de problemas se aborda mediante teoría de colas (cuando el comportamiento de las peticiones de la carga de trabajo y los tiempos de servicio de E/S pueden aproximarse mediante distribuciones simples) o mediante simulación (cuando el comportamiento de los sucesos de E/S es complejo). Ambas cuestiones están fuera del alcance de este texto.

El otro problema habitual que los diseñadores del sistema de E/S deben abordar es el de satisfacer una serie de limitaciones relativas al ancho de banda para una determinada carga de trabajo. Alternativamente, se puede plantear una versión simplificada de este problema. En este caso se parte de un sistema de E/S parcialmente configurado, y el diseñador debe equilibrar este sistema de forma que se mantenga el máximo ancho de banda alcanzable dentro de lo establecido por la parte preconfigurada del sistema.

La forma general de abordar el diseño de este tipo de sistemas es la siguiente:

1. Encontrar el elemento más débil del sistema de E/S, que es el componente del camino de E/S que va a limitar el diseño. Dependiendo de la carga de trabajo, este componente puede ser cualquiera: la CPU, el sistema de memoria, el bus de E/S, los controladores de E/S o los dispositivos. Tanto la carga de trabajo como los límites de configuración pueden determinar dónde se localiza el elemento más débil.
2. Configurar este elemento para mantener el ancho de banda requerido.
3. Determinar los requerimientos para el resto del sistema y configurarlo para mantener este ancho de banda.

La forma más sencilla de entender esta metodología es mediante un ejemplo. En la sección 6.10 haremos un análisis sencillo del sistema de E/S del sevidor Sun Fire x4150 para mostrar el funcionamiento de esta metodología.

6.9 Paralelismo y E/S: conjuntos redundantes de discos económicos

La ley de Amdahl del capítulo 1 nos recuerda que olvidarse de la E/S en esta revolución paralela es imprudente. Para demostrarlo, sirva este ejemplo sencillo.

Impacto de la E/S en las prestaciones del sistema

EJEMPLO

Supongamos que se dispone de un programa de prueba que se ejecuta en 100 segundos de tiempo total, de los cuales 90 segundos corresponden a tiempo de CPU y el resto es tiempo de E/S. Supongamos que el número de procesadores se duplica cada dos años, pero que la velocidad de los procesadores no cambia y que el tiempo de E/S no se mejora. ¿Cuánto más rápido se ejecutará nuestro programa al cabo de seis años?

RESPUESTA

Sabemos que

$$\text{tiempo total} = \text{tiempo de CPU} + \text{tiempo de E/S}$$

$$100 = 90 + \text{tiempo de E/S}$$

$$\text{tiempo de E/S} = 10 \text{ segundos}$$

La siguiente tabla muestra los tiempo de CPU de los nuevos procesadores y el tiempo total resultante.

Después de *n* años	Tiempo de CPU	Tiempo de E/S	Tiempo total	% tiempo E/S
0 años	90 segundos	10 segundos	100 segundos	10%
2 años	$\frac{90}{2}$ = 45 segundos	10 segundos	55 segundos	18%
4 años	$\frac{45}{2}$ = 23 segundos	10 segundos	33 segundos	31%
6 años	$\frac{23}{2}$ = 11 segundos	10 segundos	21 segundos	47%

La mejora en las prestaciones de la CPU al cabo de seis años es

$$\frac{90}{11} = 8$$

Sin embargo, la mejora en el tiempo total de ejecución es solamente

$$\frac{100}{21} = 4.7$$

y el tiempo de E/S ha pasado del 10% al 47% del tiempo total.

Así, la revolución paralela tiene que llegar también a la E/S y no sólo a la computación, o el esfuerzo invertido en la paralelización podría desaprovecharse en la E/S de los programas, que siempre va a necesitarse.

La motivación original de los conjuntos de discos fue la mejora de las prestaciones de la E/S (véase **sección 6.14** en el CD). A finales de los años 80, la única alternativa para el almacenamiento de altas prestaciones eran discos grandes y caros, como los discos de mayor tamaño de la figura 6.4. El argumento era que mediante el reemplazo de unos pocos discos grandes por muchos discos pequeños, las prestaciones podrían mejorarse porque habría más cabezas de lectura. Este cambio era apropiado también para los sistemas multiprocesador, porque tener muchas cabezas de lectura/escritura implica que el sistema de almacenamiento puede permitir muchos más accesos independientes, así como transferencias de grandes conjuntos de datos distribuidas entre muchos discos. Es decir, permitiría muchas operaciones de E/S por segundo y frecuencias de transferencia de datos más elevadas. Además, podría tener ventajas en el coste, potencia y espacio, porque los discos más pequeños son, en general, más eficientes por gigabyte que los discos grandes.

La desventaja de este argumento era que podría empeorar significativamente la fiabilidad. Estos discos más pequeños y baratos tenían MTTF menores que los discos grandes, y aún más importante, al reemplazar un disco por, digamos, 50 discos pequeños, ¡la razón de fallos aumentaría al menos en un factor 50!

La solución fue incorporar redundancia para que el sistema pudiera enfrentarse con los fallos de los discos sin perder información. Al disponer de muchos discos pequeños, el coste de la redundancia para mejorar la confiabilidad es pequeño en comparación con soluciones basadas en unos pocos discos grandes. De este modo, la confiabilidad era más asequible si se construía un conjunto redundante de discos económicos. De ahí su nombre: **conjunto redundante de discos económicos** (*redundant array of inexpensive disks*), abreviado **RAID**.

Conjunto redundante de discos económicos (RAID): organización de los discos que utiliza un conjunto de discos pequeños y baratos para aumentar las prestaciones y la confiabilidad.

Retrospectivamente, aunque su desarrollo estuvo motivado por las prestaciones, la confiabilidad fue un aspecto clave en la gran popularidad de los sistemas RAID. La revolución paralela ha hecho aflorar de nuevo el argumento original de las prestaciones de los sistemas RAID. El resto de esta sección examina las opciones para mejorar la confiabilidad y su impacto en el coste y las prestaciones.

¿Cuánta redundancia se necesita? ¿Se necesita información adicional para encontrar los fallos? ¿Es importante la organización de los datos y la información extra de comprobación de errores en estos discos? El artículo que acuñó el término proporcionó respuestas de distinta complejidad a estas cuestiones, comenzando con la solución más simple pero cara. La figura 6.12 muestra distintas

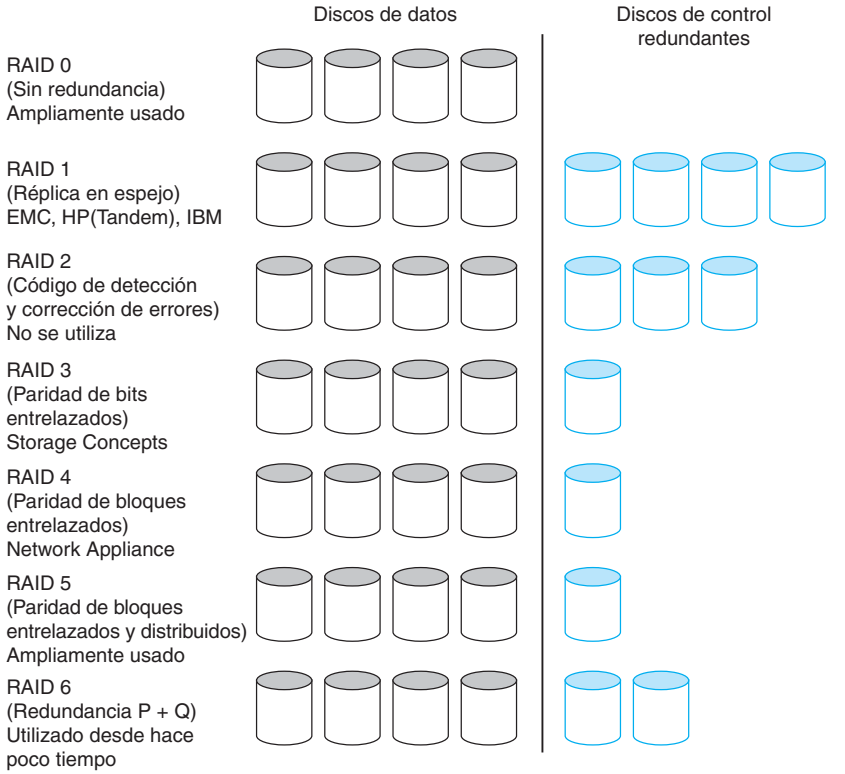


FIGURA 6.12 Organizaciones RAID para un ejemplo de cuatro discos de datos donde se muestran los discos adicionales de control de errores asociados a cada nivel RAID, así como las empresas que utilizan cada nivel. Las figura 6.13 y 6.14 explican las diferencias entre RAID 3, RAID 4 y RAID 5.

organizaciones de discos y ejemplos del coste asociado en términos del número de discos extra de comprobación. Para seguir la evolución de las distintas soluciones, los autores numeraron las organizaciones RAID; numeración que aún se utiliza.

Sin redundancia (RAID 0)

Simplemente se distribuyen los datos en varios discos, lo cual se denomina **desmontado**. Esta operación fuerza automáticamente a que los accesos se realicen a varios discos. La distribución a lo largo de un conjunto de discos hace que el conjunto sea percibido por el software como un único disco grande, lo cual simplifica la gestión del almacenamiento. También mejora las prestaciones de los accesos de grandes cantidades de datos, ya que muchos discos funcionan en paralelo. Los sistemas de edición de vídeo, por ejemplo, a menudo distribuyen los datos y puede que no se preocupen de la fiabilidad tanto como en el caso de las bases de datos.

Desmontado: asignación de bloques lógicos secuenciales a discos distintos para conseguir mejores prestaciones que la que un único disco puede proporcionar.

A veces el término RAID 0 es inapropiado ya que no existe redundancia. Sin embargo, a menudo los niveles RAID son establecidos por un informático cuando crea un sistema de almacenamiento, y RAID 0 es una de las posibles opciones. Por ello, el término *RAID 0* ha sido ampliamente utilizado.

Réplica en espejo (RAID 1)

Réplica en espejo: escritura de datos idénticos en varios discos para incrementar la disponibilidad de los datos.

Esta organización tradicional que se utiliza para tolerar fallos de disco, denominada **réplica en espejo** o *vigía*, utiliza el doble de discos que utiliza RAID 0. Cuando los datos se escriben en uno de los discos, estos datos también se escriben en un disco redundante, de forma tal que existen siempre dos copias de la información. Si un disco falla, el sistema accede a su copia y lee el contenido para obtener la información deseada. La organización réplica en espejo es la solución RAID más cara, ya que requiere la mayor cantidad de discos.

Detección de errores y código de corrección (RAID 2)

RAID 2 utiliza un esquema de detección y corrección de errores que es utilizado muy frecuentemente en las memorias (véase el **apéndice C**). Ya que RAID 2 ha caído en desuso, no lo describiremos aquí.

Paridad de bits entrelazados (RAID 3)

Grupo de protección: conjunto de discos o bloques de datos que comparte un mismo disco o bloque de códigos de control de errores.

El coste de una mayor disponibilidad puede ser reducido a $1/N$, donde N es el número de discos de un **grupo de protección**. En vez de tener una copia completa de los datos originales de cada disco, sólo necesitamos añadir la información redundante suficiente para restaurar la información que pierde en un fallo. Las lecturas o escrituras se dirigen a todos los discos del grupo, con un único disco adicional que almacena la información para el control de errores en caso de que exista un fallo. RAID 3 es popular en aplicaciones con grandes conjuntos de datos, tales como las aplicaciones multimedia y algunos códigos científicos.

La *paridad* es uno de tales esquemas. Los lectores que no estén familiarizados con la paridad pueden pensar que el disco redundante contiene la suma de todos los datos de los otros discos. Cuando un disco falla, entonces se sustrae todos los datos de los discos que funcionan correctamente del disco de paridad; la información restante debe coincidir con la información que falta. La paridad es simplemente la suma módulo dos.

A diferencia de RAID 1, muchos discos deben ser leídos para determinar los datos que faltan. El supuesto que hay detrás de esta técnica es que tardar más en reponerse de un fallo pero gastar menos en almacenamiento redundante es un buen compromiso.

Paridad de bloques entrelazados (RAID 4)

RAID 4 utiliza la misma proporción entre discos de datos y discos de control de errores que utiliza RAID 3, pero el acceso a los datos es distinto. La paridad se almacena en bloques y está asociada a un conjunto de bloques de datos.

En RAID 3, cada acceso se dirige a todos los discos. Sin embargo, algunas aplicaciones prefieren accesos más pequeños, lo cual permite que accesos independientes ocurran en paralelo. Este es el objetivo de los niveles 4 a 6 de RAID. Puesto que la información de detección de errores en cada sector se analiza en las lecturas para ver si los datos son correctos, las "pequeñas lecturas" a cada disco sólo pueden producirse de forma independiente si el acceso mínimo es un sector. En el contexto RAID, un acceso pequeño se dirige a un único disco en un grupo de protección, mientras que un acceso grande se dirige a todos los discos de un grupo de protección.

Las escrituras constituyen otro aspecto de análisis tal y como aparece en la figura 6.13. Parece que cada pequeña escritura requiera acceder a todos los demás

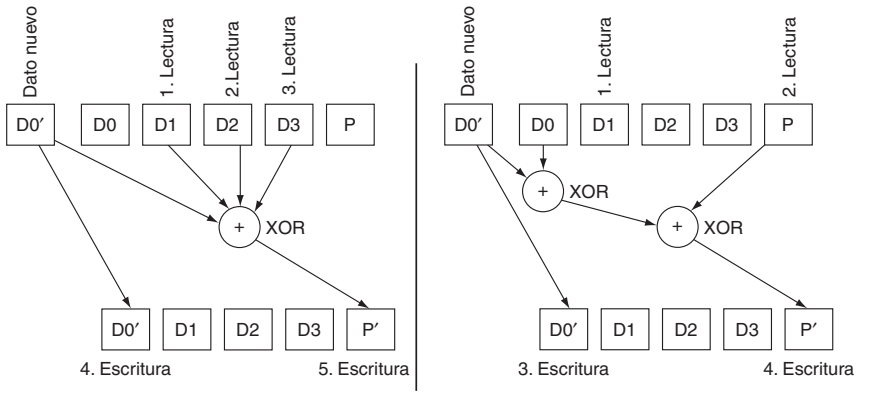


FIGURA 6.13 Actualización de una pequeña escritura sobre RAID 3 en comparación con RAID 4. Esta optimización para pequeñas escrituras reduce el número de accesos a discos, así como el número de discos ocupados. Esta figura supone que tenemos cuatro bloques de datos y un bloque de paridad. El cálculo sencillo de paridad en RAID 3 que aparece a la izquierda de la figura lee los bloques D1, D2 y D3 antes de sumar el bloque D0' para calcular la nueva paridad P'. (En el caso de que usted se lo estuviera preguntando, el nuevo dato D0' viene directamente desde la CPU, así que los discos no están implicados en su lectura). El atajo proporcionado por RAID 4 que se representa a la derecha lee el valor antiguo D0 y lo compara con el nuevo valor D0' para observar qué bits cambian. Entonces, se lee la paridad antigua P y después se modifican los correspondientes bits para generar P'. La función lógica OR exclusiva hace esta operación. En este ejemplo se sustituyen tres lecturas a disco (D1, D2, D3) y dos escrituras a disco (D0', P') en las cuales se involucran todos los discos, por dos lecturas a disco (D0, P) y dos escrituras a disco (D0', P') lo cual involucra sólo a dos discos. Aumentando el tamaño del grupo de paridad se aumenta el ahorro que proporciona el atajo. RAID 5 utiliza el mismo atajo.

discos para leer el resto de la información necesaria para recalcular la nueva paridad. Una "pequeña escritura" requiere leer los datos antiguos y la paridad antigua, añadir la nueva información y después escribir la nueva paridad al disco de paridad y los nuevos datos al disco de datos.

El aspecto clave para reducir este tiempo añadido es que la paridad coincide con una suma de información; observando qué bits cambian cuando se escribe la nueva información, necesitamos cambiar sólo los correspondientes bits en el disco de paridad. La parte derecha de la figura 6.13 muestra el atajo. Debemos leer los datos antiguos del disco que van a ser escritos, se comparan los datos antiguos con los datos nuevos para ver qué bits cambian, se lee la paridad antigua, se cambian los correspondientes bits, y finalmente se escriben los nuevos datos y nueva paridad. Por ello, la pequeña escritura implica cuatro accesos a disco a dos únicos discos en lugar del acceso a todos los discos. Esta organización es RAID 4.

Paridad distribuida de bloques entrelazados (RAID 5)

RAID 4 soporta eficientemente una combinación de lecturas y escrituras de grandes bloques de datos, así como lecturas y escrituras de pequeños bloques de datos. Un inconveniente consiste en que el disco de paridad debe ser actualizado en cada escritura, y por eso el disco de paridad crea un cuello de botella cuando se producen escrituras consecutivas.

Para solucionar el cuello de botella generado por las escrituras de la paridad, la información de paridad puede ser distribuida a lo lardo de todos los discos de

forma tal que no exista un único cuello de botella para las escrituras. La organización distribuida de la paridad se denomina RAID 5.

La figura 6.14 muestra cómo están distribuidos los datos en RAID 4 y RAID 5. Como se observa en la organización de la derecha, en RAID 5 la paridad asociada con cada fila de bloques de datos no está exclusivamente restringida a un único disco. Esta organización permite que múltiples escrituras se realicen simultáneamente siempre y cuando los bloques de paridad no se encuentren en el mismo disco. Por ejemplo, en una escritura al bloque 8 de la derecha se debe también acceder al bloque de paridad P2, por lo que se ocupan los discos primero y tercero. Una segunda escritura al bloque 5 de la derecha, implica una actualización de su bloque de paridad P1 y accesos al segundo y cuarto disco, que podrían ocurrir simultáneamente con la escritura del bloque 8. Las mismas escrituras en la organización de la izquierda producen modificaciones de los bloques P1 y P2, ambos en el quinto disco, lo cual origina un cuello de botella.

Redundancia P+Q (RAID 6)

Los esquemas basados en paridad protegen contra un único fallo que se autoidentifica. Cuando la corrección de un único fallo no es suficiente, la paridad puede ser generalizada para tener un segundo chequeo de los datos utilizando otro disco con información de control de errores. Este segundo bloque para el control de errores permite recuperarse de un segundo fallo. Por ello, el almacenamiento adicional es dos veces el de RAID 5. El pequeño atajo de escritura que se muestra en la figura 6.13 también funciona, excepto que ahora existen seis accesos a disco en vez de cuatro para actualizar la información tanto de P como de Q.

Resumen de RAID

RAID 1 y RAID 5 se utilizan extensamente en servidores; una estimación establece que el 80% de los discos de los servidores se configuran con algún sistema RAID.

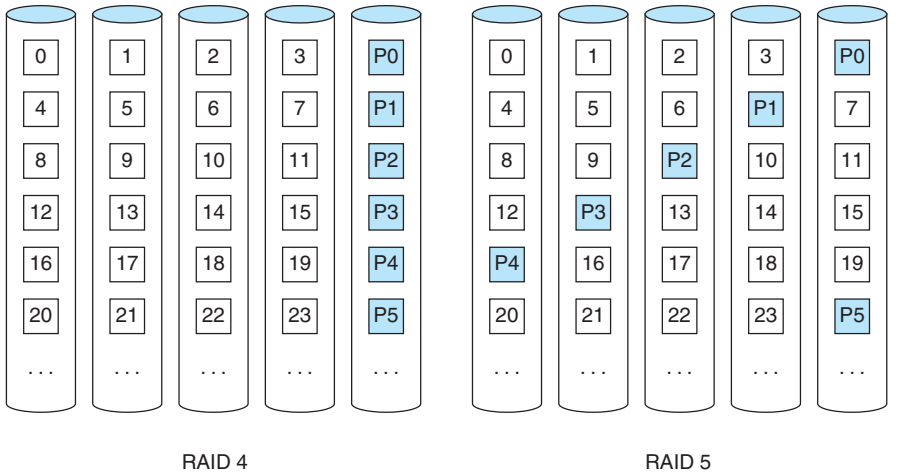


FIGURA 6.14 Paridad de bloques entrelazados (RAID 4) frente a paridad de bloques entrelazados y distribuidos (RAID 5). Distribuyendo los bloques de paridad en todos los discos, algunas escrituras de pequeños bloques de datos pueden ser realizadas en paralelo.

Un punto débil de los sistemas RAID es la reparación. En primer lugar, para evitar la falta de disponibilidad de los datos durante la reparación, la matriz debe ser diseñada para permitir que los discos que fallen sean reemplazados sin tener que apagar el sistema. Los RAIDs disponen de suficiente redundancia para permitir que funcionen continuamente, pero el **reemplazo en caliente** de los discos establece restricciones en el diseño físico y eléctrico de la matriz y de las interfaces de los discos. En segundo lugar, podría producirse otro fallo durante la reparación, por lo que el tiempo de reparación influye en la posibilidad de perder datos: cuanto más aumenta el tiempo de reparación, mayor es la probabilidad de que se produzca otro fallo que ocasionaría la pérdida de datos. En vez de tener que esperar a que el informático vuelva a instalar un disco bueno, algunos sistemas incluyen **piezas de recambio** de forma tal que los datos pueden ser reconstruidos inmediatamente después de descubrir el fallo. El usuario puede entonces reemplazar los discos defectuosos de una forma más relajada. Un informático determina en última instancia qué discos se deben reemplazar. Como se muestra en la figura 6.3, los informáticos son sólo humanos, así que de vez en cuando reemplazan discos que funcionan correctamente en lugar de los defectuosos, originando fallos de disco irrecuperables.

Reemplazo en caliente: reemplazo de un componentes hardware mientras el sistema está encendido.

Piezas de recambio: recursos hardware reservados que pueden ser instalados en lugar de un componente defectuoso.

Además de diseñar el sistema de RAID para facilitar su reparación, hay que tener en cuenta como cambia la tecnología de los discos con el paso del tiempo. Aunque los fabricantes de discos establezcan MTTF muy altos en sus productos, estos números se establecen en condiciones nominales. Si una determinada matriz de discos ha sido sometida a ciclos de temperatura debido, por ejemplo, a fallos en el sistema de aire acondicionado, o a desplazamientos debidos a un mal diseño, construcción o instalación de la carcasa, las frecuencias de fallos pueden ser de 3 a 6 veces mayores (véase la falacia en la página 613). El cálculo de la fiabilidad del RAID supone la independencia de los fallos de los discos, pero los fallos de los discos podrían estar correlacionados entre sí porque tal deterioro debido a las condiciones del entorno probablemente le influye a todos los discos de la matriz. Otro problema es que como el ancho de banda de los discos está aumentando más lentamente que su capacidad, el tiempo de reparación de un disco del RAID está aumentando, lo que a su vez aumenta la probabilidad de un segundo fallo. Por ejemplo, podría tardarse casi tres horas en leer secuencialmente un disco SATA de 1000 GB, suponiendo que no hay interferencias. Como el RAID dañado probablemente seguiría proporcionando datos, la reconstrucción podría alargarse considerablemente. Además, otro problema es que la lectura de muchos datos durante la reconstrucción implica aumentar la probabilidad de un error de lectura no corregible, lo que podría producir una pérdida de datos. Otros argumentos que aumentan la inquietud en lo relativo a fallos múltiples simultáneos son el cada vez mayor número de discos en los RAID y la utilización de discos SATA, que son más lentos y tienen mayor capacidad que los discos tradicionales.

Así, estas tendencias han llevado a un interés creciente en la protección contra más de un fallo y, en consecuencia, el nivel 6 de RAID es una opción cada vez más aceptada en este campo.

Autoevaluación

¿Cuáles de las siguientes frases son ciertas sobre los niveles RAID 1, 3, 4, 5 y 6?

1. Los sistemas RAID dependen de la redundancia para conseguir una alta disponibilidad.

2. RAID 1 (réplica en espejo) tiene asociado la mayor cantidad de discos extra para control de errores.
3. Para las escrituras de pequeños volúmenes de datos, RAID 3 (paridad de bits entrelazados) tiene asociado la peor productividad.
4. Para las escrituras de grandes volúmenes de datos, RAID 3, 4 y 5 tienen asociados las mismas productividades.

Extensión: Una cuestión es la de cómo la técnica de réplica en espejo interacciona con la técnica de desmontado. Suponga que usted tuviera que almacenar, por ejemplo, el equivalente a cuatro discos de datos, y puede usar ocho discos físicos. ¿Crearía cuatro pares de discos (cada uno organizado como un RAID 1) y después distribuiría los datos a lo largo de los cuatro pares RAID 1? Alternativamente, ¿crearía usted dos conjuntos de cuatro discos (cada uno organizado como RAID 0) y entonces realizaría las escrituras a ambos conjuntos RAID 0? La terminología RAID ha evolucionado para nombrar RAID 1+0 o RAID 10 ("espejos desmontados") al primer tipo, y RAID 0+1 o RAID 01 ("desmontado especular").

6.10 Casos reales: servidor Sun Fire x4150

Además de la revolución en la construcción del microprocesador, estamos asistiendo a una revolución en la forma en que se distribuye el software. En lugar del modelo tradicional de un software vendido en un CD o enviado por internet para instalarlo en el computador, la alternativa es el *software como un servicio*. Es decir, se utiliza internet para hacer nuestro trabajo en un computador que ejecuta el software que necesitamos para recibir el servicio que deseamos. Probablemente, el ejemplo más popular es la búsqueda en la web, pero hay más servicios como la edición y almacenamiento de fotos, procesamiento de documentos, almacenamiento en bases de datos, mundos virtuales, etc. Si hiciésemos una búsqueda detallada, muy probablemente encontraríamos una versión de servicio de casi todos los programas que usamos en nuestro computador de sobremesa.

Este cambio ha llevado a la construcción de grandes centros de datos donde se encuentran los computadores y discos para la ejecución de los servicios que utilizan millones de usuarios externos. ¿Cómo deberían ser los computadores diseñados para estos grandes centros de datos? Claramente, no necesitan monitores ni teclados. Por supuesto, si en el centro de datos hay 10 000 computadores, debe primarse la eficiencia en la ocupación de espacio y en el consumo potencia, además de las inquietudes tradicionales respecto al coste y las prestaciones.

Una pregunta relacionada es cómo debería ser el almacenamiento en un centro de datos. Aunque hay muchas opciones, una versión habitual es incluir discos con el procesador y la memoria, formando con esta unidad completa un bloque de construcción. Para evitar problemas de fiabilidad, la aplicación hace copias redundantes y es responsable de mantenerlas coherentes y de la recuperación en caso de fallos.

El sector industrial en Tecnologías de la Información se ha puesto de acuerdo en algunos estándares del diseños físico de los computadores de un centro de datos, específicamente en el armario (*rack*) utilizado para los computadores. El más habitual es el armario de 19 pulgadas (482.6 mm) de ancho. Los computadores diseñados para el

armario son catalogados, naturalmente, como *montura del armario* (*rack mount*), o llamados *subarmario* (*subrack*) o simplemente estante. Puesto que la altura tradicional del hueco para introducir el estante es 1.75 pulgadas (44.45 mm), esta distancia se llama *unidad del armario* (*rack unit*) o sencillamente *unidad* (*U*). El armario más habitual de 19 pulgadas tiene una altura de 42 U, equivalente a 42 x 1.75 o 73.5 pulgadas. La profundidad de los estantes es variable.

FIGURA 6.15 Armario estándar de 19 pulgadas ocupado por 42 servidores 1U. Este armario tiene 42 servidores 1 U "caja de pizza".
Fuente: http://gchepldesk.ualberta.ca/news/07mar06/cbhd_news_07mar06.php

Así, el computador *montura de armario* más pequeño tiene 19 pulgadas de ancho y 1.75 pulgadas de altura, y a menudo se le llama computador 1U o servidor 1U. Por sus dimensiones se han ganado el alias de *caja de pizza*. La figura 6.15 muestra un ejemplo de un armario estándar ocupado por 42 servidores 1U.

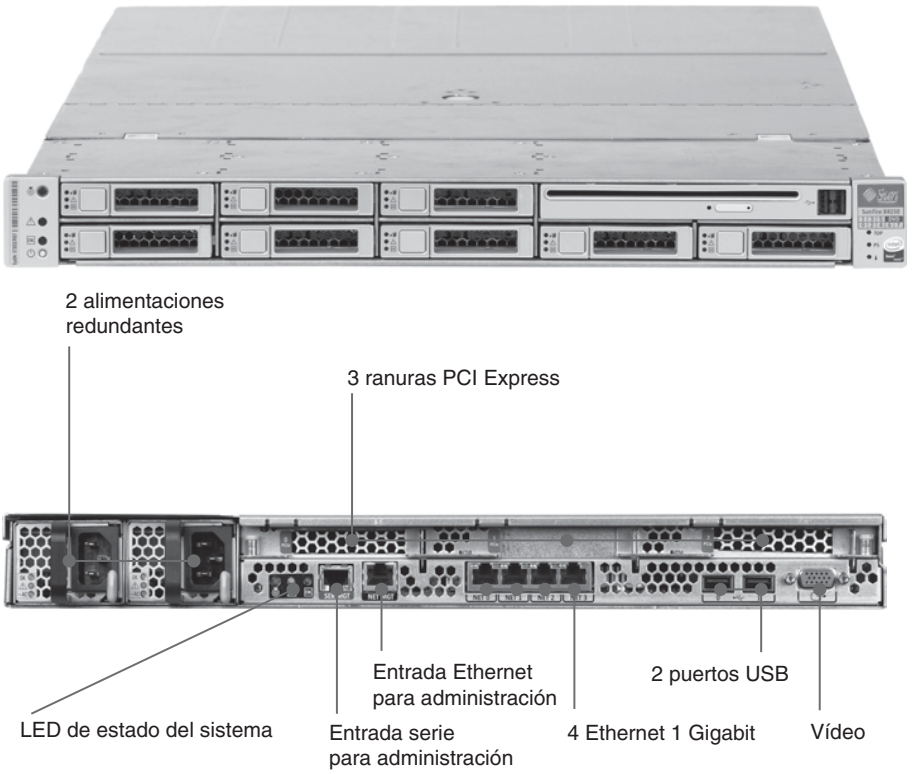


FIGURA 6.16 Partes frontal y trasera del servidor 1U Sun Fire x4150. Sus dimensiones son 1.75 pulgadas de alto y 19 pulgadas de ancho. Los 8 discos de 2.5 pulgadas pueden ser reemplazados desde la parte frontal. En la parte superior derecha tiene un DVD y dos puertos USB. En la foto se han puesto etiquetas a los componentes de la parte trasera. Tiene alimentación y ventilación redundantes que permiten que el servidor siga funcionando aunque falle alguno de estos componentes.

La figura 6.16 muestra el Sun Fire x4150, un ejemplo de servidor 1U. En su configuración máxima este estante 1U contiene:

- 8 procesadores a 2.66 GHz, distribuidos en dos conectores (*sockets*) (2 Intel Xeon 5345).
- 64 GB de memoria DRAM DDR2-667, distribuidos en 16 FBDIMMs de 4 GB.
- 8 discos de 2.5 pulgadas SAS con 73 GB y 15 000 RPM.
- 1 controlador de RAID (que soporta RAID 0, RAID 1, RAID 5 y RAID 6).
- 4 puertos Ethernet 10/100/1000.
- 3 puertos PCI Express x8.
- 4 puertos USB 2.0 externos y 1 interno.

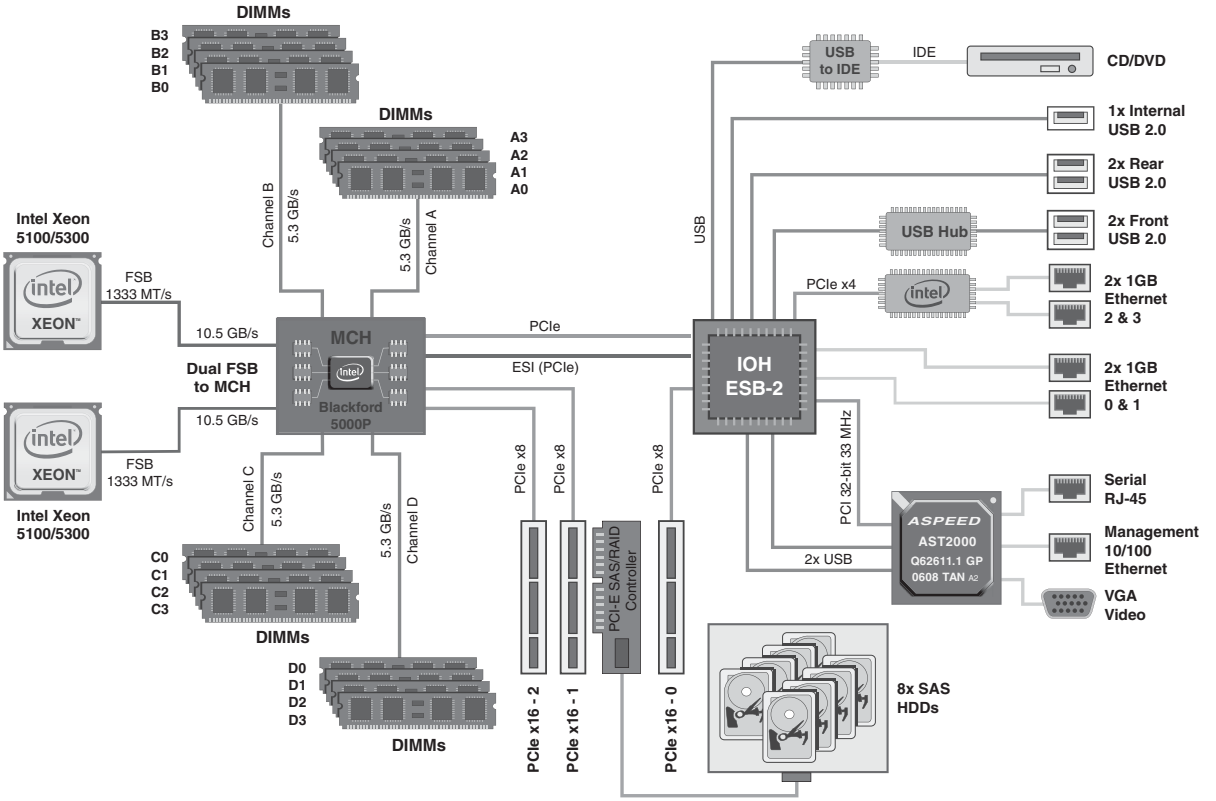


FIGURA 6.17 Conexiones lógicas y ancho de banda de los componentes del Sun Fire x4150. Los tres conectores PCIe permiten que se enchufen tarjetas x16, pero solo proporciona 8 vías de ancho de banda al MCH. Fuente: figura 5 de "SUN FIRETM 4150 AND X4450. SERVER ARCHITECTURE" (véase www.sun.com/servers/x64/x4150/).

La figura 6.17 muestra las conexiones y el ancho de banda de los chips de la placa base. En las figuras 6.9 y 6.10 se describen los discos SAS del Sun Fire x4150.

Para aclarar los consejos sobre el diseño de un sistema de E/S de la sección 6.8, vamos a hacer una evaluación simple de las prestaciones para determinar donde puede estar el cuello de botella de una aplicación hipotética.

EJEMPLO

Diseño del sistema de E/S

Consideremos las siguientes suposiciones sobre el Sun Fire x4150:

- El programa de usuario emplea 200 000 instrucciones en cada operación de E/S.
- El sistema operativo emplea de media 100 000 instrucciones en cada operación de E/S.

- La carga de trabajo consiste en lecturas de 64 KB.
- Cada procesador puede ejecutar (ejecución sostenida) 1000 millones de instrucciones por segundo.

¿Cuál es el ritmo de E/S sostenido máximo de un Sun Fire x4150 completamente cargado en lecturas aleatorias y lecturas secuenciales? Supongamos que las lecturas siempre se pueden hacer en un disco inactivo, en caso de existir (es decir, ignorar los conflictos de disco), y que el controlador del RAID no es el cuello de botella.

RESPUESTA

En primer lugar, vamos a determinar el ritmo de E/S de un único procesador. Cada E/S emplea 200 000 instrucciones de usuario y 100 000 instrucciones del sistema operativo, entonces

Ritmo máximo de E/S de 1 procesador =

$$\frac{\text{Ritmo de ejecución de instrucciones}}{\text{instrucciones por E/S}} = \frac{1 \times 10^9}{200 + 100 \times 10^3} = 3.333 \frac{\text{E/S}}{\text{por segundo}}$$

Como cada conector Intel 5345 tiene cuatro procesadores, puede llevar a cabo 13 333 IOPS. Dos conectores con ocho procesadores pueden hacer 26 667 IOPS.

Determinemos ahora el IOPS de los discos SAS de 2.5 pulgadas descritos en la figura 6.5, para lecturas aleatorias y secuenciales. En lugar de utilizar el tiempo de búsqueda medio proporcionado por el fabricante, supondremos la cuarta parte de este tiempo, como ocurre habitualmente (véase sección 6.3). El tiempo de cada lectura aleatoria de un disco es:

Tiempo por E/S a disco = búsqueda + retardo rotacional + tiempo de transferencia

$$= \frac{2.9}{4}\text{ms} + 2.0\text{ ms} \frac{64\text{ KB}}{112\text{ MB/seg}} = 3.3\text{ms}$$

Así, cada disco puede completar 1000 ms /3.3 ms o 303 E/S por segundo y, por lo tanto ocho discos llevan a cabo 2424 lecturas aleatorias por segundo.

Para determinar el número de lecturas secuenciales, tenemos que dividir el ancho de banda entre el tamaño de la transferencia:

$$\frac{112\text{ MB/seg}}{64\text{ KB}} = 1750\text{ IOPS}$$

Ocho discos pueden hacer 14 000 lecturas secuenciales de 64 KB.

Queremos determinar si los caminos entre el disco y la memoria y el procesador constituyen el cuello de botella. Comenzaremos con la conexión PCI Express entre la tarjeta RAID y el chip del puente norte. Como cada vía de un PCIe tiene un ancho de banda 250 MB/seg, entonces ocho discos tienen un ancho de banda de 2 GB/seg.

$$\text{Frecuencia de E/S máxima} = \frac{\text{ancho de banda del PCI}}{\text{byte por E/S}} = \frac{2 \times 10^9}{64 \times 10^3} =$$

$$31\,250 \frac{\text{E/S}}{\text{por segundo}}$$

Ocho discos haciendo transferencias secuenciales utilizan menos de la mitad de la conectividad proporcionada por un PCIe x8.

Una vez que los datos están en el MCB, hay que escribirlos en la DRAM. El ancho de banda de una FBDIMM DDR2 a 667 MHz es 5336 MB/seg. Un único DIMM puede hacer

$$\frac{5336 \text{ MB/seg}}{64 \text{ MB}} = 83\,375 \text{ IOPS}$$

La memoria no es un cuello de botella, incluso cuando hay un único DIMM, y en un Sun Fire x4150 completamente configurado disponemos de 16.

La última conexión en la cadena es el bus del sistema que conecta el puente norte con el conector Intel 5345. Su ancho de banda pico es 10.6 GB/seg, pero la en la sección 7.10 se indica que no debe utilizarse más de la mitad de este ancho de banda pico. Cada operación de E/S transfiere 64 KB, entonces

$$\text{Frecuencia de E/S máxima del bus del sistema} = \frac{\text{ancho de banda del bus}}{\text{byte por E/S}} = \frac{5.3 \times 10^9}{64 \times 10^3} = 81\,540 \ \frac{\text{E/S}}{\text{por segundo}}$$

Hay un bus del sistema por conector, entonces el pico del bus del sistema dual es 150 000 IOPS, y una vez más, no es el cuello de botella.

De este modo, un Sun Fire x4150 totalmente configurado puede mantener el ancho de banda pico de ocho discos, 2424 lecturas aleatorias por segundo o 14 000 lecturas secuenciales por segundo.

Observe el elevado número de suposiciones simplificadoras que han sido necesarias para hacer este ejemplo. En la práctica, muchas de estas simplificaciones podrían no ser aplicables en aplicaciones críticas con E/S intensiva. Por esta razón, a menudo la única forma convincente de evaluar las prestaciones de un sistema de E/S es ejecutar una carga de trabajo realista o un programa de prueba relevante.

Como se ha mencionado al principio de la sección, estos nuevos centros de datos tienen que ser cuidadosos con el espacio y la potencia, y también con coste y las prestaciones. La figura 6.18 muestra la potencia pico y en estado inactivo de un Sun Fire 4150 totalmente configurado, detallado para cada componente. Analicemos ahora configuraciones alternativas del Sun Fire x4150 para ahorrar potencia.

Evaluación de la potencia de un sistema de E/S

EJEMPLO

Reconfiguremos el Sun Fire x4150 para minimizar la potencia, suponiendo que la única actividad en este servidor 1 U es la carga de trabajo del ejemplo anterior.

RESPUESTA

Para alcanzar las 2424 lecturas aleatorias por segundo de bloques de 64 KB del ejemplo anterior necesitamos los ocho discos y el controlador PCI RAID. De los cálculos anteriores, un único DIMM puede hacer 80 000 IOPS, por lo tanto podemos ahorrar potencia en la memoria. La memoria mínima del Sun Fire x4150 es dos DIMMs, así que podemos ahorra la potencia (y el coste) de 14 DIMM de 4 GB. Un conector puede soportar 13 333 IOPS, por lo tanto podemos reducir en uno el número de conectores Intel E5345. Utilizando los valores de la figura 6.18, la potencia total del sistema ahora es:

	Componentes			Sistema			
Elemento	Inactivo	Pico	Número	Inactivo		Pico	
Conexión Intel 2.66 GHz E5345, chips Intel 5000 MCB/IOH, controladores Ethernet, alimentación, ventiladores, ...	154 W	215 W	1	154 W	37%	215 W	39%
Conexión Intel 2.66 GHz E5345 adicional	22 W	79 W	1	22 W	5%	79 W	14%
FBDIMM DDR2-667 5300 de 4 GB	10 W	11 W	16	160 W	39%	176 W	32%
Disco SAS 15k de 73 GB	8 W	8 W	8	64 W	15%	64 W	12%
Controlador de RAID PCIe x8	15 W	1	1	15 W	4%	15 W	3%
Total	—	—	—	415 W	100%	549 W	100%

FIGURA 6.18 Potencia pico y en estado inactivo de un Sun Fire 4150 totalmente configurado. Datos obtenidos al ejecutar SPECJBB con 29 configuraciones diferentes, por lo tanto la potencia pico podría ser diferente al ejecutar otras aplicaciones (fuente: www.sun.com/servers/x64/x4150/calc).

$$\text{Potencia inactivo}_{\text{lecturas aleatorias}} = 154 + 2 \times 10 + 8 \times 8 + 15 = 253 \text{ W}$$

$$\text{Potencia pico}_{\text{lecturas aleatorias}} = 215 + 2 \times 11 + 8 \times 8 + 15 = 316 \text{ W}$$

lo que supone reducir la potencia en un factor 1.6 o 1.7.

El sistema original puede realizar 14 000 lecturas secuenciales de bloques de 64 KB por segundo. Para ello se necesitan todos los discos y los controladores de disco, y el mismo número de DIMMs pueden manejar esta carga. Esta carga de trabajo excede la potencia computacional de un conector Intel E5345, por lo que es necesario añadir un segundo conector.

$$\text{Potencia inactivo}_{\text{lecturas secuenciales}} = 154 + 22 + 2 \times 10 + 8 \times 8 + 15 = 275 \text{ W}$$

$$\text{Potencia pico}_{\text{lecturas secuenciles}} = 215 + 79 + 2 \times 11 + 8 \times 8 + 15 = 395 \text{ W}$$

lo que supone reducir la potencia en un factor 1.4 o 1.5.

Aspectos avanzados: redes

Las redes son cada vez más populares, y a diferencia de otros dispositivos de E/S, existen muchos libros y cursos sobre ellas. Para los lectores que no han recibido cursos o leído libros sobre redes de computadores, la **sección 6.11** en el CD proporciona una visión de conjunto de los principales temas y la terminología, incluyendo las redes de interconexión, el modelo OSI, las familias de protocolos como TCP/IP, las redes de larga distancia como ATM, las redes de área local como Ethernet y las redes inalámbricas como IEEE 802.11.

6.12 Falacias y errores habituales

Falacia: el tiempo medio de fallo de los discos (MTTF) es de 1 200 000 horas o casi 140 años, así que los discos prácticamente nunca fallan.

Las prácticas propagandísticas de los fabricantes de discos pueden llevar a conclusiones erróneas a los usuarios. ¿Cómo se calcula el MTTF? En una fase temprana del proceso los fabricantes ponen miles de discos en una habitación y los hacen funcionar durante meses, contando el número de ellos que fallan. El cómputo de MTTF es el número total de horas que los discos están funcionando dividido entre el número de discos que fallan.

El problema es que este número excede con creces el tiempo de vida del disco, que habitualmente se acepta que es de cinco años o 43 800 horas. Para que este valor grande de MTTF tenga algún sentido, los fabricantes de discos argumentan que el cálculo corresponde a un usuario que compra un disco, y entonces reemplaza el disco cada cinco años: el tiempo de vida programado para el disco. La afirmación es que si muchos clientes (y sus tataranietos) hicieran esto durante la siguiente centuria, de media podrían reemplazar el disco 27 veces antes de que se produjera un fallo, o alrededor de 140 años.

Una medida más útil sería el porcentaje de discos que fallan en un año, llamado frecuencia de fallos anual (*annual failure rate, AFR*). Suponga que tenemos 1000 discos con un MTTF de 1 200 000 horas y que los discos se usan 24 horas al día. Si se reemplazan los discos que fallan por uno nuevo con las mismas características de fiabilidad, el número de los discos que fallarían durante cinco años (8 760 horas) sería:

$$\text{Discos que fallan} = \frac{1000 \text{ discos} \times 8760 \text{ horas/disco}}{1\,200\,000 \text{ horas/fallo}} = 7.3$$

Dicho de otra forma, el AFR es 0.73%. Los fabricantes de discos están comenzando a indicar el AFR y el MTTB para que los usuarios conozcan mejor lo que pueden esperar de sus productos.

Falacia: La frecuencia de fallos de un disco en operación es igual a la indicada en sus especificaciones.

Dos estudios recientes han evaluado una gran cantidad de discos para comparar los resultados en operación y sus especificaciones. Uno de los estudios analizó casi 100 000 discos ATA y SCSI con unas especificaciones de MTTF entre 1 000 000 y 1 500 000 horas o AFR entre 0.6% y 0.8%. Se encontró que el AFR estaba entre el 2% y el 4%, a menudo entre tres y cinco veces más elevado que lo indicado en las especificaciones [Schroeder and Gibson, 2007]. El segundo estudio, de más de 100 000 discos ATA con un AFR de aproximadamente 1.5% en las especificaciones, encontró que frecuencias de fallos de 1.7% en el primer año de operación crecían hasta el 8.6% en el tercer año, es decir, entre cinco y seis veces lo especificado [Pinheiro, Weber and Barroso, 2007].

Falacia: un bus de 1 GB/seg puede transferir 1 GB de datos en 1 segundo.

En primer lugar, es generalmente imposible utilizar el 100% de cualquiera de los recursos del computador. En el caso del bus, se es afortunado si se consigue entre un 70% y un 80% del ancho de banda pico. Las principales razones de no poder alcanzar un uso del 100% son el tiempo para enviar la dirección, el tiempo para reconocer las señales, y las paradas que se producen mientras se espera a poder usar un bus ocupado.

En segundo lugar, la definición de megabyte de almacenamiento no coincide con la de megabyte por segundo de ancho de banda. Tal como se ha expuesto en la página 596, las medidas de ancho de banda de E/S se dan habitualmente en base 10 (1 GB/seg = 10^9 bytes/seg), mientras que 1 GB de datos normalmente es una medida en base 2 (1 MB = 2^{30} bytes). ¿Cuán importante es esta distinción? Si se pudiera usar el 100% del bus en una transferencia de datos, el tiempo para transferir 1 GB de datos en un bus de 1 GB/seg es en realidad:

$$\frac{2^{30}}{10^9} = \frac{1\,073\,741\,824}{1\,000\,000\,000} = 1\,073\,741\,824 \approx 1.07 \text{ segundos}$$

Error habitual: tratar de mejorar sólo las características internas de la red y no considerar la conexión origen-final (end-to-end).

El problema es proporcionar a bajo nivel características que sólo pueden lograrse al más alto nivel, satisfaciendo así los requisitos de comunicación sólo de un modo parcial. Saltzer, Reed y Clark [1984] explican el *argumento origen-final* como:

> *La función en cuestión sólo puede ser especificada de forma correcta y completa con el conocimiento y la ayuda de la aplicación que se encuentra en los puntos finales del sistema de comunicación. Por tanto, no es posible proporcionar esta función como una característica intrínseca del sistema de comunicación.*

Su ejemplo de dificultad fue una red en el MIT que usaba varias puertas de enlace (*gateways*), cada una de las cuales añadía una suma de verificación (*checksum*) entre una puerta y la siguiente. Los programadores de la aplicación suponían que esta suma de verificación garantizaba la precisión, creyendo de forma incorrecta que el mensaje estaba protegido mientras estaba almacenado en la memoria de cada puerta de enlace. Una de estas puertas de enlace sufría un error transitorio que consistía en intercambiar una pareja de bytes por cada millón de bytes transferidos. A lo largo del tiempo, el código fuente de un sistema operativo había sido enviado a través de la puerta repetidas veces, y se había corrompido. La única solución fue corregir los ficheros fuente infectados comparando los listados en papel y ¡reparando el código a mano! Si las sumas de verificación hubieran sido calculadas y comprobadas por la aplicación que se ejecutaba en los sistemas finales, entonces sí que se podría haber garantizado la seguridad.

Las comprobaciones intermedias tienen su papel siempre y cuando se realice una comprobación en la comunicación entre el origen y el destino final. Esta verificación origen-final puede descubrir que algo se ha roto entre dos nodos, pero no apunta al lugar donde se encuentra el problema. Las verificaciones intermedias pueden descubrir qué elemento es el que se ha roto. Se necesitan ambas para repararlo.

Error habitual: trasladar funciones de la CPU al procesador de E/S y, sin un análisis cuidadoso, esperar una mejora de las prestaciones.

Hay muchos ejemplos de este problema, aunque, los procesadores de E/S usados apropiadamente, sin duda mejoran las prestaciones. Un ejemplo frecuente es el uso de interfaces de E/S inteligentes, que, debido a una mayor sobrecarga para iniciar una operación de E/S, pueden resultar en una mayor latencia que una actividad de E/S dirigida por el procesador (aunque la productividad del sistema puede llegar a crecer si el procesador se libera el tiempo suficiente). Con frecuencia las prestaciones caen cuando el procesador de E/S tiene un rendimiento muy inferior al del procesador principal. Como consecuencia, una pequeña cantidad de tiempo del procesador principal se sustituye por una mayor cantidad de tiempo del procesador de E/S. Los diseñadores de estaciones de trabajo han vivido estas situaciones repetidamente.

Myer y Sutherland [1968] escribieron un artículo clásico sobre el compromiso entre la complejidad y las prestaciones en controladores de E/S. Tomando prestado el concepto religioso de la "rueda de la reencarnación", ellos acabaron dándose cuenta de que habían sido capturados en un lazo en el que se incrementaba continuamente la potencia del procesador de E/S hasta que llegaba a necesitar su propio coprocesador simple.

Enfocamos la tarea comenzando con un esquema simple y añadimos comandos y características que creíamos que mejorarían la potencia de la máquina. Gradualmente el procesador [de pantalla] se hacía más complejo … Finalmente el procesador de pantalla acabó semejándose a un computador completo con algunas características especiales para gráficos. Y entonces ocurrió una extraña cosa. Nos sentimos forzados a añadir al procesador un segundo procesador subsidiario, que, él mismo también comenzó a crecer en complejidad. Fue entonces cuando descubrimos la inquietante verdad. Diseñar un procesador de pantalla puede convertirse en un proceso cíclico sin fin. De hecho, encontramos el proceso tan frustrante que acabamos llamándolo la "rueda de la reencarnación".

Error habitual: usar cintas magnéticas como copia de seguridad para los discos.

De nuevo se trata tanto de una falacia como de un error habitual. Las cintas magnéticas han formado parte de los sistemas de computación desde hace tanto tiempo como los discos y usan una tecnología similar a la de los discos, y por tanto han mantenido históricamente la misma mejora en densidad. La diferencia histórica en cuanto a coste y prestaciones entre los discos y las cintas se basa en que un disco rotatorio sellado tiene menor tiempo de acceso que una cinta secuencial, pero las bobinas de cinta magnética son removibles y se pueden usar muchas cintas por lector, que pueden ser muy grandes, y por tanto tener mucha capacidad. Por tanto, en el pasado una única cinta magnética podía mantener el contenido de muchos discos, y como era entre 10 y 100 veces más barata por gigabyte que los discos, era un medio útil para guardar copias de seguridad.

La afirmación era que las cintas magnéticas deberían seguir a las prestaciones de los discos ya que las innovaciones en los discos deberían ayudar a mejorar las cintas. Esta afirmación era importante porque las cintas representaban un mercado reducido y no se podían permitir un gran esfuerzo en investigación y desarrollo. Una razón de que el mercado fuera pequeño es que los propietarios de computadores de sobremesa generalmente no hacen copias de seguridad en cinta, y como estos computadores eran con mucho el mayor mercado para los discos, representaban un pequeño mercado para las cintas.

Así, el mayor mercado ha conducido a que los discos mejoren mucho más rápido que las cintas. Desde 2000 hasta 2002, el disco más popular tenía una mayor capacidad que la cinta más popular. En el mismo periodo de tiempo el precio por gigabyte de los discos ATA bajó por debajo del de las cintas. Los defensores de las cintas afirmaban entonces que las cintas tenían unos requerimientos de compatibilidad que no se imponían a los discos; los lectores de cintas debían ser capaces de leer o escribir en las cuatro generaciones previas de cintas. Como los discos eran sistemas cerrados, las cabezas de los discos solo necesitaban leer los platos que estaban dentro de ellos, y esta ventaja explica la razón de que los discos mejoraban mucho más rápidamente.

Hoy en día, algunas organizaciones han descartado completamente las cintas, y usan redes y discos remotos para replicar los datos geográficamente. Los lugares se escogen para evitar que tenga lugar algún desastre en ambos sitios, posibilitando una recuperación instantánea. (El gran tiempo de recuperación es otro serio inconveniente de la naturaleza secuencial de las cintas). Esta solución depende de los avances en la capacidad de los discos y del ancho de banda de las redes para que económicamente tenga sentido, pero ambos elementos están sometidos a una gran inversión y por tanto han tenido recientemente mejores resultados que las cintas.

Falacia: El mejor lugar para planificar los accesos a disco es el sistema operativo.

Como se ha mencionado en la sección 6.3, las interfaces de alto nivel como ATA o SCSI proporcionan direcciones de bloques lógicos al sistema operativo. Dado este alto nivel de abstracción, lo mejor que puede hacer el sistema operativo para intentar contribuir a aumentar las prestaciones es ordenar las direcciones de los bloques lógicos en orden creciente. Sin embargo, como el disco conoce la proyección real de direcciones lógicas en la geometría física de sectores, pistas y superficies, puede reducir las latencias de búsqueda y rotacional mediante la replanificación.

Por ejemplo, supongamos una carga de trabajo de cuatro lecturas [Anderson, 2003]:

Operación	Dirección de comienzo del bloque lógico	Longitud
Lectura	724	8
Lectura	100	16
Lectura	9987	1
Lectura	26	128

El sistema operativo puede reordenar las cuatro lecturas según la dirección del bloque lógico

Operación	Dirección de comienzo del bloque lógico	Longitud
Lectura	26	128
Lectura	100	16
Lectura	724	8
Lectura	9987	1

Dependiendo de la posición relativa del dato en el disco, este reordenamiento puede empeorar el tiempo de acceso, como se muestra en la figura 6.19. Las lecturas planificadas por el disco terminan en tres cuartos de revolución del disco, mientras que las planificadas por el sistema operativo tardan tres revoluciones.

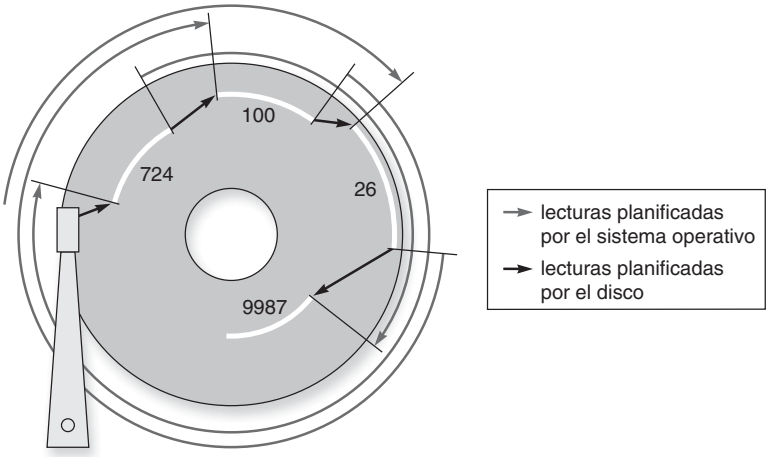


FIGURA 6.19 Ejemplo que muestra la planificación de los accesos por el Sistema operativo frente a la planificación hecha por el disco. La planificación del disco necesita tres revoluciones para terminar las cuatro lecturas, y la con planificación hecha por disco, las cuatro lecturas terminan en tres cuartos de revolución (extraído de [Anderson, 2003]).

Error habitual: usar la velocidad de pico de una parte del sistema de E/S para hacer previsiones y comparaciones de prestaciones.

Muchos de los componentes del sistema de E/S (dispositivos, controladores, buses) se especifican usando sus anchos de banda de pico. En la práctica estos anchos de banda de pico se basan en suposiciones poco realistas sobre el sistema, o son inalcanzables debido a otras limitaciones del mismo. Por ejemplo, para referirse al rendimiento del bus se usa con frecuencia la velocidad de transferencia de pico usando un sistema de memoria que es imposible construir. En sistemas conectados en red, el tiempo de sobrecarga del software para iniciar la comunicación es generalmente ignorado.

El bus PCI de 32 bits y 33 MHz tiene un ancho de banda de pico de unos 133 MB/seg. En la práctica, incluso para transferencias de gran tamaño, es difícil mantener más de 80 MB/seg para sistemas de memoria realistas.

La ley de Amdahl también nos recuerda que la productividad de un sistema de E/S estará limitada por el componente de menor rendimiento en el camino de E/S.

6.13 Conclusiones finales

Los sistemas de E/S se evalúan según diferentes criterios: fiabilidad, variedad de dispositivos de E/S que pueden soportar, máximo número de dispositivos de E/S, coste y prestaciones, medido tanto en términos de latencia como de productividad. Estos objetivos llevan a una amplia variedad de esquemas para realizar la interfaz con los dispositivos de E/S. En sistemas de gama baja o media, el uso de un DMA con búfer es posiblemente el mecanismo de transferencia dominante. En sistemas de gama alta, la latencia y el

ancho de banda pueden ser ambos importantes, y el coste puede ser secundario. Los sistemas de E/S en este extremo superior de la escala se caracterizan frecuentemente por tener muchos caminos a dispositivos de E/S, con un nivel limitado de almacenamiento en búferes. Habitualmente, la capacidad de acceder en cualquier momento a los datos de un dispositivo de E/S (alta disponibilidad) se hace más importante a medida que el sistema crece. Como resultado, los mecanismos de redundancia y control de errores son más y más predominantes a medida que se aumenta el tamaño del sistema.

Las peticiones de almacenamiento y conectividad están creciendo a velocidades sin precedentes, en parte debido a que la demanda de poseer toda la información al alcance de la mano es cada vez mayor. Existen estimaciones de que la cantidad de información creada en 2002 fue de 5 exabytes (equivalente a 500 000 copias de los textos de la biblioteca del congreso de EE.UU), y que la cantidad total de información en el mundo se ha doblado en los últimos tres años [Lyman y Varian, 2003].

Las direcciones futuras de la E/S incluyen extender el alcance de las redes alámbricas e inalámbricas, de modo que prácticamente cada dispositivo tenga su propia dirección IP, e incrementando la participación de la memoria flash en los sistemas de almacenamiento.

Comprender las prestaciones de los programas

Las prestaciones de un sistema de E/S, tanto si se mide en términos de ancho de banda como de latencia, dependen de todos los elementos en el camino entre el dispositivo y la memoria, incluido el sistema operativo que genera las órdenes de E/S. El ancho de banda de los buses, de la memoria y del dispositivo determinan la máxima velocidad de transferencia desde o hacia el dispositivo. De forma similar, la latencia depende de la latencia del dispositivo, así como de cualquier otra latencia que puedan imponer el sistema de memoria o los buses. El ancho de banda efectivo y la latencia de respuesta también dependen de otras peticiones de E/S que pueden competir por los recursos en el camino de la E/S. Finalmente, el sistema operativo es un cuello de botella. En algunos casos, el sistema operativo emplea mucho tiempo en enviar al dispositivo una petición de E/S originada por un programa de usuario, dando lugar a una latencia elevada. En otros casos, el sistema operativo verdaderamente limita el ancho de banda de E/S debido a limitaciones en el número de operaciones de E/S que puede soportar de forma concurrente.

Se debe mantener en mente que mientras que las prestaciones puede ayudar a vender un sistema de E/S, lo que los usuarios piden abrumadoramente a sus sistemas de E/S es fiabilidad y capacidad.

6.14 Perspectiva histórica y lecturas recomendadas

La historia de los sistemas de E/S es fascinante. La sección 6.14 proporciona una breve historia de los discos magnéticos, RAID, bases de datos, *Internet*, *World Wide Web*, y de cómo *Ethernet* continúa triunfando sobre sus contendientes.

6.15 Ejercicios

Contribución de Perry Alexander, Universidad de Kansas

Ejercicio 6.1

La figura 6.2 describe varios dispositivos de E/S, su comportamiento, interlocutor y ritmo de transferencia de datos. Sin embargo, estas clasificaciones a menudo no dan una visión completa del flujo de datos dentro del sistema. En este ejercicio se exploran diversas clasificaciones para los siguientes dispositivos.

a.	Videojuegos
b.	GPS manual

6.1.1 [5]<6.1> Para los dispositivos de la tabla, identifique la interfaz de E/S y clasifíquelos en términos de comportamiento e interlocutor.

6.1.2 [5]<5.1> Para las interfaces identificadas en el problema anterior, haga una estimación de su ritmo de transferencia de datos.

6.1.3 [5]<5.1> Para las interfaces identificadas en el problema anterior, determine cuál es la mejor medida de las prestaciones: el ritmo de transferencia de datos o el ritmo de operaciones.

Ejercicio 6.2

El tiempo medio entre fallos (MTBF), el tiempo medio hasta que se hace la reparación (MTTR) y el tiempo medio hasta que se produce un fallo (MTTF), son métricas útiles para evaluar la fiabilidad y disponibilidad de un sistema de almacenamiento. Explorare estos conceptos contestando a preguntas sobre dispositivos con las siguientes métricas.

	MTTF	MTTR
a.	5 años	1 semana
b.	10 años	5 días

6.2.1 [5]<6.1, 6.2> Calcule el MTBF para los dispositivos de la tabla.

6.2.2 [5]<6.1, 6.2> Calcule la disponibilidad de los dispositivos de la tabla.

6.2.3 [5]<6.1, 6.2> ¿Qué ocurre con la disponibilidad a medida que el MTTR se aproxima a 0? ¿Es ésta una situación realista?

6.2.4 [5]<6.1, 6.2> ¿Qué ocurre con la disponibilidad si el MTTR se hace muy alto, es decir, un dispositivo que es difícil de reparar? ¿Implica esto que el dispositivo tiene poca disponibilidad?

Ejercicio 6.3

Los tiempos mínimo y medio de lectura y escritura en un dispositivo de almacenamiento se utilizan frecuentemente para comparar dispositivos. Utilizando las técnicas del capítulo 6, calcule valores relacionados con el tiempo de lectura y escritura en discos con las siguientes características.

	Tiempo medio de búsqueda	RPM	Ritmo de transferencia del disco	Ritmo de transferencia del controlador
a.	11 ms	7200	34 Mbytes/s	480Mbits/s
b.	9 ms	7200	30 Mbytes/s	500 Mbits/s

6.3.1 [10]<6.2, 6.3> Calcule el tiempo medio necesario para leer o escribir un sector de 1024 bytes en cada disco de la tabla.

6.3.2 [10]<6.2, 6.3> Calcule el tiempo medio necesario para leer o escribir un sector de 2048 bytes en cada disco de la tabla.

6.3.3 [10]<6.2, 6.3> Determine el factor determinante para las prestaciones en cada disco. Específicamente, si se quisiera mejorar cualquier aspecto del disco, ¿cuál se debería escoger? Si no hay un factor dominante, explique la razón.

Ejercicio 6.4

En última instancia, el diseño de sistemas de almacenamiento exige tener en cuenta los escenarios de utilización y los parámetros del disco, porque situaciones diferentes requieren métricas diferentes. Respondiendo a las siguientes cuestiones sobre las aplicaciones de la tabla, investigue las diferentes formas de evaluación de los sistemas de almacenamiento.

a.	Base de datos en línea de satélites de la NASA
b.	Sistema de videojuegos

6.4.1 [5]<6.2, 6.3> Para cada aplicación, ¿mejorarían las prestaciones si se disminuye el tamaño del sector durante las lecturas y escrituras? Razone la respuesta.

6.4.2 [5]<6.2, 6.3> Para cada aplicación, ¿mejorarían las prestaciones si se aumenta la velocidad de rotación del disco? Razone la respuesta.

6.4.3 [5]<6.2, 6.3> Para cada aplicación, ¿mejorarían las prestaciones si se aumenta la velocidad de rotación del disco y disminuye el MTTF? Razone la respuesta.

Ejercicio 6.5

La memoria Flash es uno de primeros competidores serios para los discos magnéticos tradicionales. Respondiendo a las siguientes cuestiones sobre las aplicaciones de la tabla, investigue las implicaciones de las memorias Flash.

a.	Base de datos en línea de satélites de la NASA
b.	Sistema de videojuegos

6.5.1 [5]<6.2, 6.3, 6.4> A medida que nos movemos hacia discos de estado sólido construidos a partir de memorias Flash, ¿qué cambiará en los tiempos de lectura del disco suponiendo que el ritmo de transferencia de datos permanece constante?

6.5.2 [10]<6.2, 6.3, 6.4> Suponiendo que el coste es un factor importante en el diseño, ¿podrían las aplicaciones de la tabla beneficiarse de tener discos Flash de estado sólido?

6.5.3 [10]<6.2, 6.3, 6.4> Suponiendo que el coste NO es un factor importante en el diseño, ¿podrían las aplicaciones de la tabla no ser apropiadas para discos Flash de estado sólido?

Ejercicio 6.6

Respondiendo a las siguientes cuestiones sobre las prestaciones de memorias Flash con las características de la tabla, investigue la naturaleza de este tipo de memoria.

	Ritmo de transferencia de datos	Ritmo de transferencia del controlador
a.	34 MB/s	480 MB/s
b.	30 MB/s	500 MB/s

6.6.1 [10]<6.2, 6.3, 6.4> Calcule el tiempo medio de lectura o escritura de un sector de 1024 bytes en una memoria Flash con las características de la tabla.

6.6.2 [10]<6.2, 6.3, 6.4> Calcule el tiempo medio de lectura o escritura de un sector de 512 bytes en una memoria Flash con las características de la tabla.

6.6.3 [5]<6.2, 6.3, 6.4> La figura 6.6 muestra que el tiempo de acceso de lectura y escritura de una memoria Flash aumenta a medida que aumenta la capacidad de la memoria. ¿Esto es algo inesperado? ¿Qué factores intervienen en este comportamiento?

Ejercicio 6.7

La E/S puede hacerse de forma síncrona o asíncrona. Explore las diferencias entre ambas alternativas respondiendo a las siguientes preguntas sobre las prestaciones de los periféricos de la tabla.

a.	Ratón
b.	Controlador de memoria

6.7.1 [5]<6.5> ¿Qué tipo de bus (síncrono o asíncrono) sería el más apropiado para la comunicación entre la CPU y el periférico de la tabla?

6.7.2 [5]<6.5> ¿Qué problemas podría tener un bus síncrono largo en la comunicación entre la CPU y el periférico de la tabla?

6.7.3 [5]<6.5> ¿Qué problemas podría tener un bus asíncrono en la comunicación entre la CPU y el periférico de la tabla?

Ejercicio 6.8

Entre los buses que actualmente se utilizan con mayor frecuencia se encuentran los siguientes: FireWire (IEEE 394), USB, PCI y SATA. Aunque los cuatro son buses asíncronos, tienen diferentes implementaciones, y por lo tanto diferentes características. Respondiendo a las siguientes preguntas sobre los buses y los periféricos de la tabla, investigue las diferencias entre los buses.

a.	Disco duro externo
b.	Teclado

6.8.1 [5]<6.5> Seleccione el bus apropiado (FireWire, USB, PCI o SATA) para los periféricos de la tabla. Razonar la respuesta. (Las características de cada bus se pueden encontrar en la figura 6.8.)

6.8.2 [20]<6.5> Utilice recursos en línea o de la biblioteca para encontrar la estructura de cada tipo de bus y resuma sus características. Indique qué hace el controlador del bus y dónde está físicamente el control.

6.8.3 [15]<6.5> Indique las limitaciones de cada tipo de bus. Explique por qué estas limitaciones deben tenerse en cuenta al utilizar el bus.

Ejercicio 6.9

La comunicación con los dispositivos de E/S se realiza con una combinación de encuesta, interrupciones, asignación en el espacio de memoria y órdenes de E/S

especiales. Responda a las siguientes preguntas sobre la comunicación con subsistemas de E/S para las aplicaciones de la tabla que utilizan combinaciones de las técnicas anteriores.

a.	Controlador de videojuegos
b.	Monitor de computador

6.9.1 [5]<6.6> Describa la técnica de encuesta. Para las aplicaciones de la tabla, ¿ésta técnica apropiada para la comunicación con la CPU?

6.9.2 [5]<6.6> Describa la técnica de comunicaciones dirigidas por interrupciones. Para las aplicaciones de la tabla, si la encuesta no es una técnica apropiada, explique la técnica dirigida por interrupciones que podría utilizarse.

6.9.3 [10]<6.6> Para las aplicaciones de la tabla, esboce el diseño de una comunicación con asignación en el espacio de memoria. Indique las posiciones de memoria reservadas e indique sus contenidos.

6.9.4 [10]<6.6> Para las aplicaciones de la tabla, esboce el diseño de una comunicación basada en órdenes especiales. Indique las órdenes y su interacción con el dispositivo.

6.9.5 [5]<6.6> ¿Tiene sentido definir un subsistema de E/S que use una combinación de asignación en el espacio de memoria y comunicación basada en órdenes especiales? Razone la respuesta.

Ejercicio 6.10

La sección 6.6 define un proceso de 8 pasos para el procesamiento de las interrupciones. Los registros de Causa y de Estado proporcionan, conjuntamente, información sobre la causa de la interrupción y el estado del sistema de procesamiento de la interrupción. Respondiendo a las siguientes preguntas sobre la combinación de interrupciones de la tabla, investigue el procesamiento de las interrupciones.

a.	Apagado	Sobrecalentamiento	Controlador de Ethernet
b.	Sobrecalentamiento	Reinicio	Controlador del ratón

6.10.1 [5]<6.6> Cuando se detecta una interrupción, el registro de Estado se guarda y todas las interrupciones, excepto las de mayor prioridad, se deshabilitan. ¿Por qué se deshabilitan las interrupciones de baja prioridad? ¿Por qué se guarda el registro de Estado antes de deshabilitar las interrupciones?

6.10.2 [10]<6.6> Priorice la interrupciones indicadas en la tabla.

6.10.3 [10]<6.6> Esboce como se procesaría cada una de la interrupciones de la tabla.

6.10.4 [5]<6.6> ¿Qué ocurre si el bit de habilitación del registro de Causa no se activa al procesar la interrupción? ¿Qué valor debería tomar la máscara de interrupción para lograr el mismo efecto?

6.10.5 [5]<6.6> La mayoría de los sistemas dirigidos por interrupciones se implementan en el sistema operativo. ¿Qué soporte hardware se podría añadir para obtener un procesamiento de las interrupciones más eficiente? Compare la respuesta con el potencial soporte hardware para las llamadas a funciones.

6.10.6 [5]<6.6> En algunas implementaciones dirigidas por interrupciones, una interrupción origina un salto inmediato a un vector de interrupción. En lugar de un registro de Causa, donde cada interrupción activa un bit, cada interrupción tiene su propio vector de interrupciones. Con esta alternativa, ¿se puede utilizar el mismo sistema de prioridad de interrupciones? ¿Hay alguna ventaja en este esquema?

Ejercicio 6.11

El Acceso Directo a Memoria (DMA) permite que los dispositivos accedan directamente a memoria en lugar de acceder a través de la CPU. Así se pueden mejorar de forma muy importante las prestaciones de los periféricos, pero añade cierta complejidad a la implementación del sistema de memoria. Respondiendo a las siguientes preguntas sobre los periféricos de la tabla, investigue las implicaciones del DMA.

a.	Tarjeta gráfica
b.	Tarjeta de sonido

6.11.1 [5]<6.6> ¿Renuncia la CPU al control de la memoria cuando se utiliza el DMA? Por ejemplo, ¿puede un periférico comunicarse con la memoria directamente evitando completamente la CPU?

6.11.2 [10]<6.6> De los periféricos de la tabla, ¿cuál podría beneficiarse del DMA? ¿Con qué criterios se determina si el DMA es ventajoso?

6.11.3 [10]<6.6> De los periféricos de la tabla, ¿con cuál se podrían plantear problemas de coherencia con la cache? ¿Con qué criterios se determina si debe tenerse en cuenta la coherencia?

6.11.4 [5]<6.6> Describa los problemas que podrían producirse al combinar DMA y memoria virtual. De los periféricos de la tabla, ¿con cuál se producirían tales problemas? ¿Cómo pueden evitarse?

Ejercicio 6.12

Las medidas de las prestaciones de la E/S pueden variar significativamente de una aplicación a otra. Si en algunas ocasiones las prestaciones están dominadas por el

número de transacciones procesadas, en otras están dominadas por la productividad de los datos. Respondiendo a las siguientes preguntas sobre las aplicaciones de la tabla, investigue la evaluación de las prestaciones de la E/S.

a.	Búsqueda en la web
b.	Edición de sonido

6.12.1 [10]<6.7> Para las aplicaciones de la tabla, ¿son las prestaciones de la E/S las que determinan las prestaciones del sistema?

6.12.2 [10]<6.7> Para las aplicaciones de la tabla, ¿se miden mejor las prestaciones de la E/S utilizando la productividad de los datos?

6.12.3 [5]<6.7> Para las aplicaciones de la tabla, ¿se miden mejor las prestaciones de la E/S utilizando el número de transacciones procesadas?

6.12.4 [5]<6.7> ¿Hay alguna relación entre las medidas de las prestaciones de los dos problemas anteriores y la elección entre las técnicas de encuestas o comunicaciones dirigidas por interrupciones?

Ejercicio 6.13

Los programa de prueba juegan un papel importante en la evaluación y selección de los dispositivos periféricos. Para que los programa de prueba sean realmente útiles, deben tener propiedades similares a las que se encontrará un dispositivo durante su vida operativa. Respondiendo a las siguientes preguntas sobre las aplicaciones de la tabla, investigue los programas de prueba y la selección de los dispositivos.

a.	Búsqueda en la web
b.	Edición de sonido

6.13.1 [5]<6.7> Defina, para las aplicaciones de la tabla, las características que debería tener un conjunto de programas de prueba de evaluación de un subsistema de E/S.

6.13.2 [5]<6.7> Utilice recursos en línea o de la biblioteca para identificar un conjunto estándar de programas de prueba para las aplicaciones de la tabla. ¿Por qué es útil un conjunto estándar de programas de prueba?

6.13.3 [5]<6.7> ¿Tiene sentido evaluar un subsistema de E/S separadamente del sistema del que forma parte? ¿Qué hay de la evaluación de una CPU?

Ejercicio 6.14

La estrategia más popular para introducir paralelismo y redundancia en los sistemas de almacenamiento es el RAID. El nombre Conjunto Redundante de Discos

Económicos tiene varias implicaciones que exploramos en el contexto de las actividades de la tabla.

a.	Servicios de bases de datos en línea
b.	Edición de sonido

6.14.1 [10]<6.9> RAID 0 utiliza el desmontado para forzar accesos paralelos en varios discos. ¿Por qué esta técnica mejora las prestaciones del disco? Para cada una de las actividades de la tabla, ¿ayuda el desmontado a mejorar las prestaciones?

6.14.2 [5]<6.9> RAID 1 hace copias de los datos en varios discos. Suponiendo que los discos económicos tienen un MTBF menor que los discos caros, ¿cómo puede obtenerse un MTBF menor utilizando redundancia con discos económicos? Explique la respuesta utilizando la definición matemática del MTBF. Para cada una de las actividades de la tabla, ¿ayuda la réplica en espejo a mejorar las prestaciones?

6.14.3 [5]<6.9> Al igual que RAID 1, RAID 3 proporciona mayor disponibilidad de los datos. Explique las ventajas y desventajas de RAID1 y RAID3. Las actividades de la tabla, ¿obtendrían alguna ventaja al utilizar RAID 3 en lugar de RAID 3?

Ejercicio 6.15

Tanto RAID 3 como RAID 4 y RAID 5 utilizan un sistema de paridad para proteger los bloques de datos. Específicamente, se asocia un bloque de paridad con un conjunto de bloques de datos. Cada fila de la tabla muestra los valores de los bloques de datos y de paridad, tal como se ha descrito en la figura 6.13.

	Nuevo D0	D0	D1	D2	D3	P
a.	FEFE	00FF	A387	F345	FF00	4582
b.	AB9C	F457	0098	00FF	2FFF	A387

6.15.1 [10]<6.9> Calcule la nueva paridad P' para RAID 3.

6.15.2 [10]<6.9> Calcule la nueva paridad P' para RAID 4.

6.15.3 [5]<6.9> ¿Cuál es más eficiente, RAID3 o RAID 4? ¿Hay alguna razón que haga que RAID 3 sea mejor que RAID 4?

6.15.4 [5]<6.9> RAID 4 y RAID 5 usan aproximadamente el mismo mecanismo para el cálculo y almacenamiento de la paridad de un bloque de datos. ¿En qué difieren RAID 4 y RAID 5? ¿En qué aplicaciones es más eficiente RAID 5?

6.15.5 [5]<6.9> RAID 4 y RAID 5 mejoran las prestaciones de RAID 3 a medida que crece el tamaño del bloque protegido. ¿Por qué? ¿Hay alguna situación en la que RAID 3 es más eficiente que RAID 4 y RAID 5?

Ejercicio 6.16

La aparición de los servidores Web para comercio electrónico, almacenamiento en línea y comunicaciones ha transformado a los servidores de disco en una aplicación crítica. Dos métricas usuales para evaluar las prestaciones de los servidores de disco son la disponibilidad y la velocidad, pero una tercera métrica está cobrando cada vez mayor importancia, el consumo de potencia. Conteste a las siguientes preguntas sobre la configuración y evaluación de servidores de disco con los siguientes parámetros.

	Instrucciones de programa / operaciones de E/S	Instrucciones del SO/operaciones de E/S	Carga de trabajo (KB leídos)	Velocidad del procesador (Instrucciones/ segundo)
a.	250 000	50 000	128	4000 millones
b.	100 000	50 000	64	4000 millones

6.16.1 [10]<6.8, 6.10> Encuentre la razón de E/S sostenida máxima para lecturas y escrituras aleatorias. Ignore los conflictos de disco y suponga que el controlador de RAID no es el cuello de botella. Siga la estrategia de la sección 6.10 y haga suposiciones similares cuando sea necesario.

6.16.2 [10]<6.8, 6.10> Suponga que está configurando un servidor Sun Fire x4150 tal como se ha descrito en la sección 6.10. Determine si una configuración con 8 discos tendrá un cuello de botella en la E/S. Repita el problema para configuraciones con 16, 4 y 2 discos.

6.16.3 [10]<6.8, 6.10> Determine si el bus PCI, la DIMM o el bus del sistema constituyen un cuello de botella de E/S. Utilice los parámetros y las suposiciones de la sección 6.10.

6.16.4 [5]<6.8, 6.10> Explique por qué un sistema real utiliza programas de prueba o aplicaciones reales para evaluar las prestaciones reales.

Ejercicio 6.17

Determinar las prestaciones de un servidor con un conjunto de datos de información relativamente completo es fácil. Sin embargo, cuando se comparan servidores de diferentes fabricantes que proporcionan datos diferentes, es más complicado. Respondiendo a las siguientes preguntas sobre las aplicaciones de la tabla, investigue el proceso de encontrar y evaluar servidores

Servidor Web

6.17.1 [15]<6.8, 6.10> Para las aplicaciones de la tabla, identifique las características de tiempo de ejecución de un sistema operativo. Escoja características que soportarán una evaluación similar a la realizada en el ejercicio 6.16.

6.17.2 [15]<6.8, 6.10> Encuentre un servidor disponible en el mercado que pueda ser adecuado para la aplicación de la tabla. Antes de evaluar el servidor, explique las razones por las que lo ha escogido.

6.17.3 [20]<6.8, 6.10> Utilizando métricas similares a las utilizadas en el capítulo 6 y el ejercicio 6.16, evalúe el servidor seleccionado en el problema 6.17.2 y compárelo con el servidor sun Fire x4150 del ejercicio 6.16. ¿Cuál escogería? ¿Le han sorprendido los resultados del análisis? Concretamente, ¿cambiaría su elección?

6.17.4 [15]<6.8, 6.10> Identifique un conjunto estándar de programas de prueba útil para comparar el servidor elegido en el problema 6.17.2 y el Sun Fire x4150.

Ejercicio 6.18

Las medidas y estadísticas que proporcionan por los vendedores de sistemas de almacenamiento deben interpretarse con cuidado para obtener predicciones ajustadas sobre el comportamiento del sistema. En la siguiente tabla se muestran datos de varios discos.

	Número de discos	Horas/disco	Horas/fallo
a.	1000	8 760	1 000 000
b.	1000	10 512	1 500 000

6.18.1 [10]<6.12> Calcule la frecuencia de fallos anual (AFR) de los discos de la tabla.

6.18.2 [10]<6.12> Suponga que la frecuencia de fallos anual varía a lo largo de la vida útil de los discos de la tabla. Específicamente, suponga que el AFR es tres veces mayor en el primer mes de funcionamiento y que a partir del quinto año se duplica cada año. ¿Cuántos discos deberían reemplazarse después de 7 años de funcionamiento? ¿Y después de 10 años?

6.18.3 [10]<6.12> Suponga que los discos con frecuencias de fallos bajas son más caros. Es decir, los discos más caros duplicarán cada año su frecuencia de fallos a partir del séptimo año, en lugar del quinto. Si se quiere mantener los discos operativos durante 7 años, ¿cuánto más deberíamos pagar por los discos? ¿Y durante 10 años?

Ejercicio 6.19

Suponga que, para los discos del ejercicio 6.18, el vendedor ofrece una configuración RAID 0 que incrementa la productividad del sistema de almacenamiento un 70%, y una configuración RAID 1 que disminuye el AFR de la pareja de discos en un factor 2. Suponga que el coste de cada configuración es 1.6 veces el coste de la solución original.

6.19.1 [5]<6.9, 6.12> A partir únicamente de los parámetros originales, ¿recomendaría la actualización a RAID 0 o RAID 1? Suponga que los parámetros de los discos individuales son los de la tabla del ejercicio 6.18.

6.19.2 [5]<6.9, 6.12> Suponiendo que su compañía trabaja con un motor de búsqueda global con una gran granja de discos, ¿tendría sentido, desde el punto de vista económico, una actualización a RAID 0 o RAID 1? Suponga que su modelo de prestaciones está basado en el número de anuncios servidos.

6.19.3 [5]<6.9, 6.12> Repita el problema 6.19.2 para una gran granja de discos de una compañía de copias de seguridad en línea. ¿Tendría sentido, desde el punto de vista económico, una actualización a RAID 0 o RAID 1? Suponga que el modelo de prestaciones está basado en la disponibilidad del servidor.

Ejercicio 6.20

La evaluación del día a día y el mantenimiento de computadores operativos implica muchos de los conceptos discutidos en el capítulo 6. Con las siguientes preguntas se exploran las complejidades de la evaluación de los sistemas.

6.20.1 [20]<6.10, 6.12> Configure el Sun Fire x4150 para proporcionar 10 terabytes de almacenamiento para un sistema con 1000 procesadores que ejecutan simulaciones de bioinformática. La configuración debe minimizar el consumo de potencia y tener en cuenta la productividad y disponibilidad del conjunto de discos. Asegúrese de que tiene en cuenta las propiedades de simulaciones complejas a la hora de determinar la configuración más adecuada.

6.20.2 [20]<6.10, 6.12> Recomiende un sistema de copias de seguridad y archivo de datos para el conjunto de discos del problema 6.20.1. Compare y contraste las capacidades de discos, cintas y almacenamiento en línea. Utilice recursos de internet o de la biblioteca para identificar los posibles servidores. Evalúe el coste y la idoneidad para la aplicación con los parámetros descritos en el capítulo 6. Seleccione parámetros de comparación utilizando las propiedades de la aplicación y de los requisitos indicados.

6.20.3 [15]<6.10, 6.12> Los vendedores de los sistemas identificados en el problema 6.20.2 han ofrecido la posibilidad de evaluar sus sistemas *in situ*. Identifique los programas de prueba que usaría para determinar el mejor sistema para su aplicación. Determine el tiempo necesario para recolectar los datos necesarios para hacer la evaluación.

Respuestas a las autoevaluaciones

$6.2, página 575: 2 y 3 son verdaderas.
$6.3, página 579: 3 y 4 son verdaderas.
$6.4, página 582: Todas son verdaderas (suponiendo que 40 MB/s es comparable a 100 MB/s).
$6.5, página 585: 1 es verdadera.
$6.6, página 594: 1 y 2.
$6.7, página 598: 1 y 2. 3 es falsa porque la mayoría de los programas de prueba TPC incluyen el coste.
$6.9, página 605: Todas son verdaderas.

7

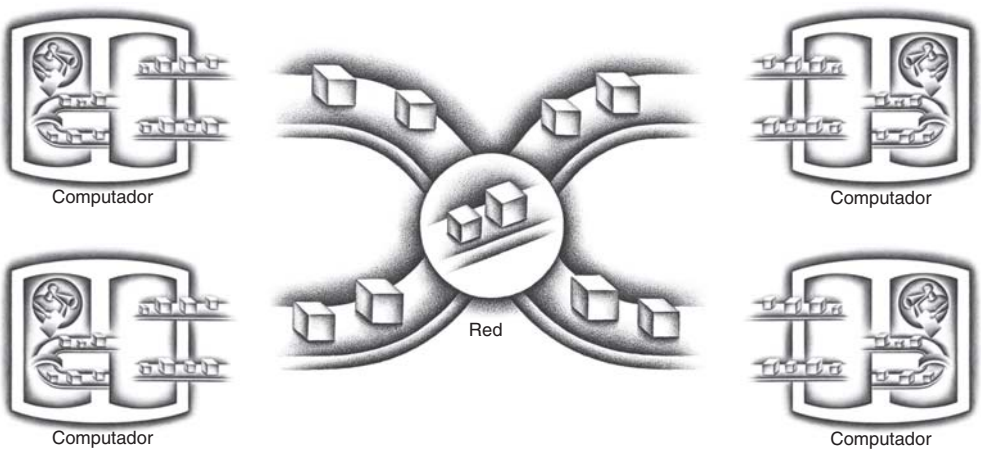
Multinúcleos, multiprocesadores y clústeres

Hay en el mar mejores peces que los que hasta ahora se han podido pescar.

Proverbio irlandés

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Organización de los multiprocesadores y clústeres



"Más allá del horizonte de la luna y de sus montes, y del valle de las sombras cabalga, donoso calbaga, —le dijo la sombra—, ¡si vas en busca de El dorado!

Edgar Allan Poe, "El Dorado", estrofa 4, 1849

7.1 Introducción

Los arquitectos de computadores han buscado durante mucho tiempo El Dorado del diseño de los computadores: crear sistemas potentes simplemente conectando muchos computadores más pequeños ya existentes. Esta visión dorada es el origen de los **multiprocesadores**. Idealmente, los clientes encargan tantos procesadores como puedan permitirse y reciben un computador con unas prestaciones proporcionales al número de procesadores. Para ello, el software de los multiprocesadores debe diseñarse para funcionar correctamente con un número variable de procesadores. Como se ha mencionado en el capítulo 1, el consumo de potencia se ha convertido en el principal enemigo de centros de datos y microprocesadores. Se pueden obtener mejores prestaciones por vatio o por julio reemplazando los procesadores grandes e ineficientes por muchos procesadores más pequeños y eficientes, siempre y cuando el software sea capaz de aprovecharlos al máximo. Así, en los multiprocesadores se unen la mejora de la eficiencia energética y la escalabilidad de las prestaciones.

Como el software de los multiprocesadores debe ser escalable, algunos diseños mantienen su operatividad aún en presencia de fallos en el hardware, es decir, si en un multiprocesador con n procesadores falla uno, el sistema podría continuar funcionando correctamente con $n–1$ procesadores. De este modo se mejora la disponibilidad (véase el capítulo 6).

El término altas prestaciones puede significar alta productividad de trabajos independientes, lo que se llama **paralelismo a nivel de trabajos** o **paralelismo a nivel de procesos**. Estos trabajos paralelos son aplicaciones independientes y actualmente constituyen una carga de trabajo importante y habitual de los computadores paralelos. Utilizaremos el término **programa de procesamiento paralelo** para referirnos a un programa que se ejecuta en varios procesadores simultáneamente.

En las últimas décadas ha habido grandes problemas científicos que han necesitado computadores mucho más rápidos que los disponibles en cada momento, y estos problemas han servido para justificar muchos diseños de computadores paralelos. Hablaremos de algunos de ellos en este capítulo. Algunos de estos problemas pueden ser resueltos con un **clúster** formado por microprocesadores alojados en servidores o PCs independientes. Además, los clústeres son útiles también para otras aplicaciones no científicas, tales como motores de búsqueda, servidores Web y bases de datos.

Como se ha descrito en el capítulo 1, el problema del consumo de potencia ha propiciado que los multiprocesadores sean el centro de atención, y los futuros aumentos de las prestaciones aparentemente pasan por poner más procesadores en cada chip, en lugar de disponer de mayores de frecuencias de reloj o mejores CPI. Estos sistemas se llaman **microprocesadores multinúcleo** en lugar de microprocesadores multiprocesador, probablemente para evitar nombres redundantes. En consecuencia, los procesadores en un chip multinúcleo se llaman **núcleos**. La tendencia actual es duplicar el número de núcleos cada dos años y, por lo tanto, los programadores que tienen que preocuparse del rendimiento deben reconvertirse en programadores paralelos, ya que programa secuencial es equivalente a programa lento.

Multiprocesador: sistema con al menos dos procesadores. En contraste, el monoprocesador tiene un único procesador.

Paralelismo a nivel de trabajos o paralelismo a nivel de procesos: utilización de varios procesadores para la ejecución de varios programas independientes simultáneamente.

Programa de procesamiento paralelo: un único programa que se ejecuta en varios procesadores simultáneamente.

Clúster: conjunto de computadores conectados con una red de área local (*Local area network*, LAN) que opera como un único multiprocesador.

Microprocesador multinúcleo: microprocesador que contiene varios procesadores (núcleos) en el mismo circuito integrado.

El principal reto que debe afrontar la industria de los computadores es crear hardware y software para facilitar la escritura de programas de procesamiento paralelo, que se ejecuten eficientemente en términos de prestaciones y consumo de potencia a medida que el número de núcleos escala geométricamente.

Este giro repentino en el diseño de los microprocesadores ha pillado desprevenido a muchos, por lo que hay mucha confusión en la terminología y en lo que ésta significa. La figura 7.1 intenta clarificar los términos *serie, paralelo, secuencial y concurrente*. Las columnas representan el software que es inherentemente secuencial o concurrente, mientras que las filas representan el hardware que es serie o paralelo. Por ejemplo, los programadores de compiladores piensan que éste es un programa secuencial: los pasos son análisis léxico, análisis sintáctico, generación de código, optimización, etc. Por el contrario, los programadores de sistemas operativos piensan en ellos como un conjunto de programas concurrentes: procesos cooperantes que procesan sucesos de E/S de trabajos independientes que se ejecutan en un computador.

		Software	
		Secuencial	Concurrente
Hardware	Serie	Multiplicación de matrices en MatLab ejecutándose en un Intel Pentium 4	Sistema operativo Windows Vista ejecutándose en un Intel Pentium 4
	Paralelo	Multiplicación de matrices en MatLab ejecutándose en un Intel Xeon e5345 (Clovertown)	Sistema operativo Windows Vista ejecutándose en un Intel Xeon e5345 (Clovertown)

FIGURA 7.1 Categorización de hardware y software y ejemplos del punto de vista de la aplicación sobre concurrencia y el punto de vista del hardware sobre paralelismo.

De la figura 7.1 podemos destacar que el software concurrente puede ejecutarse en un hardware serie, como el sistema operativo en el monoprocesador Intel Pentium 4, o en un hardware paralelo, como el SO en el procesador Intel Xeon e5345 (Clovertown). Lo mismo podemos decir de un software secuencial. Por ejemplo, el programador de MatLab escribe una multiplicación de matrices pensando secuencialmente, pero la puede ejecutar en serie en el Pentium 4 o en paralelo en el Xeon e5345. Se podría pensar que el único reto de la revolución paralela es averiguar como conseguir que el software secuencial tenga altas prestaciones en un hardware paralelo, pero además hay que conseguir que los programas concurrentes tengan altas prestaciones cuando se ejecutan en un multiprocesador y que estas aumenten con el número de procesadores. Teniendo en mente esta distinción, en el resto del capítulo hablaremos de *programa de procesamiento paralelo* o *software paralelo* para referirnos a software secuencial o concurrente que se ejecuta en un hardware paralelo.

En la siguiente sección se describe el porqué es difícil crear programas de procesamiento paralelo eficientes. En las secciones 7.3 y 7.4 se describen dos alternativas de una característica fundamental en el hardware paralelo: la disponibilidad o no de un espacio de direcciones único para todos los procesadores del sistema. Las dos versiones más populares de estas alternativas son los *multiprocesadores de memoria compartida* y los

clústeres. La sección 7.5 describe la ejecución *multihilo*, un término que a menudo se confunde con el multiprocesamiento, en parte porque ambos se apoyan en la concurrencia de los programas. En la sección 7.6 se discute una clasificación más antigua que la mostrada en la figura 7.1, y además se describen dos estilos de la arquitectura del repertorio de instrucciones que soportan la ejecución de aplicaciones secuenciales en hardware paralelo, SIMD y vectorial. La sección 7.7. describe un estilo de computador relativamente nuevo, que proviene del ámbito del hardware para computación gráfica, la *Unidad de Procesamiento Gráfico (Graphic Processing Unit, GPU)*. En el apéndice A se describe la GPU con mayor detalle. A continuación, en la sección 7.9, se discute la dificultad de encontrar programas de prueba paralelos. En la siguiente sección se describe un modelo de prestaciones nuevo, sencillo e intuitivo que sirve de ayuda en el diseño de aplicaciones y arquitecturas. Utilizamos este modelo en la sección 7.11 para evaluar cuatro computadores multinúcleo recientes con dos aplicaciones diferentes. Terminamos con las falacias y errores habituales y las conclusiones para el paralelismo.

Antes de introducirnos en el camino del paralelismo, no podemos olvidar nuestras incursiones previas en los capítulos anteriores:

- Capítulo 2, sección 2.11: Paralelismo e instrucciones: sincronización
- Capítulo 3, sección 3.6: Paralelismo y aritmética del computador: Asociatividad
- Capítulo 4, sección 4.10: Paralelismo y paralelismo a nivel de instrucciones avanzado
- Capítulo 5, sección 5.8: Paralelismo y jerarquías de memoria: coherencia de cache
- Capítulo 6, sección 6.9: Paralelismo y E/S: conjuntos redundantes de discos económicos

Autoevaluación Verdadero o falso: Para beneficiarse de la utilización de un multiprocesador, la aplicación debe ser concurrente.

7.2 La dificultad de crear programas de procesamiento paralelo

La dificultad del paralelismo no está en el hardware; la principal dificultad es que muy pocas aplicaciones se han reescrito para completar sus tarea en menos tiempo en un multiprocesador. Es difícil escribir software que utilice varios procesadores para terminar una tarea de forma más rápida, y el problema empeora al aumentar el número de procesadores.

¿Por qué esto es así? ¿Por qué es más difícil desarrollar programas de procesamiento paralelo que programas secuenciales?

La razón principal es que las prestaciones y la eficiencia de un programa de procesamiento paralelo en un multiprocesador *tienen* que ser mejores; en caso contrario es preferible utilizar un programa secuencial en un monoprocesador, ya que la programación es más sencilla. De hecho, las técnicas de diseño de monoprocesadores, tales como superescalar y ejecución fuera de orden, aprovechan el paralelismo a nivel de

instrucción (véase el capítulo 4), normalmente sin la participación del programador. Estas técnicas disminuyen la demanda de reescritura de programas para multiprocesadores, porque sin que los programadores tengan que hacer nada más, sus programas secuenciales se ejecutarán más rápido en los nuevos computadores.

¿Por qué es difícil escribir programas de procesamiento paralelo que sean rápidos, especialmente cuando aumenta el número de procesadores? En el capítulo 1 se utilizó la analogía de ocho periodistas intentando escribir un artículo con la esperanza de hacer el trabajo ocho veces más rápido. Para conseguirlo, la tarea debe dividirse en ocho partes de la misma complejidad, porque en caso contrario algunos periodistas estarían sin hacer nada, esperando a que terminasen los que tienen las partes más complejas. Otro riesgo para las prestaciones se produciría cuando los periodistas tienen que emplear demasiado tiempo en comunicarse con sus compañeros en lugar de trabajar en su parte de la tarea. Podemos extraer algunas conclusiones de esta analogía con la programación paralela, los desafíos incluyen la planificación de las tareas, el equilibrio de la carga, el tiempo de sincronización y la sobrecarga de las comunicaciones entre colaboradores. El reto es todavía más difícil cuantos más periodistas participen en la escritura del artículo y cuantos más procesadores estén disponibles para la programación paralela.

Nuestra discusión del capítulo 1 revela otro obstáculo, la ley de Amdahl. Esta ley nos recuerda que incluso las partes pequeñas de un programa deben paralelizarse si se quiere hacer un buen uso de los muchos núcleos disponibles.

El reto de la aceleración

EJEMPLO

Suponga que queremos que un programa se ejecute 90 veces más rápido con 100 procesadores. ¿Qué parte del programa original puede ser secuencial?

RESPUESTA

La ley de Amdahl (capítulo 1) dice

tiempo de ejecución después de la mejora =

$$\frac{\text{Tiempo de ejecución afectado por la mejora}}{\text{Total de la mejora}} + \text{Tiempo de ejecución no afectado}$$

Se puede reformular la ley de Amdahl en términos de aceleración frente al tiempo de ejecución de la versión original:

$$\text{Aceleración} = \frac{\text{Tiempo de ejecución antes}}{\left(\begin{array}{c}\text{Tiempo de}\\ \text{ejecución}\\ \text{antes}\end{array} - \begin{array}{c}\text{Tiempo de}\\ \text{ejecución}\\ \text{afectado}\end{array}\right) + \frac{\text{Tiempo de ejecución afectado}}{100}}$$

Esta ecuación se reescribe considerando que el tiempo de ejecución antes es 1 para alguna unidad de tiempo y expresando el tiempo de ejecución afectado por la mejora como un porcentaje del tiempo de ejecución original:

$$\text{Aceleración} = \frac{1}{(1 - \text{Porcentaje tiempo afectado}) + \frac{\text{Porcentaje tiempo afectado}}{100}}$$

Sustituyendo nuestro objetivo, aceleración igual a 90, en la anterior ecuación:

$$90 = \frac{1}{(1 - \text{Porcentaje tiempo afectado}) + \frac{\text{Porcentaje tiempo afectado}}{100}}$$

Simplificando y despejando el porcentaje de tiempo afectado:

$$90 \times (1 - 0.99 \times \text{porcentaje de tiempo afectado}) = 1$$
$$90 - (90 \times 0.99 \times \text{porcentaje de tiempo afectado}) = 1$$
$$90 - 1 = 90 \times 0.99 \times \text{porcentaje de tiempo afectado}$$
$$\text{porcentaje de tiempo afectado} = 89/89.1 = 0.999$$

Es decir, para obtener una aceleración igual a 90 con 100 procesadores, el porcentaje secuencial debe ser sólo del 0.1%.

Aún así, hay aplicaciones con abundante paralelismo.

El reto de la aceleración: el mayor problema

EJEMPLO

Suponga que queremos hacer dos sumas: una suma de 10 variable escalares y una suma matricial de dos matrices bidimensionales 10 por 10. ¿Qué aceleración se obtendrá con 10 y 100 procesadores? Ahora calcule la aceleración suponiendo que las matrices son de 100 por 100.

RESPUESTA

Si aceptamos que las prestaciones son función del tiempo de suma, t, entonces hay 10 sumas que no se benefician de la utilización de procesadores paralelos y 100 que sí. Si el tiempo en un procesador es $110t$, el tiempo de ejecución con 10 procesadores es

$$\text{Tiempo de ejecución después de la mejora} = \frac{\text{Tiempo de ejecución afectado por la mejora}}{\text{total de la mejora}} + \text{Tiempo de ejecución no afectado}$$

$$\text{Tiempo de ejecución afectado por la mejora} = \frac{100t}{10} + 20t = 20t$$

Entonces la aceleración con 10 procesadores es $110t/20t = 5.5$. El tiempo de ejecución con 100 procesadores es

$$\text{Tiempo de ejecución después de la mejora} = \frac{100t}{100} + 10t = 11t$$

Entonces la aceleración con 100 procesadores es $110t/11t = 10$.

Así, para este tamaño de problema, se obtiene el 55% de la máxima aceleración posible con 10 procesadores, pero solo el 10% con 100. Veamos qué ocurre cuando aumentamos las dimensiones de la matriz. El tiempo de ejecución

del programa secuencial ahora es $10t + 10\,000t = 10\,010t$. El tiempo de ejecución con 10 procesadores es

$$\text{Tiempo de ejecución después de la mejora} = \frac{10\,000t}{10} + 10t = 1010t$$

Entonces la aceleración con 10 procesadores es $10\,010t/1010t = 9.9$. El tiempo de ejecución con 100 procesadores es

$$\text{Tiempo de ejecución después de la mejora} = \frac{10\,000t}{100} + 10t = 110t$$

Entonces la aceleración con 100 procesadores es $10\,010t/110t = 91$. Así, para un tamaño de problema mayor, obtenemos aproximadamente el 99% de la máxima aceleración posible con 10 procesadores, y más del 90% con 100.

Estos ejemplos ponen de manifiesto que en un multiprocesador es más difícil obtener buenas aceleraciones manteniendo el tamaño del problema fijo que incrementando el tamaño del problema. Esto nos permite introducir dos términos que describen dos formas diferentes de escalado. **Escalado fuerte** hace referencia a la aceleración con un tamaño de problema fijo y **escalado débil** implica que el tamaño del problema aumenta proporcionalmente al número de procesadores. Supongamos que el tamaño del problema, M, es el conjunto de trabajo en memoria principal y que se dispone de P procesadores. Entonces, la memoria por procesador en escalado fuerte es aproximadamente M/P, y en escalado débil es aproximadamente M.

Escalado fuerte: aceleración obtenida en un multiprocesador sin aumentar el tamaño del problema.

Escalado débil: aceleración obtenida en un multiprocesador aumentando el tamaño del problema proporcionalmente al número de procesadores.

Dependiendo del tipo de aplicación, se puede defender cualquier tipo de escalado. Por ejemplo, los programas de pruebas TPC-C de una base de datos crédito-débito (capítulo 6) requieren que el número de cuentas de clientes aumente para aumentar el número de transacciones por minuto. El argumento es lo absurdo que resulta pensar que un número dado de clientes, de repente, van a utilizar los cajeros automáticos 100 veces al día, simplemente porque el banco tiene un computador más rápido. En lugar de esto, si se quiere demostrar que el sistema puede multiplicar por 100 el número de transacciones por minuto, se debe hacer el experimento multiplicando por 100 el número de clientes.

Este último ejemplo muestra la importancia del equilibrio de la carga.

El reto de la aceleración: el equilibrio de la carga

EJEMPLO

En el problema anterior, para obtener una aceleración de 91 con 100 procesadores, se supuso que la carga estaba perfectamente equilibrada. Es decir, cada procesador hacía el 1% del trabajo total. Muestre qué impacto tiene sobre la aceleración que un procesador tenga más carga que el resto. Calcule con el 2% y el 5%.

RESPUESTA

Si un procesador tiene el 2% de la carga, entonces tiene que hacer 2% × 10 000 = 200 sumas, y los otros 99 procesadores se reparten las 9800 sumas restantes. Como están trabajando simultáneamente, podemos calcular el tiempo de ejecución como un máximo

$$\text{Tiempo de ejecución después de la mejora} = \text{Max}\left(\frac{9800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

La aceleración cae hasta 10 $101t/210t = 48$. Si un procesador tiene el 5% de la carga, debe hacer 500 sumas:

$$\text{Tiempo de ejecución después de la mejora} = \text{Max}\left(\frac{9500t}{99}, \frac{500t}{1}\right) + 10t = 510t$$

La aceleración cae todavía más, hasta 10 $101t/510t = 20$. Este ejemplo pone de manifiesto la importancia del equilibrio de la carga; un procesador con el doble de carga que el resto hace caer la aceleración casi a la mitad, y con una carga 5 veces la del resto reduce la aceleración casi en un factor 5.

Autoevaluación Verdadero o falso: El escalado fuerte no está limitado por la ley de Amdahl.

7.3 Multiprocesadores de memoria compartida

Dada la dificultad de reescribir los programas antiguos para que se ejecuten correctamente en un hardware paralelo, la pregunta natural que cualquiera podría hacerse es qué hacen los diseñadores de computadores para facilitar esta tarea. Una respuesta a esta pregunta es que, en algunos sistemas, proporcionan un espacio de direcciones físicas único compartido por todos los procesadores. Así, los programas no tienen que preocuparse de donde se van a ejecutar, ya que lo único que necesitan saber es que se van a ejecutar en paralelo. Con este enfoque, podemos hacer que todas las variables de un programa estén disponibles para todos los procesadores en cualquier momento. La alternativa es tener espacios de direcciones separados para cada procesador, donde la compartición debe hacerse de modo explícito; esta opción se describe en la siguiente sección. Cuando el espacio de direcciones es común —lo usual en los chips multinúcleo— el hardware proporciona coherencia de cache para tener una visión coherente de la memoria compartida (véase sección 5.8 del capítulo 5).

Sistema multiprocesador de memoria compartida (SMP): procesador paralelo con un espacio de direcciones único, que implica comunicaciones implícitas con cargas y almacenamientos.

Acceso a memoria uniforme (UMA): multiprocesador que tiene siempre el mismo tiempo de acceso a memoria principal sin importar qué procesador hace el acceso y qué palabra es la solicitada.

Un **multiprocesador de memoria compartida (*shared memory multiprocessor*, SMP)** es un sistema que ofrece al programador un *espacio de direcciones físicas único* para todos los procesadores, aunque un término más apropiado habría sido multiprocesador de *direcciones* compartidas. Observe que estos sistemas pueden ejecutar trabajos independientes en su espacio de direcciones virtuales propio, incluso compartiendo el espacio de direcciones físicas. La comunicación entre procesadores se realiza a través de variables compartidas en memoria, y todos los procesadores tienen la capacidad de acceder a cualquier posición de memoria vía cargas y almacenamientos. La organización clásica de un SMP se muestra en la figura 7.2.

Hay dos tipos de multiprocesadores con espacio de direcciones único. En el primer tipo todos los accesos a memoria tardan el mismo tiempo, independientemente de qué procesador hace el acceso y de qué palabra de memoria se solicita. Estos sistemas se llaman multiprocesadores con **acceso a memoria uniforme (*uniform memory access*, UMA)**. En el segundo tipo, algunos accesos a memoria son más rápidos que otros, dependiendo de qué procesador quiere acceder a que palabra. Tales sistemas se

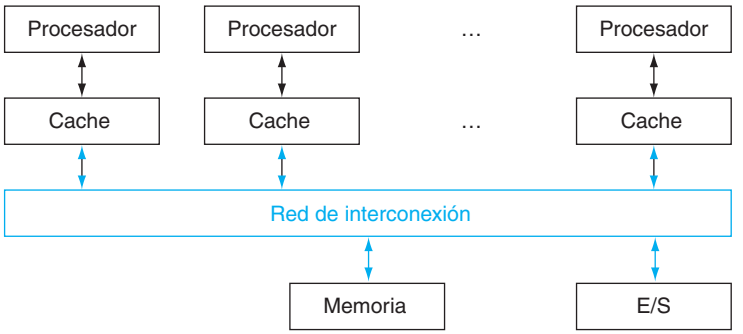


FIGURA 7.2 Organización clásica de un multiprocesador de memoria compartida.

denominan multiprocesadores con **acceso a memoria no uniforme (*nonuniform memory access*, NUMA)**. Como era de esperar, es más difícil programar un multiprocesador NUMA que un multiprocesador UMA, pero los sistemas NUMA escalan mejor a tamaños (número de procesadores) más grandes y tienen latencias de acceso a memoria menores en el acceso a la memoria cercana al procesador.

Como normalmente los procesadores que operan en paralelo comparten datos, es necesario introducir una coordinación en la operación sobre los datos compartidos; de lo contrario, un procesador podría comenzar a operar sobre un dato antes de que otro procesador terminase de modificar ese mismo dato. Esta coordinación se llama **sincronización**. Cuando la compartición se lleva a cabo en un espacio de direcciones único, es necesario disponer de un mecanismo separado de sincronización. Una de las alternativas es el uso de **bloqueos (*lock*)** para una variable compartida. En cada instante un solo procesador puede hacerse con el bloqueo, y el resto de los procesadores interesados en acceder a la variable compartida deben esperar hasta que el primero desbloquea la variable. En la sección 2.11 del capítulo 2 se describen las instrucciones de MIPS para el bloqueo.

Acceso a memoria no uniforme (NUMA): tipo de multiprocesador con espacio de direcciones único en el cual algunos accesos a memoria son mucho más rápidos que otros dependiendo de qué procesador hace el acceso y a qué palabra se accede.

Sincronización: proceso de coordinar el comportamiento de dos o más procesos que pueden estar ejecutándose en procesadores diferentes.

Bloqueo: mecanismo de sincronización que sólo permite el acceso a un dato compartido de un procesador en cada instante.

Un programa de procesamiento paralelo simple para un espacio de direcciones compartido

EJEMPLO

Suponga que queremos sumar 10 000 números en un multiprocesador de memoria compartida con tiempos de acceso a memoria uniformes y 100 procesadores.

RESPUESTA

Una vez más, el primer paso es dividir el conjunto de números en subconjuntos del mismo tamaño. No se reparten los subconjuntos en espacios de memoria diferentes porque hay un único espacio de memoria; lo único que se necesita es darle direcciones de comienzo diferentes a cada procesador. `Pn` es un número entre 0 y 99 que identifica al procesador. Todos los procesadores ejecutan un lazo que suma su subconjunto de números:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i+1)
  sum[Pn] = sum[Pn] + A[i]; /* suma el subconjunto asignado */
```

Reducción: función que procesa una estructura de datos y devuelve un solo valor.

El siguiente paso es sumar estas sumas parciales. Esta acción se llama **reducción**. Divide y vencerás. La mitad de los procesadores suman parejas de sumas parciales, un cuarto suma parejas de las nuevas sumas parciales, y así hasta que se obtiene el resultado final. La naturaleza jerárquica de esta reducción se ilustra en la figura 7.3.

En este ejemplo, los dos procesadores deben sincronizarse antes de que el procesador "consumidor" intente leer de memoria los resultados obtenidos por procesador "productor"; en caso contrario, el consumidor puede leer un valor antiguo del dato. Cada procesador debe tener su propia versión de la variable `i` contador del lazo, por lo tanto debemos indicar que es una variable "privada". Este es el código (`mitad` también es una variable privada):

```
mitad=100; /* 100 procesadores */
repeat
      synch( ); /* espera por terminación de suma parcial*/
      if (mitad%2 != 0 && Pn == 0)
          sum[0] = sum[0] + sum[mitad-1];
          /* suma condicional necesaria cuando mitad es
         impar; el procesador 0 tiene el elemento que falta */
      mitad = mitad/2; /* dividir línea de sumas */
      if (Pn < mitad) sum[Pn] = sum[Pn] + sum[Pn+mitad];
until (mitad == 1) ; /* finalizar cuando la suma es sum[0] */
```

Autoevaluación

Verdadero o falso: Los multiprocesadores de memoria compartida no aprovechan el paralelismo a nivel de trabajos.

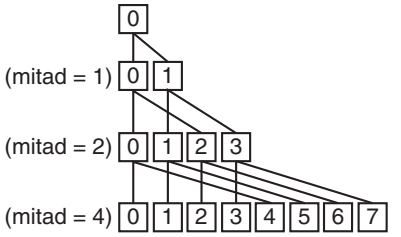


FIGURA 7.3 Últimos cuatro niveles de una reducción que suma los resultados de cada procesador, desde la parte inferior a la superior. Los procesadores cuyo número `i` es menor que `mitad`, suman el valor calculado por el procesador de número (`i+mitad`) a su suma parcial.

Extensión: Una alternativa a la compartición del espacio de direcciones físicas sería tener espacios de direcciones físicas separados pero compartiendo un espacio de direcciones virtuales común, dejando que sea el sistema operativo quien se encargue de las comunicaciones. Esta alternativa se ha probado, pero tenía un sobrecoste demasiado elevado como para ser una abstracción práctica para el programador.

7.4 Clústeres y otros multiprocesadores de paso de mensajes

La alternativa a la compartición del espacio de direcciones es que cada procesador disponga de su propio espacio de direcciones físicas privado. La figura 7.4 muestra la organización clásica de un multiprocesador con múltiples espacios de direcciones privados. Las comunicaciones se tienen que hacer mediante **paso de mensajes** explícito, que tradicionalmente es el nombre que reciben este tipo de computadores. Dado que se dispone de rutinas para **enviar** y **recibir mensajes**, la coordinación se realiza con el paso de mensajes, ya que un procesador sabe cuando se ha enviado un mensaje y el procesador que lo recibe sabe cuando llega un mensaje. Si el procesador que envía un mensaje necesita confirmación de que el mensaje ha llegado a su destinatario, el procesador que lo recibe debe enviar un mensaje de agradecimiento al remitente.

Paso de mensajes: comunicación entre varios procesadores mediante el envío y recepción explícita de información.

Rutina de envío de mensajes: rutina utilizada en los procesadores con memoria privada para enviar información a otro procesador.

Rutina de recepción de mensajes: rutina utilizada en los procesadores con memoria privada para aceptar un mensaje de otro procesador.

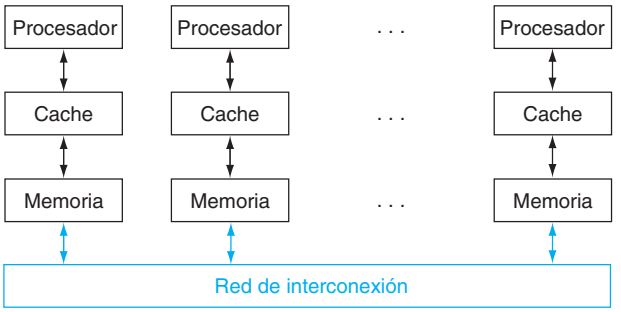


FIGURA 7.4 Organización clásica de un multiprocesador con múltiples espacios de direcciones privados, llamado tradicionalmente multiprocesador de paso de mensajes. Observe que a diferencia del SMP de la figura 7.2, la red de interconexión no se encuentra entre las caches y la memoria, sino entre nodos procesador-memoria.

Algunas aplicaciones concurrentes se ejecutan bien en un hardware paralelo, independientemente de si es de memoria compartida o de paso de mensajes. En particular, el paralelismo a nivel de trabajos y aplicaciones con pocas comunicaciones —como por ejemplo la búsqueda en la Web, servicios de correo electrónico y servidores de ficheros— no necesitan direccionamiento compartido para ejecutarse eficientemente.

Ha habido varios intentos de construir computadores de altas prestaciones basados en redes de paso de mensajes de altas prestaciones, que ofrecían comunicaciones con mejores prestaciones que los clústeres con redes de área local. El problema fue que eran mucho más caros. Dado este elevado coste, pocas aplicaciones podrían justificar la necesidad de unas comunicaciones con mejores prestaciones. De este modo, los **clústeres** se convirtieron en el ejemplo de computador paralelo de paso de mensajes más extendido en la actualidad. Los clústeres son generalmente un conjunto computadores convencionales conectados a través de sus puertos de E/S utilizando conmutadores de red y cables estándar. Cada uno ejecuta una copia diferente del sistema operativo. Virtualmente, cada servicio de internet confía en clústeres de servidores y conmutadores convencionales.

Clústeres: Conjunto de computadores conectados, vía los puertos de E/S, a través de conmutadores de red estándar para formar un multiprocesador de paso de mensajes.

Una de las desventajas de los clústeres ha sido que el coste de administrar un cluster con n computadores es prácticamente el mismo que el de administrar n computadores independientes, mientras que el coste de administrar un multiprocesador de memoria compartida con n procesadores es aproximadamente el mismo que el de administrar un computador.

Esta debilidad es una de las razones de la popularidad de las máquinas virtuales (capítulo 5), que facilitan la administración de los clústeres. Por ejemplo, las MV hacen posible parar y comenzar programas atómicamente, lo que simplifica las actualizaciones de software. Las MV pueden incluso trasladar un programa de un computador en un cluster a otro sin detener el programa, permitiendo que un programa pueda trasladarse en caso de un fallo del hardware.

Otra desventaja de los clústeres es que los procesadores están normalmente conectados a través de los puertos de E/S de cada computador, mientras que los núcleos de un multiprocesador están conectados a través de los puertos de conexión con memoria, que tienen mayor ancho de banda y menor latencia, y permiten una comunicación con prestaciones mucho más elevadas.

Una debilidad final es el sobrecoste de la división de la memoria: un cluster con n máquinas tiene n memorias independientes y n copias del sistema operativo, mientras que un multiprocesador de memoria compartida permite que un único programa utilice casi toda la memoria del computador y sólo necesita una copia del SO.

Eficiencia de la memoria

EJEMPLO

Suponga que un procesador de memoria compartida tiene 20 GB de memoria principal, cinco computadores en cluster tienen 4 GB cada uno y el SO ocupa 1 GB. ¿Cuánto espacio más hay disponible para los usuarios con memoria compartida?

RESPUESTA

La relación entre la memoria para programas de usuario en el sistema de memoria compartida frente al cluster sería:

$$\frac{20-1}{5\times(4-1)}=\frac{19}{15}\approx 1.25$$

Entonces, el espacio disponible en los computadores de memoria compartida es un 25% mayor que en los clústeres.

Recuperemos el ejemplo de suma de la sección anterior para ver el impacto de múltiples memorias privadas y de las comunicaciones explicitas.

Un programa de procesamiento paralelo sencillo para paso de mensajes

EJEMPLO

Suponga que queremos sumar 100 000 números en un multiprocesador de paso de mensajes con 100 procesadores, cada uno con una memoria privada.

RESPUESTA

Como hay varios espacios de direcciones, el primer paso es distribuir los 100 subconjuntos a cada memoria local. El procesador que tiene los 100 000 números envía un subconjunto a cada nodo procesador-memoria.

El siguiente paso es hacer la suma de cada subconjunto. Este paso es un lazo en cada procesador: leer una palabra de la memoria local y sumarla a una variable local:

```
sum = 0;
for (i = 0; i<1000; i = i + 1)/* lazo en cada subconjunto */
  sum = sum + AN[i];  /* suma en los conjuntos locales */
```

El último paso es la reducción de estas 100 sumas parciales. Como cada suma parcial está situada en un procesador diferente, tendremos que enviar las sumas parciales a través de la red de interconexión para acumularlas en la suma final. En lugar de enviar todas las sumas parciales a un procesador, lo que resultaría en una acumulación secuencial de las sumas parciales, utilizamos otra vez la estrategia divide y vencerás.

En primer lugar, la mitad de los procesadores envían sus sumas parciales a la otra mitad de los procesadores, donde se suman dos sumas parciales. A continuación, la cuarta parte de los procesadores (la mitad de la mitad) envía esta nueva suma parcial a otra cuarta parte de los procesadores (la mitad restante de la mitad) para otra ronda de sumas. Así, seguimos dividiendo a la mitad, enviando y recibiendo hasta tener una única suma de todos los números. Supongamos que `Pn` es el número del procesador, `send(x,y)` es una rutina que envía el valor `y` al procesador `x` a través de la red de interconexión y `receive( )` es una función que acepta un valor que llega por la red para este procesador. El código es el siguiente:

```
limite = 100; mitad = 100 /* 100 procesadores */
repeat
  mitad = (mitad+1)/2; /* linea de división envía vs,
  recibe */
  if (Pn >= mitad && Pn < limite) send(Pn - mitad, sum);
  if (Pn < (limite/2)) sum = sum + receive( );
  limite = mitad;  /* límite superior de los remitentes */
until (mitad == 1); /* terminar con suma final */
```

Este código divide a los procesadores en remitentes y receptores, cada receptor recibe sólo un mensaje, por lo tanto, un procesador receptor se parará hasta que recibe un mensaje. Así, enviar y recibir pueden ser utilizados como primitivas de sincronización y comunicación y los procesadores están pendientes de la transmisión de los datos.

Si hay un número impar de nodos, el nodo de la mitad no participa en enviar/recibir. El límite se fija de forma que este nodo es el nodo más alto en la siguiente iteración.

Extensión: Este ejemplo supone implícitamente que el paso de mensajes es tan rápido como la suma, cuando en realidad, el envío y recepción de mensajes es mucho más lento. Una optimización para obtener un mejor equilibrio entre cálculo y comunicación podría ser tener menos nodos recibiendo muchas sumas de otros procesadores.

Interfaz hardware software

Los computadores que realizan la comunicación con paso de mensajes en lugar de memoria compartida con coherencia cache son mucho más fáciles de diseñar (véase sección 5.8 en el capítulo 5). La ventaja desde el punto de vista del programador es la comunicación explícita, lo que significa que hay menos sorpresas en las prestaciones que con los computadores memoria compartida con coherencia cache. El inconveniente es la portabilidad, es más difícil reescribir un programa secuencial para un sistema con paso de mensajes porque cada comunicación debe identificarse con anterioridad o el programa no funcionará correctamente. La memoria compartida con coherencia cache permite que el hardware averigüe qué dato debe compartirse, lo que facilita la portabilidad. A la vista de los pros y contras de la comunicación implícita, hay diferentes opiniones sobre cuál es el camino más corto hacia las altas prestaciones.

Uno de los puntos débiles del cluster, tener memorias separadas para la memoria de usuario, se convierte en una ventaja para la disponibilidad del sistema. Como un cluster es un conjunto de computadores independientes conectados a través de una red de área local, es mucho más fácil reemplazar una máquina sin apagar el sistema que en un SMP. Fundamentalmente, las direcciones compartidas implican que es difícil aislar un procesador y reemplazarlo sin que el sistema operativo tenga que hacer un esfuerzo heroico. Por el contrario, como el software del cluster es una capa que se ejecuta sobre el sistema operativo local de cada computador, es mucho más sencillo desconectar y reemplazar la máquina estropeada.

Puesto que los clústeres se construyen con computadores completos y redes escalables independientes, este aislamiento facilita además la expansión del sistema sin necesidad de cerrar la aplicación que se está ejecutando en la capa más elevada del cluster.

Otros aspectos, como menor coste, alta disponibilidad, eficiencia energética mejorada y posibilidad de expansión rápida e incremental, hacen que los clústeres sean muy atractivos para los proveedores de servicios en la Web. Los motores de búsqueda, que millones de nosotros utilizamos cada día, dependen de esta tecnología; Google, eBay, Microsoft, Yahoo y otros, tienen varios centros de datos con clústeres de decenas de miles de procesadores. Claramente, el uso de múltiples procesadores en las compañías de servicios en Internet ha sido altamente exitoso.

Extensión: Otra forma de computación a gran escala es la *computación grid*, donde los computadores están distribuidos en áreas muy grandes y los programas que se están ejecutando deben comunicarse a través de redes de transporte de datos. La única forma de computación grid, y la más popular, fue promovida por el proyecto SETI@home. Se basa en la observación de que millones de PCs están inactivos en

algún momento sin hacer nada útil; estos PCs podrían ser recolectados y puestos a hacer algún trabajo útil, siempre y cuando se desarrollase un software que se ejecutase en los PCs y los configurase como una pieza independiente del problema a resolver. El primer ejemplo fue el proyecto *Search for ExtraTerrestrial Intelligence (SETI)*. Más de 5 millones de usuarios de computadores en más de 200 países se unieron a SETI@home y contribuyeron con 19 000 millones de horas de trabajo de los computadores. Al final de 2006, el grid SETI@home operaba a 257 TeraFLOPS.

Autoevaluación

1. Verdadero o falso: Al igual que los SMP, los sistemas de paso de mensajes utilizan bloqueos para la sincronización.
2. Verdadero o falso: A diferencia de los SMP, los sistemas de paso de mensajes necesitan varias copias del programa de procesamiento paralelo y del sistema operativo.

7.5 Ejecución multihilo en hardware

La **ejecución multihilo en hardware (*hardware multithreading*)** permite que varios hilos *(threads)* compartan las unidades funcionales del procesador y se ejecuten simultáneamente. Para facilitar la compartición de los recursos del procesador, cada hilo debe tener su propio estado; por ejemplo, cada hilo debería tener una copia del banco de registros y el PC. La memoria puede compartirse gracias a la memoria virtual, que ya incluye soporte para multiprogramación. Además, el hardware debe tener la capacidad de conmutar de forma rápida la ejecución entre diferentes hilos. En particular, el cambio de hilo debe ser mucho más rápido que el cambio de proceso, que típicamente necesita cientos o miles de ciclos del procesador, mientras que el cambio de hilo debe ser instantáneo.

Ejecución multihilo en hardware: aumento de la utilización del procesador mediante la conmutación a otro hilo cuando un hilo se para.

Hay dos alternativas para la ejecución multihilo en hardware. En la **ejecución multihilo de grano fino (*fine-grained multithreading*)** se cambia de hilo en cada instrucción, y como resultado se obtiene una ejecución entrelazada de varios hilos. Este entrelazado se hace habitualmente siguiendo un turno rotatorio entre los hilos, saltándose los hilos parados en ese momento. Para que la ejecución multihilo de grano fino sea práctica, el procesador debe ser capaz de cambiar de hilo en cada ciclo del reloj. Una ventaja clave de la ejecución multihilo de grano fino es su capacidad de ocultar las pérdidas de prestaciones debidas a paradas largas y cortas, porque cuando un hilo se para se pueden ejecutar instrucciones de otros hilos. Por el contrario, la principal desventaja es la pérdida de velocidad en la ejecución de un hilo individual, porque un hilo que esté listo para ejecutarse sin paradas se verá interrumpido por instrucciones de otros hilos.

Ejecución multihilo de grano fino: versión de la ejecución multihilo en hardware que conmuta entre hilos después de cada instrucción.

La **ejecución multihilo de grano grueso (*coarse-grained multithreading*)** se desarrolló como una alternativa a la ejecución multihilo de grano fino. La conmutación entre hilos se realiza sólo cuando hay paradas largas, por ejemplo, en un fallo de la cache del segundo nivel. Este cambio alivia el requisito de cambios de hilos sin coste y la probabilidad de hacer más lenta la ejecución de un hilo individual es mucho menor, porque solamente se ejecutan instrucciones de otros hilos cuando cuando se produce una parada larga en el hilo en cuestión. Sin embargo, la ejecución

Ejecución multihilo de grano grueso: versión de la ejecución multihilo en hardware que conmuta entre hilos sólo después de un suceso importante, por ejemplo un fallo de cache.

multihilo de grano grueso tiene un importante inconveniente: su limitada capacidad de reducir las pérdidas en las prestaciones debidas a paradas cortas. Esta limitación viene dada por el coste del llenado del pipeline. Como un procesador con ejecución multihilo de grano grueso captura instrucciones de un solo hilo, cuando se produce una parada el pipeline debe vaciarse y el nuevo hilo que comienza la ejecución debe volver a llenarlo antes de ser capaz de terminar instrucciones. Debido a este sobrecoste del llenado, la ejecución multihilo de grano grueso es más útil para reducir la penalización de las paradas de alto coste, donde el tiempo de llenado del pipeline es muy pequeño comparado con el tiempo de parada.

Ejecución multihilo simultánea: versión de la ejecución multihilo en hardware que reduce el coste de la ejecución multihilo mediante la utilización de los recursos de las microarquitecturas con planificación dinámica y emisión múltiple de instrucciones.

La **ejecución multihilo simultánea (*simultaneous multithreading, SMT*)** es una variante de la ejecución multihilo en hardware que utiliza los recursos de un procesador con planificación dinámica y emisión múltiple de instrucciones para explotar el paralelismo a nivel de hilo al mismo tiempo que explota el paralelismo a nivel de instrucción. El punto clave que ha motivado el desarrollo de los SMT es que los procesadores con emisión múltiple de instrucciones tienen a menudo un número mayor de unidades funcionales que las que puede utilizar de forma efectiva un único hilo. Además, con el renombrado de registros y la planificación dinámica, se puede emitir varias instrucciones de diferentes hilos sin preocuparnos de las dependencias entre ellas; las dependencias se resuelven gracias a la planificación dinámica.

Como se confía en la existencia de mecanismos dinámicos, los SMT no conmutan en cada ciclo, sino que están siempre ejecutando instrucciones de varios hilos, dejando que el hardware determine qué instrucciones se ejecutan y asocie los registros renombrados al hilo adecuado.

La figura 7.5 ilustra, de forma conceptual, las diferencias en la capacidad del procesador para explotar los recursos superescalares para las siguientes configuraciones. La parte superior muestra como se ejecutarían de forma independiente cuatro hilos en un procesador sin soporte para ejecución multihilo. La parte inferior muestra como se combinan los cuatro hilos para ejecutarse de forma más eficiente con las tres alternativas de ejecución multihilo:

- Procesador superescalar con ejecución multihilo de grano grueso
- Procesador superescalar con ejecución multihilo de grano fino
- Procesador superescalar con ejecución multihilo simultánea

En el procesador superescalar sin soporte para ejecución multihilo, el uso de las ranuras *(slot)* de emisión está limitado por la falta de paralelismo a nivel de instrucciones. Además, una parada larga, como por ejemplo un fallo en la cache de instrucciones, puede provocar que todo el procesador quede inactivo.

En un procesador superescalar con ejecución multihilo de grano grueso, las paradas largas se ocultan parcialmente conmutando a otro hilo que utilice los recursos del procesador. Aunque así se reduce el número de ciclos en los que el procesador está completamente inactivo, el sobrecoste de llenado todavía deja ciclos inactivos y las limitaciones del ILP hacen que no se usen todas las ranuras de emisión. En el caso de grano fino, el entrelazado de los hilos elimina la mayoría de los ciclos con las ranuras de emisión totalmente vacías. Sin embargo, como en

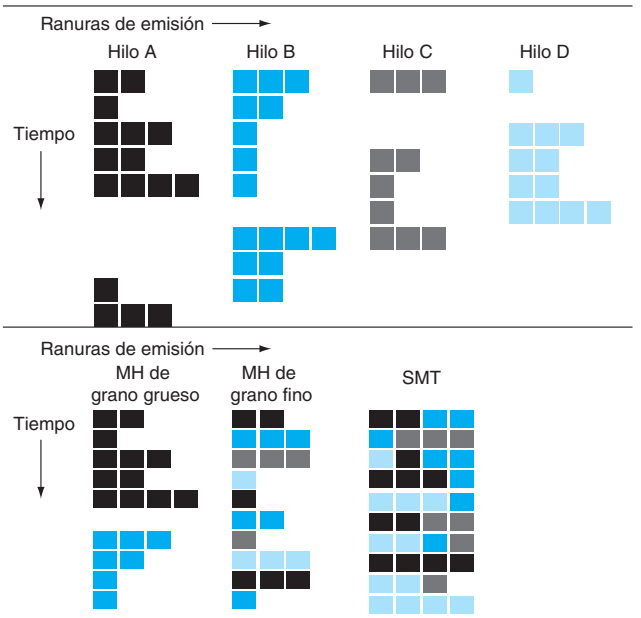


FIGURA 7.5 Diferentes alternativas sobre como cuatro hilos utilizan las ranuras de emisión de un procesador superescalar. En la parte superior se muestra como se ejecutaría cada hilo en un procesador superescalar estándar sin soporte para ejecución multihilo. En la parte inferior se muestra como los hilos se ejecutarían juntos en tres alternativas diferentes de ejecución multihilo. La dimensión horizontal representa la capacidad de emisión de instrucciones en cada ciclo. La dimensión vertical representa una secuencia de ciclos de reloj. Un cuadro vacío (blanco) indica que la ranura de emisión correspondiente está sin utilizar en ese ciclo de reloj. Los sombreados en gris y en color corresponden a los diferentes hilos en el procesador con ejecución multihilo. El efecto adicional de llenado de la ejecución multihilo de grano grueso, que no se muestra en la figura, podría causar pérdidas adicionales en la productividad.

cada ciclo sólo se emiten instrucciones de un hilo, todavía quedan ranuras vacías en algunos ciclos debido a las limitaciones del paralelismo a nivel de instrucción.

En los SMT se explota simultáneamente el paralelismo a nivel de hilo y el paralelismo a nivel de instrucciones, con varios hilos utilizando las ranuras de emisión en cada ciclo. Idealmente, la utilización de las ranuras de emisión está limitada por el desequilibrio entre las necesidades de recursos y la disponibilidad de recursos de los hilos. En la práctica, hay otros factores que pueden restringir la utilización de las ranuras. Aunque la figura 7.5 simplifica enormemente el funcionamiento real de estos procesadores, ilustra las ventajas potenciales sobre las prestaciones de la ejecución multihilo en general, y de los SMT en particular. Por ejemplo, el procesador multinúcleo Intel Nehalem soporta ejecución SMT con dos hilos para mejorar la utilización de los núcleos.

Concluimos con tres observaciones. Primero, por el capítulo 1 sabemos que el muro de la potencia está redirigiendo el diseño hacia procesadores más sencillos y más eficientes energéticamente. Los recursos infrautilizados de los procesadores con ejecución fuera de orden pueden reducirse y, entonces, se implementarán formas más sencillas de ejecución multihilo. Por ejemplo, el microprocesador Sun UltraSPARC T2 (Niagara 2) de la sección 7.11 es un ejemplo de vuelta a una microarquitectura más simple con ejecución multihilo de grano fino.

Segundo, un reto clave para las prestaciones es la tolerancia de las altas latencias debidas a fallos de cache. Los computadores con ejecución multihilo de grano fino como el UltraSPARC T2 conmutan a otro hilo cuando se produce un fallo de cache, lo que es probablemente más efectivo para ocultar la latencia de la memoria que intentar llenar las ranuras de emisión no usadas como en los SMT.

Una tercera observación es que la clave del éxito de la ejecución multihilo en hardware es la utilización más eficiente del hardware por la compartición de componentes entre diferentes tareas. Los diseños multinúcleo también comparten recursos, por ejemplo, dos procesadores pueden compartir una unidad de punto flotante o la cache L3. Esta compartición reduce algunos de los beneficios de la ejecución multihilo comparado con proporcionar más núcleos sin ejecución multihilo.

Autoevaluación

1. Verdadero o falso: Los sistemas con ejecución multihilo y los sistemas multinúcleo dependen del paralelismo para obtener mayor eficiencia.
2. Verdadero o falso: La ejecución multihilo simultánea utiliza hilos para mejorar la utilización de los recursos de un procesador con ejecución fuera de orden y planificación dinámica.

7.6 SISD, MIMD, SIMD, SPMD y procesamiento vectorial

Otra categorización de los computadores paralelos, propuesta en los años 1960, aún se utiliza hoy en día. Se basa en el número de flujos de instrucciones y de datos. La figura 7.6 muestra esta clasificación. Así, un monoprocesador convencional tiene un flujo de instrucciones y un flujo de datos, y en un multiprocesador hay varios flujos de instrucciones y varios flujos de datos. Estas dos categorías se abrevian por **SISD** (*Single Instruction Single Data streams*) y **MIMD** (*Multiple Instruction Multiple Data streams*), respectivamente.

SISD (*Single Instruction, Single Data streams*): un monoprocesador.

MIMD (*Multiple Instruction, Multiple Data streams*): un multiprocesador.

		Flujos de datos	
		Único (*single*)	**Varios (*multiple*)**
Flujos de instrucciones	Único *(single)*	SISD: Intel Pentium 4	SIMD: Instrucciones SSE del x86
	Varios *(multiple)*	MISD: Actualmente no hay ningún ejemplo	MIMD: Intel Xeon e5345 (Clovertown)

FIGURA 7.6 Categorización del hardware, con ejemplos, basada en el número de flujos de instrucciones y de datos: SISD, SIMD, MISD y MIMD.

Es posible escribir programas separados que se ejecuten en los diferentes procesadores de un computador MIMD y conseguir que colaboren para la consecución de un objetivo coordinado más ambicioso. Sin embargo, lo más habitual es que los programadores escriban un único programa que se ejecute en todos los procesadores del computador MIMD, introduciendo sentencias condicionales para conseguir que procesadores diferentes ejecuten secciones de código diferentes. Este estilo de programación se llama **Programa Único Datos Múltiples** (***Single Program Multiple Data, SPMD***), y es el estilo habitualmente utilizado en los computadores MIMD.

SPMD *(Single Program, Multiple Data stream)*: modelo de programación convencional de los MIMD, donde un único programa se ejecuta en todos los procesadores.

Si es difícil encontrar ejemplos de computadores útiles que puedan ser clasificados como MISD, lo contrario tiene mucho más sentido: los computadores **SIMD**, que operan sobre vectores de datos. Por ejemplo, una instrucción SIMD podría sumar 64 números enviando 64 flujos de datos a 64 ALUs para hacer 64 sumas en un único ciclo de reloj.

SIMD (*Single Instruction, Multiple Data streams*): un multiprocesador. La misma instrucción se ejecuta sobre varios flujos de datos, como en los procesadores vectoriales o los procesadores en array.

La principal virtud de los SIMD es que todas las unidades de ejecución paralelas están sincronizadas y responden a una única instrucción que procede de un único contador de programa (PC). Desde la perspectiva del programador, este estilo de programación es próximo al más familiar SISD. Aunque todas las unidades ejecutan la misma instrucción, cada unidad de ejecución tiene sus propios registros de direcciones y, por lo tanto, cada unidad tiene direcciones de datos diferentes. Así, en términos de la figura 7.1, una aplicación secuencial podría compilarse para ejecutarse en un hardware serie organizado como un SISD, o en un hardware paralelo organizado como un SIMD.

La motivación original de los SIMD fue amortizar el coste de la unidad de control en docenas de unidades de ejecución. Otra ventaja es el tamaño reducido de la memoria de programa —un SIMD necesita sólo una copia del código que se está ejecutando simultáneamente en varias unidades, mientras que los MIMD de paso de mensajes pueden necesitar una copia en cada procesador y los MIMD de memoria compartida necesitarán varias caches de instrucciones.

Los SIMD son eficientes cuando se trabaja con conjuntos de datos en lazos tipo *for*. Así, para extraer paralelismo en SIMD debe haber muchas estructuras de datos iguales, lo que se denomina **paralelismo a nivel de datos (*Data-level paralelism*)**. La mayor debilidad del SIMD se encuentra en las sentencias *case* o *switch*, donde cada unidad de ejecución tiene que hacer una operación diferente sobre sus datos, dependiendo de qué datos tiene. Las unidades de ejecución con los datos equivocados se deshabilitan para que las unidades con datos apropiados puedan continuar. Esencialmente, estas situaciones se ejecutan con factor $1/n$ en las prestaciones, siendo n el número de casos.

Paralelismo a nivel de datos: paralelismo que se alcanza al operar sobre datos independientes.

Los llamados *procesadores en array* que inspiraron la categoría de los SIMD se quedaron en el camino (véase **sección 7.14** en el CD), pero actualmente se mantienen activas dos interpretaciones de SIMD.

SIMD en x86: extensiones multimedia

La variación del SIMD más ampliamente utilizada se encuentra en casi todos los microprocesadores actuales y es la base de los cientos de instrucciones MMX y SSE de los microprocesadores x86 (véase capítulo 2). Estas instrucciones se añadieron para mejorar las prestaciones de los programas multimedia. Permiten que el hardware disponga de muchas ALUs que operan simultáneamente o, dicho de otra forma, permiten la partición de una ALU ancha en varias ALUs paralelas más estrechas que operan simultáneamente. Por ejemplo, podríamos ver un componente hardware como una ALU de 64 bits o dos ALUs de 32 bits o cuatro ALUs de 16 bits u ocho ALUs de 8 bits. El ancho de las cargas y almacenamientos es, simplemente, el de la ALU más ancha. De este modo, el programador puede utilizar la misma instrucción de transferencia de datos para transferir un dato de 64 bits o dos datos de 32 bits o cuatro datos de 16 bits u ocho datos de 8 bits.

Este paralelismo de bajo coste para datos enteros estrechos fue la motivación original de las instrucciones MMX de x86. Como resultado de mantener la ley de Moore, se añadió más hardware a estas extensiones multimedia y ahora la extensión SSE2 permite la ejecución simultánea de parejas de números en punto flotante de 64 bits.

El ancho de la operación y de los registros se codifica en el código de operación de las instrucciones multimedia. Al crecer el ancho de los datos de los registros y de las operaciones, se disparó el número de códigos de operación de las instrucciones multimedia y ahora hay cientos de instrucciones SSE (véase capítulo 2).

Procesamiento vectorial

Una interpretación más antigua y elegante del SIMD es la arquitectura vectorial, que ha estado muy identificada con Cray Computers. Otra vez, se ajusta a problemas con un gran paralelismo a nivel de datos. En lugar de tener 64 ALUs haciendo 64 sumas simultáneamente, como en los antiguos procesadores en arrays, en la arquitectura vectorial se segmenta la ALU para alcanzar buenas prestaciones a un coste más reducido. La filosofía básica de las arquitecturas vectoriales es recoger los datos de memoria, ponerlos ordenados en un gran conjunto de registros, operar sobre ellos secuencialmente en los registros y escribir los resultados en memoria. El elemento clave de la arquitectura vectorial es el conjunto de registros vectoriales. Así, una arquitectura vectorial podría tener 32 registros vectoriales con 64 datos de 64 bits cada uno.

EJEMPLO

Comparación entre código vectorial y código convencional

Supongamos que se extiende la arquitectura del repertorio de instrucciones MIPS con instrucciones vectoriales y registros vectoriales. Las operaciones vectoriales utilizan los mismos nombres que las operaciones MIPS pero añadiendo la letra "V". Por ejemplo, `addv.d` suma dos vectores de precisión doble. Las instrucciones vectoriales pueden tener como operandos dos registros vectoriales (`addv.d`) o un registro vectorial y un registro escalar (`addvs.d`). En este último caso, el valor en el registro escalar se usa como operando de todas las operaciones —la instrucción `addvs.d` suma el contenido de un registro escalar cada elemento de un registro vectorial. Las instrucciones de carga y almacenamiento vectorial se denotan `lv` y `sv` y cargan o almacenan un vector completo de datos en precisión doble. Un operando es el registro vectorial que se va a cargar o almacenar y el otro, un registro de propósito general del MIPS, contiene la dirección de comienzo del vector en memoria. A partir de esta breve descripción, veamos el código MIPS convencional y el código MIPS vectorial para la operación

$$Y = a \times X + Y$$

donde X e Y son vectores de números en punto flotante de precisión doble, que inicialmente residen en memoria, y a es una variable escalar de precisión doble. (Este ejemplo es llamado también lazo DAXPY y forma el lazo más interno del programa de prueba Linpack; DAXPY viene de double precision $a \times X$ plus Y). Supongamos que las direcciones de comienzo de X e Y están en `$s0` y `$s1`, respectivamente.

RESPUESTA

El código MIPS convencional para DAXPY es el siguiente:

```
      l.d     $f0,a($sp)      ;carga el escalar a
      addiu   r4,$s0,#512     ;límite superior de la carga
lazo: l.d     $f2,0($s0)      ;carga x(i)
      mul.d   $f2,$f2,$f0     ;a × x(i)
      l.d     $f4,0($s1)      ;carga y(i)
      add.d   $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d     $f4,0($s1)      ;almacena en y(i)
      addiu   $s0,$s0,#8      ;incrementa índice a x
      addiu   $s1,$s1,#8      ;incrementa índice a y
      subu    $t0,r4,$s0      ;calcula el límite
      bne     $t0,$zero,lazo  ;comprueba si terminado
```

El código MIPS vectorial para DAXPY es:

```
      l.d     $f0, a($sp)     ;carga el escalar a
      lv      $v1, 0($s0)     ;carga vector x
      mulvs.d $v2, $v1, $f0   ;multiplicación vector-
                               escalar
      lv      $v3, 0($s1)     ;carga vector y
      addv.d  $v4, $v2, $v3   ;suma del vector y al
                               producto
      sv      $v4, 0($s1)     ;almacena el resultado
```

Hay algunas comparaciones interesantes entre los segmentos de código del ejemplo. La más llamativa es que el procesador vectorial reduce de forma significativa el ancho de banda dinámico de las instrucciones, ejecutando sólo 6 instrucciones frente a casi 600 del MIPS. Esta reducción se produce porque las operaciones vectoriales operan sobre 64 datos y porque las instrucciones de control del lazo, que son casi la mitad de las instrucciones en el lazo MIPS, no son necesarias en el código vectorial. Como es de esperar, esta reducción en el número de instrucciones capturadas y ejecutadas contribuye al ahorro de energía.

Otra diferencia importante es la frecuencia de los riesgos (capítulo 4). En el código MIPS cada instrucción `add.d` debe esperar por el resultado de `mult.d`, y cada `s.d` debe esperar por el resultado de `add.d`. En el procesador vectorial, cada instrucción vectorial debe parar sólo en el primer elemento del vector y los elementos siguientes fluirán suavemente en el pipeline. Así, sólo hay una parada por cada operación vectorial, en lugar de una por cada elemento del vector. En este ejemplo, la frecuencia de paradas en MIPS es aproximadamente 64 veces mayor que en VMIPS. Las paradas en el pipeline MIPS pueden reducirse con el desenrollamiento de lazos (véase capítulo 4). Sin embargo, la gran diferencia en el ancho de banda de instrucciones no puede reducirse.

Extensión: El lazo en el ejemplo anterior se ajusta perfectamente a la longitud del vector de la arquitectura. Cuando los lazos son más cortos, las arquitecturas vectoriales utilizan un registro que reduce la longitud de las operaciones vectoriales. Cuando los lazos son más largos, se itera sobre la longitud total del vector y se añade código para manejar los datos restantes. Este proceso se llama *seccionamiento (strip mining).*

Vectorial frente a escalar

Las instrucciones vectoriales tienen propiedades interesantes comparadas con las arquitecturas del repertorio de instrucciones convencionales, que en este contexto llamaremos arquitecturas escalares:

- Una instrucción vectorial especifica una gran cantidad de trabajo; es equivalente a ejecutar un lazo entero. Por lo tanto, las necesidades de captura y decodificación de instrucciones se reducen significativamente.
- Al utilizar una instrucción vectorial, el compilador o el programador indica que el cálculo de cada resultado del vector es independiente del cálculo de otros resultados en el mismo vector, de este modo no es necesario que el hardware compruebe las dependencias de datos dentro de una instrucción vectorial.
- Es más sencillo escribir aplicaciones eficientes con paralelismo a nivel de datos con arquitecturas y compiladores vectoriales que con multiprocesadores MIMD.
- Sólo hay que comprobar dependencias de datos entre dos instrucciones vectoriales una vez por operando, en lugar de una vez para cada elemento del vector. La reducción de las comprobaciones también ayuda a reducir el consumo de potencia.
- Las instrucciones vectoriales que acceden a memoria tienen un patrón de acceso conocido. Si los elementos del vector son adyacentes, la captura del vector es sencilla en un sistema de memoria compuesto por bancos fuertemente entrelazados. Así, el coste de la latencia de acceso a memoria para el vector completo es equivalente a un solo acceso a memoria, en lugar de un acceso para cada elemento del vector.
- Como una instrucción vectorial con un comportamiento predeterminado reemplaza un lazo completo, no existen los riesgos de control debidos al salto del final del lazo.
- Los ahorros en ancho de banda de las instrucciones y comprobación de riesgos, unidos al uso eficiente del ancho de banda de memoria, dan a las arquitecturas vectoriales ventajas en potencia y energía frente a las arquitecturas escalares.

Por estas razones, las operaciones vectoriales son más rápidas que una secuencia de operaciones escalares sobre el mismo número de datos y los diseñadores son proclives a introducir unidades vectoriales si el dominio de la aplicación las va a usar frecuentemente.

Vectorial frente extensiones multimedia

Al igual que las extensiones multimedia que se encuentran en las instrucciones SSE del x86, una instrucción vectorial especifica varias operaciones. Sin embargo, típicamente las extensiones multimedia especifican unas pocas operaciones, mientras que las instrucciones vectoriales especifican docenas de operaciones. A diferencia de las extensiones multimedia, el número de elementos en una instrucción vectorial no está en el código de operación, sino en un registro separado. Esto significa que versiones diferentes de la arquitectura vectorial pueden implementarse con un número diferente de elementos cambiando únicamente el contenido de este registro y manteniendo así la compatibilidad binaria. Por el contrario, cada vez que en la arquitectura de la extensión multimedia del x86 cambia la longitud del "vector" hay que añadir un nuevo conjunto de códigos de operación.

También a diferencia de las extensiones multimedia, la transferencia de datos no tiene porqué ser contigua. Las arquitecturas vectoriales soportan accesos con separaciones *(stride)*, en los que el hardware carga uno de cada n elementos en memoria, y accesos indexados, en los que el hardware obtiene las direcciones de los elementos que deben ser cargados en un registro vectorial.

Como en las extensiones multimedia, la arquitectura vectorial dispone de flexibilidad en el ancho de los datos. Así, es fácil hacer operaciones sobre 32 datos de 64 bits o 64 datos de 32 bits o 128 datos de 16 bits o 256 datos de 8 bits.

En general, las arquitecturas vectoriales son muy eficientes en la ejecución de programas de procesamiento paralelo con paralelismo a nivel de datos; se ajustan mejor a la tecnología de los compiladores y evolucionan más fácilmente a lo largo del tiempo que las extensiones multimedia de la arquitectura x86.

Autoevaluación

Verdadero o falso: Como se pone de manifiesto en la arquitectura x86, las extensiones multimedia pueden considerarse arquitecturas vectoriales con vectores cortos que soportan únicamente transferencias secuenciales de datos vectoriales.

Extensión: A la vista de las ventajas de las arquitecturas vectoriales, ¿por qué no son más populares fuera del ámbito de la computación de altas prestaciones? Ha habido varios problemas relacionados con los estados más complejos para los registros vectoriales, el aumento del tiempo necesario para realizar un cambio de contexto y la dificultad para procesar los fallos de página en cargas y almacenamientos vectoriales; además las instrucciones SIMD tenían algunos de los beneficios de las instrucciones vectoriales. Sin embargo, Intel ha hecho recientemente algunos anuncios que sugieren que las arquitecturas vectoriales pueden jugar todavía un papel relevante. El *Advanced Vector Instruction (AVI)* de Intel, anunciado para el año 2010, expande el ancho de los registros SSE de 128 a 256 bits de forma inmediata y permite expansiones hasta 1024 bits. Este último ancho es equivalente a 16 número punto flotante de precisión doble. No está claro aún si habrá cargas y almacenamientos vectoriales. Además, la arquitectura con la que Intel se introducirá en el mercado de las GPU en 2010 —el nombre del producto es "Larrabee"— tendrá instrucciones vectoriales.

Extensión: Otra ventaja de las arquitecturas multimedia y las extensiones vectoriales es la relativa facilidad para extender una arquitectura del repertorio de instrucciones escalar con esas instrucciones para mejorar las prestaciones de las operaciones con paralelismo a nivel de datos.

7.7 Introducción a las unidades de procesamiento gráfico

Una justificación importante para añadir instrucciones SIMD a las arquitecturas existentes fue que muchos microprocesadores de PCs y estaciones de trabajo estaban conectados a monitores gráficos, y una parte cada vez más importante del tiempo de procesamiento se empleaba en los gráficos. Por este motivo, al aumentar, tal como se indica en la ley de Moore, el número de transistores disponibles en los microprocesadores, se utilizaron en parte para mejorar el procesamiento de los gráficos.

Este aumento de los recursos disponibles derivado de la ley de Moore permitió, además de mejorar el procesamiento de los gráficos por parte de la CPU, añadir funciones en los chips controladores de gráficos y vídeo para acelerar los gráficos 2D y 3D. Además, en el segmento de los productos de alta calidad, facilitó que añadieran a las estaciones de trabajo tarjetas gráficas caras, típicamente tarjetas de Silicon Graphics, para crear imágenes de calidad fotográfica. Estas tarjetas de alta calidad se utilizaron para la creación de imágenes por computador que más tarde encontraron su hueco en los anuncios de televisión y las películas. Así, los controladores gráficos y de vídeo tenían un objetivo claro para absorber el aumento de los recursos computacionales; recursos que se prestaban desde el ámbito de los supercomputadores a los microprocesadores en la búsqueda de mayores prestaciones.

La industria de los juegos por computador, tanto en PCs como en consolas específicas como la Sony PlayStation, fue una fuerza motriz decisiva para la mejora del procesamiento gráfico. El rápido crecimiento de esta industria animó a muchas empresas a hacer fuertes inversiones para el desarrollo de hardware gráfico cada vez más rápido; esta realimentación positiva tuvo como consecuencia que el procesamiento gráfico mejorarse a una velocidad mayor que el procesamiento de propósito general de los microprocesadores.

Como la comunidad de gráficos y juegos tiene objetivos diferentes de los de la comunidad de desarrollo de microprocesadores, el procesamiento gráfico evolucionó con un estilo propio de procesamiento y terminología. A medida que los procesadores gráficos aumentaban en potencia, se ganaron el nombre de *Unidades de Procesamiento Gráfico (Graphics Processing Units)* o *GPUs*, para distinguirse de las CPUs. Estas son algunas de las características diferenciales de las GPUs frente a las CPUs:

- Las GPUs son aceleradores que complementan a la CPU, es decir, las GPUs no están capacitadas para hacer todas las funciones de la CPU y este papel les permite dedicar todos sus recursos al procesamiento gráfico. No importa que la GPU no pueda hacer ciertas tareas, porque en un sistema con una GPU y una CPU, la CPU puede hacer todo lo que la GPU no puede. Así, la combinación CPU-GPU es un ejemplo de *multiprocesamiento heterogéneo*, donde los procesadores no tienen por qué ser iguales. (Otro ejemplo es la arquitectura IBM Cell, que se describe en la sección 7.11, que fue también diseñada para acelerar el procesamiento gráfico 2D y 3D).
- La interfaz de programación de las GPUs son las interfaces de programación de aplicaciones (*application programming interfaces*, API) de alto nivel, como por

ejemplo OpenGL y Microsoft DirectX, asociados con los lenguajes de sombreado gráfico de alto nivel, tales como *C for Graphics* (Cg) de NVIDIA y *High Level Shader Language* (HLSL) de Microsoft. Los compiladores tienen como objetivo obtener una traducción a lenguajes intermedios estándar para la industria en lugar de a instrucciones máquina, y el software controlador de la GPU genera instrucciones máquina optimizadas específicas para la GPU. Mientras estas APIs y lenguajes evolucionan rápidamente para adoptar los nuevos recursos hardware disponibles por la Ley de Moore, la libertad en la compatibilidad de las instrucciones binarias facilita que los diseñadores de GPUs exploren nuevas arquitecturas sin el temor a quedar marcados para siempre por la implementación de experimentos fallidos. Este ambiente lleva a innovaciones más rápidas en las GPUs que en las CPUs.

- El procesamiento gráfico consiste en dibujar los vértices de primitivas geométricas 3D, tales como líneas y triángulos, y en hacer el *sombreado* o la renderización *(rendering)* de fragmentos de píxeles de las primitivas geométricas. Por ejemplo, los vídeo juegos dibujan 20 a 30 veces más píxeles que vértices.
- Cada vértice se puede dibujar de forma independiente y cada fragmento de píxeles se puede renderizar de forma independiente. Para renderizar de forma rápida millones de píxeles por cuadro *(frame)*, la GPU ha evolucionado para ejecutar en paralelo muchos hilos de programas de sombreado de vértices y píxeles.
- Los tipos de datos de gráficos son vértices, que constan de las coordenadas (x, y, z, w), y píxeles, que constan de los componentes de color (rojo, verde, azul, alfa). (Véase el apéndice A para más detalles sobre vértices y píxeles.) Cada componente del vértice se representa como un número punto flotante de 32 bits. Originalmente, cada uno de los cuatro componentes del píxel era un entero sin signo de 8 bits, pero desde hace poco se representan como números punto flotante de precisión simple entre 0.0 y 1.0.
- El conjunto de trabajo puede estar formado por cientos de megabytes y no muestra la misma localidad temporal que los datos en la aplicaciones de propósito general. Además, hay mucho paralelismo a nivel de datos.

Estas diferencias han llevado a dos estilos arquitecturales diferentes:

- Posiblemente la mayor diferencia radica en que la GPU, a diferencia de la CPU, no se apoya en las caches multinivel para ocultar la elevada latencia de los accesos a memoria. En su lugar, la GPU oculta esta latencia con la utilización de múltiples hilos; es decir, entre que se produce una solicitud de acceso a memoria hasta que llega el dato, la GPU ejecuta cientos o miles de hilos independientes de la solicitud.
- Las GPUs utilizan paralelismo extensivo para obtener altas prestaciones, implementando muchos procesadores paralelos y muchos hilos concurrentes.
- La memoria principal de la GPU se orienta hacia el ancho de banda en lugar de hacia la latencia. Incluso hay chips DRAM para GPU más anchos y con un mayor ancho de banda que los chips DRAM para CPUs. Además, la memoria de las GPUs es tradicionalmente de menor capacidad que la memoria de los

microprocesadores; por ejemplo, en 2008 las GPUs típicamente tenían 1 GB o menos de memoria, mientras que las CPUs tenían entre 2 y 32 GB. Finalmente, no hay que olvidar que en computación de propósito general hay que tener en cuenta el tiempo de transferencia de datos entre la memoria de la CPU y la de la GPU porque, al fin y al cabo, la GPU es un coprocesador.

- Dada la dependencia de la GPU con la ejecución de muchos hilos para obtener un buen ancho de banda de memoria, las GPUs pueden alojar muchos procesadores paralelos y muchos hilos. En conclusión, los procesadores GPU utilizan ejecución multihilo a gran escala.
- En el pasado, para obtener las prestaciones necesarias en las aplicaciones gráficas, la GPU estaba compuesta por varios procesadores heterogéneos de propósito específico. En la actualidad, están afianzando la utilización de procesadores de propósito general idénticos para tener mayor flexibilidad en la programación, de forma que se parecen cada vez más a los diseños multinúcleo de la computación dominante.
- Dada la naturaleza de los datos utilizados en gráficos, datos formados por cuatro componentes, históricamente las GPUs han implementado instrucciones SIMD, al igual que las CPUs. Sin embargo, las GPUs recientes se están centrando en la utilización de instrucciones escalares para mejorar la programabilidad y la eficiencia.
- A diferencia de las CPUS, no han incorporado soporte para aritmética de punto flotante de precisión doble, porque no se ha necesitado en las aplicaciones de computación gráfica. En 2008 se anunció la primera GPU con soporte de precisión doble en hardware. Sin embargo, las operaciones de precisión simple son aún entre ocho y diez veces más rápidas que las de precisión doble, incluso en esas nuevas GPUs, mientras que en las CPUs la diferencia en prestaciones está limitada a los beneficios derivados de transferir menos bytes con memoria.

Aunque las GPUs fueron diseñadas para un conjunto limitado de aplicaciones, algunos programadores se preguntaban si podrían reescribir sus aplicaciones de forma que pudiesen explotar las altas prestaciones potenciales de la GPU. Este estilo de programación se llama *GPU de Propósito General (General Purpose GPU)* o *GPGPU*. Tras varios intentos para especificar sus aplicaciones utilizando las APIs gráficas y los lenguajes de sombreado de gráficos, se desarrollaron lenguajes de programación basados en C que permiten escribir directamente los programas para la GPU. Un ejemplo es Brook, un lenguaje de *streaming* para GPUs. El siguiente paso en la programabilidad del hardware y del lenguaje de programación es CUDA (Compute Unified Device Architecture) de NVIDIA, que permite que el programador escriba programas C que se van a ejecutar en la GPU, aunque con algunas restricciones. La mayor programabilidad de la GPU está haciendo crecer su utilización en computación paralela.

Una introducción a la arquitectura de una GPU de NVIDIA

El apéndice A profundiza en las GPUs y describe en detalle la arquitectura de la GPU más reciente de NVIDIA, llamada Tesla. Como las GPU evolucionaron en un ambiente propio, no solamente tienen arquitecturas diferentes, como ya se ha mencionado, sino

que tienen también una terminología diferente. Una vez que se aprende la terminología de las GPUs es más fácil ver las similitudes con las propuestas presentadas en secciones anteriores, tales como la ejecución multihilo de grano fino y el procesamiento vectorial.

Para ayudar en la transición al nuevo vocabulario, hacemos una rápida introducción a los términos e ideas de la arquitectura de la GPU Tesla y el entorno de programación CUDA.

El chip de la GPU está alojado en una tarjeta que se conecta al PC con una conexión PCI-Express. La llamada placa base de la GPU se integra en el conjunto de chips de la placa base, como un puente norte o un puente sur (capítulo 6).

Las GPUs se ofertan generalmente como una familia de chips con diferentes relaciones precio-prestaciones y con compatibilidad software. Los chips basados en la arquitectura Tesla se ofertan con un número de nodos que varía entre 1 y 16; NVIDIA llama a estos nodos *multiprocesadores*. A comienzos del 2008, la versión más potente era la GeForce 8800 GTX, con 16 multiprocesadores y una frecuencia de reloj de 1.35 GHz. Cada multiprocesador permite ejecución multihilo y está formado por 8 unidades de punto flotante de simple precisión y de procesamiento de enteros, que NVIDIA llama *streaming processors*.

Como la arquitectura incluye una instrucción de multiplicación-suma de punto flotante de precisión simple, las prestaciones pico de multiplicación-suma de precisión simple del chip 8800 GTX son:

$$16\ \text{Mps} \times \frac{8\ \text{SPs}}{\text{MP}} \times \frac{2\ \text{FLOPs/instr}}{\text{SP}} \times \frac{1\ \text{instr}}{\text{ciclo}} \times \frac{1.35 \times 10^9\ \text{ciclos}}{\text{segundo}}$$

$$= \frac{16 \times 8 \times 2 \times 1.35\ \text{GFLOPs}}{\text{segundo}}$$

$$= \frac{345.6\ \text{GFLOPs}}{\text{segundo}}$$

Cada uno de los 16 multiprocesadores del GeForce 8800 GTX tiene un almacenamiento local controlado por software, con una capacidad de 16 KB, y 8192 registros de 32 bits. El sistema de memoria consta de seis particiones de la DRAM 900 MHz Graphics DDR3, cada una con un ancho de 8 bytes y una capacidad de 128 MB. Así, el tamaño total de la memoria es 768 MB. El ancho de banda pico de la GDDR3 es:

$$6 \times \frac{8\ \text{bytes}}{\text{transferencia}} \times \frac{2\ \text{transferencias}}{\text{ciclo}} \times \frac{0.9 \times 10^9\ \text{ciclos}}{\text{segundo}} = \frac{6 \times 8 \times 2 \times 0.9\ \text{GB}}{\text{segundo}} = \frac{86.4\ \text{GB}}{\text{segundo}}$$

Para ocultar la latencia de la memoria, los procesadores de *streaming* tienen soporte hardware para ejecución multihilo. Cada grupo de 32 hilos se llama *trama* (*warp*). La trama es la unidad de planificación y los hilos activos en una trama —hasta 32— se ejecutan en paralelo con un estilo SIMD. Sin embargo, la arquitectura multihilo se enfrenta a los saltos condicionales permitiendo que los hilos vayan por diferentes salidas del salto. Cuando los hilos de una trama toman vías de salida divergentes, se ejecutan secuencialmente los códigos en ambas vías, con algunos hilos inactivos, haciendo así que algunos hilos se ejecuten más lentamente. Tan pronto como terminan las vías condicionales, el hardware junta los hilos en una trama completamente activa. Para obtener las mejores prestaciones, los 32 hilos de una trama deben ejecutarse en para-

lelo. De forma similar, para aumentar las prestaciones de la memoria, el hardware examina los flujos de direcciones que provienen de los diferentes hilos e intenta mezclar las solicitudes individuales para formar una pocas transferencias de bloques de memoria, pero de mayor tamaño.

En la figura 7.7 se combinan todas estas características y se compara un multiprocesador Tesla con un núcleo del Sun UltraSPARC T2, que se describe en las secciones 7.5 y 7.11. Ambos sistemas incluyen soporte hardware para ejecución multihilo. En el eje vertical se muestran los hilos planificados. Cada multiprocesador Tesla consta de ocho procesadores de *streaming,* que ejecutan ocho hilos paralelos, tal como se muestra en el eje horizontal. Como se ha mencionado anteriormente, las mejores prestaciones se obtienen cuando los 32 hilos de una trama se ejecutan a la vez con un estilo SIMD, que en la arquitectura Tesla se denomina instrucción única, múltiples hilos (*Single-Instruction Multiple Thread*, SIMT). Con el paradigma SIMT se determina de forma dinámica qué hilos de la trama pueden ejecutar una misma instrucción simultáneamente y qué hilos independientes están inactivos en cada ciclo. Por otra parte, el núcleo T2 consta de un único procesador con ejecución multihilo; de modo que en cada ciclo ejecuta una instrucción de un hilo.

El multiprocesador Tesla utiliza ejecución multihilo de grano fino para planificar las 24 tramas, que se muestran verticalmente en bloques de cuatro ciclos de reloj. De forma similar, el UltraSPARC T2 planifica ocho hilos, mostrados verticalmente, a razón de un hilo por ciclo. Así, del mismo modo que el T2 conmuta entre hilos para mantener el núcleo ocupado, el Tesla conmuta entre tramas para mantener el multi-

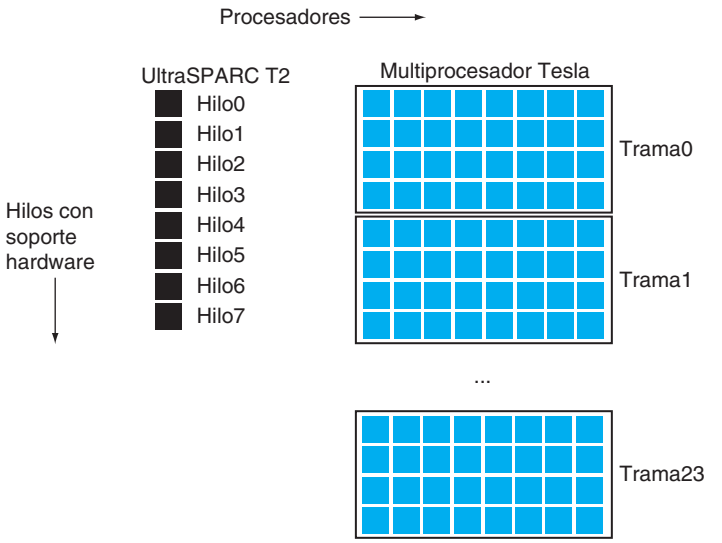


FIGURA 7.7 Comparación de un núcleo del Sun UltraSPARC T2 (Niagara 2) y un multiprocesador Tesla. El núcleo T2 en un procesador con soporte hardware para ejecución multihilo con ocho hilos. El multiprocesador Tesla consta de ocho procesadores de *streaming* y soporta ejecución multhilo con 24 tramas de 32 hilos (ocho procesadores por cuatro ciclos de reloj). El núcleo T2 cambia de hilo en cada ciclo de reloj, mientras que el Tesla conmuta sólo cada dos o cuatro ciclos. El T2 sólo implementa la ejecución multihilo en el tiempo, mientras que el Tesla implementa la ejecución multihilo en el tiempo y en el espacio, es decir, a lo largo de los ocho procesadores de *streaming* y en segmentos de 4 ciclos de reloj.

procesador ocupado. La principal diferencia radica en que el núcleo T2 tiene un procesador que conmuta hilos en cada ciclo, mientras que el Tesla conmuta tramas cada dos ciclos, como mínimo, en los ocho núcleos de *streaming*. Como la arquitectura Tesla tiene como objetivo programas con un elevado paralelismo a nivel de datos, los diseñadores consideraron que conmutar cada dos o cuatro ciclos en lugar de conmutar cada ciclo no tiene gran influencia en las prestaciones y, por el contrario, restringiendo la frecuencia de conmutación se obtiene un hardware mucho más simple.

El entorno de programación CUDA utiliza también una terminología propia. Un programa CUDA es un programa C/C++ para un sistema heterogéneo con CPU y GPU. Se ejecuta en la CPU y envía trabajos paralelos a la GPU. Cada trabajo consta de una transferencia de datos desde la memoria principal y de la *emisión de un hilo*. Un hilo es una parte del programa para la GPU. Los programadores especifican el número de hilos en el *bloque de hilos* y el número de bloques de hilos que quieren ejecutar en la GPU. El motivo por el que los programadores deben preocuparse de los bloques de hilos es que los hilos del bloque se planifican para su ejecución en el mismo multiprocesador y deben compartir la misma memoria local. De este modo, se comunican con instrucciones de carga y almacenamiento en lugar de utilizar mensajes. El compilador CUDA asigna registros a cada hilo, con la limitación de que el número de registros por hilo multiplicado por el número de hilos por bloque no puede exceder los 8192 registros por multiprocesador.

Un bloque de hilos puede tener hasta 512 hilos y cada grupo de 32 hilos en un bloque se empaqueta en una trama. Los bloques grandes obtienen mejores prestaciones que los bloques pequeños (el tamaño mínimo del bloque es un hilo). Como ya hemos mencionado anteriormente, bloques y tramas con menos de 32 hilos son menos eficientes que los bloques y tramas completos.

Un planificador hardware trata de planificar varios bloques de hilos en cada multiprocesador. Si lo consigue, divide dinámicamente el almacenamiento local de 16 KB entre los diferentes bloques de hilos.

Una perspectiva de las GPUs

Las GPUs como la arquitectura NVIDIA Tesla no encajan claramente en ninguna de las clasificaciones de los computadores existentes, como por ejemplo, la clasificación de la figura 7.6 en la página 648. Claramente, el GeForce 8800 GTX, con 16 multiprocesadores Tesla, es un MIMD. La cuestión es como clasificar un multiprocesador Tesla y sus ocho procesadores de *streaming*.

Recordemos que anteriormente se dijo que el estilo SIMD funciona perfectamente con lazos y tiene su punto débil en las sentencias *case* y *switch*. Tesla obtiene altas prestaciones explotando el paralelismo a nivel de datos y, además, facilita la programación de aplicaciones con paralelismo a nivel de hilos. Tesla permite que el programador vea el multiprocesador como un sistema MIMD de ocho procesadores de *streaming* con ejecución multihilo, pero cuando se pueden ejecutar a la vez varios hilos de la misma trama, el hardware intenta asociar los ocho procesadores de *streaming* para trabajar en un estilo SIMT. Sin embargo, cuando los hilos tienen que operar de forma independiente y seguir caminos de ejecución independientes, se ejecutarán de forma más lenta porque los 32 hilos de la trama comparten una única unidad de captura de instrucciones. Si los 32 hilos de la trama estuvieran ejecutando instrucciones independientes,

cada hilo operaría a un 1/16 de las prestaciones pico de la trama completa de 32 hilos que se ejecutan en ocho procesadores de *streaming* en cuatro ciclos de reloj.

Así, cada hilo independiente tiene su propio PC, de modo que los programadores pueden ver al Tesla como un MIMD, pero deben preocuparse de escribir sentencias de control que permitan que el hardware SIMT ejecute los programas CUDA en un estilo SIMD para alcanzar las prestaciones deseadas.

A diferencia de las arquitecturas vectoriales, que necesitan un compilador vectorizador para reconocer el paralelismo a nivel de datos en tiempo de compilación y generar instrucciones vectoriales, la implementación hardware de la arquitectura Tesla detecta el paralelismo a nivel de datos entre hilos en tiempo de ejecución. Por lo tanto, las GPU Tesla no necesitan compiladores vectorizadores y facilitan el manejo por parte del programador de la partes del programa que no presentan paralelismo de datos. Para tener una perspectiva de esta alternativa, la figura 7.8 sitúa las GPUs en una clasificación en base al paralelismo a nivel de instrucciones frente al paralelismo a nivel de datos y a si este paralelismo se detecta en tiempo de compilación o de ejecución. Esta clasificación es una muestra de que la GPU Tesla está empezando un nuevo camino en la arquitectura de computadores.

	Estático: detectado en tiempo de compilación	Dinámico: detectado en tiempo de ejecución
Paralelismo a nivel de instrucciones	VLIW	Superescalar
Paralelismo a nivel de datos	SIMD o vectorial	Multiprocesador Tesla

FIGURA 7.8 Clasificación hardware de las arquitecturas de los procesadores y ejemplos basada en estático frente a dinámico y ILP frente a DLP.

Autoevaluación Verdadero o falso: Las GPUs utilizan chips DRAM gráficos para reducir la latencia de la memoria e incrementar así las prestaciones de las aplicaciones gráficas.

7.8 Introducción a las topologías de redes para multiprocesadores

Los chips multinúcleo necesitan disponer de redes en el chip para la conexión entre los núcleos. En esta sección revisamos los pros y contras de diferentes redes para multiprocesadores.

El coste de la red depende del número de conmutadores, el número de conexiones con la red en un conmutador, el ancho (número de bits) de la conexión y la longitud de la conexión una vez que la red se ha implementado en el chip. Por ejemplo, algunos núcleos pueden ser adyacentes y otros pueden estar situados bastante alejados en el chip. Las prestaciones de la red también son multifacéticas; incluyen la latencia de envío y recepción de mensajes en una red sin carga, la productividad en términos del número máximo de mensajes que pueden transmitirse en un tiempo dado, los retrasos causados por los conflictos en alguna parte de la red y las prestaciones variables en función del patrón de comunicación. Por otra parte, como se puede imponer que los sistemas deban funcionar aún en presencia de componentes

estropeados, otro requisito de la red podría ser la tolerancia a fallos. Finalmente, en esta era de chips limitados por la potencia, la eficiencia energética de las diferentes organizaciones también es un factor a tener en cuenta.

Las redes normalmente se dibujan como grafos, representando las conexiones con los arcos del grafo, los nodos procesador-memoria con cuadrados negros y los conmutadores con círculos coloreados. En esta sección consideramos que todos las conexiones son bidireccionales; es decir, la información puede fluir en ambas direcciones. Todas las redes constan de *conmutadores* conectados a los nodos procesador-memoria y a otros conmutadores. La red más sencilla, que constituye la primera mejora de un bus estándar, es una red que conecta una secuencia de nodos:

Esta topología se llama *anillo (ring)*. Como algunos nodos no están directamente conectados, los mensajes tendrán que pasar por varios nodos intermedios antes de llegar a su destino final. A diferencia de los buses, un anillo puede realizar varias transferencias simultáneas.

Como hay muchas topologías de red donde elegir, se necesitan métricas de prestaciones para distinguir estos diseños. Hay dos muy populares. La primera es el **ancho de banda de red**, que es el ancho de banda de cada conexión multiplicado por el número de conexiones. Esta métrica representa el caso más favorable. Para un anillo con P procesadores, el ancho de banda de red sería P multiplicado por el ancho de banda de la conexión; el ancho de banda de red de un bus es simplemente el ancho de banda del bus o dos veces el ancho de banda de la conexión.

Ancho de banda de red: de manera informal, el ritmo de transferencia pico de una red; se puede referir a la velocidad de una conexión o al ritmo de transferencia colectivo de todas las conexiones de la red.

Para equilibrar este caso optimista, se incluye otra métrica que está próxima al peor caso: el **ancho de banda de bisección**. Se calcula dividiendo la máquina en dos partes, cada una con la mitad de los nodos, y sumando el ancho de banda de las conexiones que cruzan esta línea divisoria imaginaria. El ancho de banda de bisección de un anillo es dos veces el ancho de banda de la conexión y el ancho de banda de bisección de un bus es igual al ancho de banda de la conexión. Si la conexión es igual de rápida que el bus, el anillo es sólo el doble de rápido que el bus en el peor caso y P veces más rápido en el mejor caso.

Ancho de banda de bisección: ancho de banda entre dos partes iguales de un multiprocesador. Esta métrica es para el peor caso de la división del multiprocesador.

Algunas topologías de red no son simétricas, entonces ¿dónde se traza la línea imaginaria que bisecciona la máquina? Como esta métrica representa el peor caso, se debe escoger la división que lleva a las prestaciones de la red más pesimistas. Dicho en otras palabras, se deben calcular todos los posibles anchos de banda de bisección y escoger el más pequeño. Adoptamos esta visión pesimista porque a menudo los programas paralelos están limitados por la conexión más débil en la cadena de comunicación.

En el otro extremo, frente al anillo, encontramos la **red completamente conectada**, en la que cada procesador tiene una conexión bidireccional con los restantes procesadores. En este caso, el ancho de banda de red es $P \times (P - 1)/2$ y el ancho de banda de bisección es $(P/2)^2$.

Red completamente conectada: red que conecta nodos procesador-memoria proporcionando conexiones dedicadas entre los nodos.

La mejora en las prestaciones que se obtiene con las redes completamente conectadas se ve compensada por su elevado coste. Este alto coste motivó el desarrollo de nuevas topologías red con un coste cercano al coste del anillo y unas prestaciones parecidas

a las prestaciones de las redes completamente conectadas. El éxito de una topología depende en gran medida de la naturaleza de las comunicaciones en los programas paralelos que constituyen carga de trabajo que se está ejecutando en el computador.

Es difícil contar las topologías diferentes que se han propuesto y discutido en la bibliografía, pero sólo un puñado de ellas se han utilizado en procesadores paralelos comerciales. La figura 7.9 muestra dos topología populares. A menudo, en los computadores reales se añaden conexiones adicionales a estas topologías para mejorar las prestaciones y la fiabilidad.

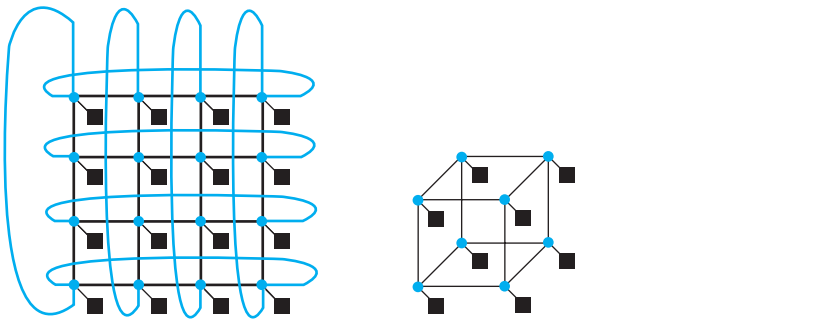
Alternativamente, en lugar de tener un procesador en cada nodo de la red, puede haber nodos con únicamente un conmutador. Los conmutadores son más pequeños que los nodos procesador-memoria-conmutador y por lo tanto se pueden empaquetar con mayor densidad, disminuyendo la distancia y aumentando las prestaciones. Estas redes se denominan **redes multietapa** para reflejar el hecho de que un mensaje debe viajar a través de varias etapas. La figura 7.10 muestra dos de las organizaciones multietapa más populares. Una red completamente conectada o una **red crossbar** permite que cualquier nodo se comunique con cualquier otro nodo en una etapa. Una *red Omega* necesita menos hardware que el crossbar ($2\ n \log_2 n$ frente a n^2 conmutadores) pero, dependiendo del patrón de comunicación, puede haber conflictos entre mensajes. Por ejemplo, la red Omega de la figura 7.10 no puede enviar un mensaje desde P_0 a P_6 al mismo tiempo que envía un mensaje desde P_1 a P_7.

Red multietapa: red con un conmutador sencillo en cada nodo.

Red crossbar: red que permite que cualquier nodo se comunique con cualquier otro nodo en una etapa.

Implementación de las topología de red

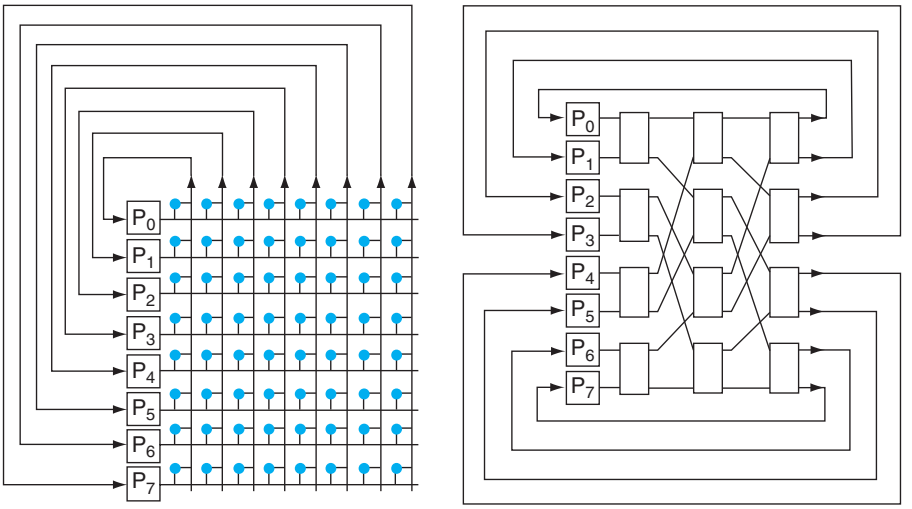
Este análisis sencillo de las redes ignora algunas consideraciones prácticas importantes en la implementación de una red. La distancia de cada conexión afecta al coste de las comunicaciones a frecuencias de reloj elevadas —generalmente, cuanto mayor sea la distancia más caro es operar con frecuencias de reloj elevadas. Por otra parte, cuando las distancias son más cortas es más fácil asociar más cables



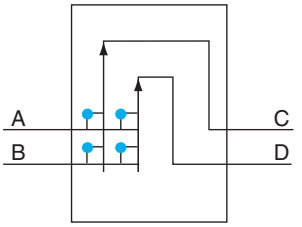
a. malla 2-D o malla de 16 nodos

b. *n*-cubo o cubo de dimensión *n* de 8 nodos ($8 = 2^3$ por lo tanto $n = 3$)

FIGURA 7.9 Topologías de red que se han utilizado en procesadores paralelos comerciales. Los círculos coloreados representa conmutadores y los cuadrados negros representan nodos procesador-memoria. Aún cuando un conmutador tiene muchas conexiones, generalmente una sola va al procesador. La topología *n*-cubo booleana es una interconexión *n*-dimensional con 2^n nodos, que necesita *n* conexiones por conmutador (más una para el procesador) y tiene por lo tanto *n* vecinos próximos. Frecuentemente, estas topologías básicas han sido complementadas con arcos adicionales para mejorar las prestaciones y la fiabilidad.



a. Crossbar b. Red Omega



c. Caja de conmutación de la red Omega

FIGURA 7.10 Topologías de red multietapa populares de 8 nodos. Los conmutadores en estas topologías son más sencillos que en topologías anteriores porque las conexiones son unidireccionales; los datos llegan por la parte inferior y salen por la conexión de la derecha. El conmutador de la figura c puede pasar A a C y B a D o B a C y A a D. El crossbar utiliza n^2 conmutadores, siendo n el número de nodos, mientras que la red Omega utiliza $2n \log_2 n$ cajas de conmutación, compuestas a su vez por cuatro conmutadores. En este caso, el crossbar tiene 64 conmutadores frente a 12 cajas de conmutación, o 48 conmutadores, de la red Omega. Sin embargo, el crossbar puede soportar cualquier combinación de mensajes entre procesadores, y la red Omega no.

a la conexión, porque la potencia necesaria para alimentar muchos cables en un chip es menor si estos son cortos. Cables más cortos son también más baratos que cables más largos. Otra limitación práctica es que la topología tridimensional debe proyectarse en un medio bidimensional como los chips. La preocupación final es la potencia; por ejemplo, los problemas de consumo de potencia pueden forzar la utilización de topologías sencillas en malla en los microprocesadores multinúcleo. El resultado final es que topologías que parecen elegantes cuando se dibujan en la pizarra pueden ser impracticables al implementarlas en silicio.

7.9 Programas de prueba para multiprocesadores

Como se ha visto en el capítulo 1, la evaluación de un sistema mediante la utilización de programas de prueba *(benchmark)* es siempre un asunto sensible, porque es una forma visible de determinar qué sistema es mejor. Los resultados afectan no sólo a las ventas de sistemas comerciales, sino también a la reputación de los diseñadores de los sistemas. Así, todos los participantes quieren ganar la competición, pero también quieren estar seguros de que si es otro el que gana, ha merecido ganar porque su sistema es realmente el mejor. Este deseo conduce a asegurar que los resultados de la evaluación no son simplemente trampas para estos programas de prueba, sino avances que contribuyen a mejorar las prestaciones de las aplicaciones reales.

Una regla básica para evitar las trampas es que no se pueden cambiar los programas de prueba. El código fuente y el conjunto de datos son fijos y sólo hay una respuesta correcta. Cualquier desviación de estas reglas invalida los resultados.

Muchos programas de prueba para multiprocesadores siguen estas normas, aunque una excepción aceptada es permitir aumentar el tamaño del problema para que se puedan ejecutar los programas de prueba en sistemas con diferente número de procesadores. Es decir, se permite, en muchos casos, un escalado débil en lugar de un escalado fuerte, aunque se debe ser cuidadoso al comparar los resultados de programas ejecutados con tamaños de problema diferentes.

En la figura 7.11 se resumen varios programas de prueba paralelos, que se describen también a continuación:

- *Linpack* es un conjunto de rutinas de álgebra lineal, y las rutinas para realizar la eliminación gaussiana constituyen lo que se denomina programa de prueba *(benchmark)* Linpack. La función DAXPY utilizada en el ejemplo de la página 650 es una pequeña parte del código fuente del programa de prueba Linpack, pero es la culpable de la mayor parte del tiempo de ejecución. Permite escalado débil, dejando que el usuario seleccione cualquier tamaño de problema. Además, permite reescribir la rutina en cualquier lenguaje y forma, mientras se calcule el resultado correcto. Dos veces al año se hacen públicos los 500 computadores con las prestaciones más rápidas en la ejecución de Linpack (*www. top500.org*). El primero de la lista se considera el computador más rápido del mundo.

- *SPECrate* es una métrica de prestaciones basada en los programas de prueba SPEC CPU, por ejemplo el SPEC CPU 2006 (véase capítulo 1). En lugar de determinar las prestaciones de programas individuales, SPECrate ejecuta simultáneamente muchas copias de los programas y, como no hay comunicación entre trabajos, mide el paralelismo a nivel de trabajos. Se pueden ejecutar tantas copias de los programas como se desee, por lo tanto es también una forma de escalado débil.

Programa de prueba	¿Escalado?	¿Reprogramación?	Descripción
Linpack	Débil	Sí	Álgebra lineal matricial densa [Dongarra, 1979]
SPECrate	Débil	No	Paralelismo de trabajos independientes [Hening, 2007]
Stanford Parallel Application for Shared Memory, SPLASH2 [Woo et al. 1995]	Fuerte (aunque tiene dos conjuntos de datos)	No	FFT 1D compleja Descomposición LU por bloques Factorización de Cholesky dispersa por bloques Ordenación radix entero Barnes-Hut Multipolo rápido adaptativo Simulación del océano Radiosidad jerárquica Trazado de rayos (*ray tracing*) Renderizado de volúmenes Simulación de agua con estructura de datos espacial Simulación de agua sin estructura de datos espacial
Programas de prueba paralelos NAS [Bailey et al., 1991]	Débil	Sí (sólo C o Fortran)	EP: vergonzosamente paralelo MG: multimalla simplificado CG: malla no estructurada para el método del gradiente conjugado FT: solución de ecuaciones en derivadas parciales 3-D con FFT IS: ordenación de grandes conjuntos de enteros
Conjunto de programas de prueba PARSEC [Bienia et al., 2008]	Débil	No	Blackscholes—Valor de acciones con ecuaciones en derivadas parciales Backscholes Bodytrack—Seguimiento del cuerpo de una persona Canneal—Algoritmo Simulated annealing para optimización de interconexionado Dedup—Compresión de nueva generación con deduplicación de datos Facesim—Simula los movimientos del rostro humano Ferret—Servidor de búsqueda de similaridad de contenidos Fluidanimate—Dinámica de fluidos para animación con el método SPH Freqmine—Minería de datos Streamcluster—Agrupamiento de un flujo de entrada Swaptions—Valor de una carpeta de acciones Vips—Procesamiento de imágenes x264—Codificación de vídeo H.264
Patrones de Diseño de Berkeley [Asanovic et al. 2006]	Fuerte o débil	Sí	Máquina de estado finitos Lógica combinacional Recorrido de grafos Malla estructurada Matriz densa Matriz dispersa Métodos espectrales (FFT) Programación dinámica N-cuerpos MapReduce Árboles Vuelta atrás (*backtrack*)/Ramificación y acotación (*branch and bound*) Modelo de inferencia gráfico Malla no estructurada

FIGURA 7.11 Ejemplos de programas de prueba paralelos.

- *SPLASH* y *SPLASH2 (Stanford Parallel Application for Shared Memory)* son el resultado de los trabajos llevados a cabo en Stanford en la década de los 90 para obtener un conjunto de programas de prueba paralelos con objetivos similares a los programas del SPEC CPU. Incluyen núcleos computacionales y aplicaciones, muchos de ellos provenientes de la comunidad de la computación de altas prestaciones. Requiere escalado fuerte aunque proporciona dos conjuntos de datos.

- Los programas de pruebas paralelos *NAS (NASA Advanced Supercomputing)* fueron otro intento de la década de los 90 para la evaluación de multiprocesadores. Constan de cinco núcleos computacionales de dinámica de fluidos y permiten escalado débil mediante la definición de varios conjuntos de datos. Al igual que Linpack, se pueden reescribir los programas pero siempre en C o Fortran.

- El conjunto de programas de prueba más reciente, *PARSEC (Princeton Application Repository for Shared Memory Computers),* consta de programas con ejecución multihilo que utilizan **Pthreads** (hilos POSIX) y **OpenMP** (*Open MultiProcessing*). Se centran en mercados emergentes y constan de nueve aplicaciones y tres núcleos computacionales. Ocho explotan el paralelismo de datos, tres el paralelismo segmentado y uno el paralelismo no estructurado.

Pthreads: API de UNIX para crear y manipular hilos. Incluida en una biblioteca.

OpenMP: API escrita en C, C++ o Fortran para multiprocesadores de memoria compartida que se ejecuta en plataformas UNIX y Microsoft. Incluye directivas del compilador, una biblioteca y directivas de tiempo de ejecución.

El inconveniente de estas restricciones tradicionales en los programas de prueba es que la innovación se ve limitada a la arquitectura y al compilador. A menudo no pueden utilizarse estructuras de datos mejoradas, algoritmos, lenguajes de programación, etc., porque darían resultados engañosos. Un sistema podría ser el vencedor debido, por ejemplo, al algoritmo en lugar de al hardware o al compilador.

Aunque estas normas son comprensibles cuando los fundamentos de la computación son relativamente estables, como ocurría en los años 1990 y en la primera mitad de esta década, no son convenientes al comienzo de una revolución. Para que esta revolución tenga éxito, es necesario facilitar las innovaciones a todos los niveles.

Recientemente investigadores de la Universidad de California en Berkeley han recomendado una nueva alternativa. Identificaron 13 patrones de diseño que, según ellos, serán parte de las aplicaciones del futuro, y se implementaron con entornos o núcleos computacionales. Algunos ejemplos son las matrices dispersas, mallas estructuradas, máquinas de estados finitos y recorrido de grafos. Manteniendo las definiciones a un nivel alto, esperan alentar innovaciones en cualquier nivel del sistema. Así, el sistema más rápido en la resolución de matrices dispersas podría utilizar cualquier estructura de datos, algoritmo, lenguaje de programación, además de nuevas arquitecturas y compiladores. La figura 7.11 muestra ejemplos de estos programas de prueba.

Autoevaluación

Verdadero o falso: El principal inconveniente de los programas de prueba para computadores paralelos es que la reglas establecidas para asegurar la imparcialidad eliminan también la innovación.

7.10 Roofline: un modelo de prestaciones sencillo

Esta sección se basa en un artículo de Williams y Patterson [2008]. En los últimos años, la cultura establecida en arquitectura de computadores llevó a diseños similares en todos los microprocesadores. Casi todos los servidores y computadores de sobremesa utilizaban caches, segmentación, emisión de instrucciones superescalar, predicción de saltos y ejecución fuera de orden. Los repertorios de instrucciones variaban, pero todos los microprocesadores habían salido de la misma escuela de diseño.

Es muy probable que el cambio a sistemas multinúcleo introduzca una mayor diversidad en los microprocesadores, porque no hay una sabiduría convencional que establezca para qué arquitectura es más fácil escribir programas de procesamiento paralelo correctos que se ejecuten eficientemente y escalen con el número de núcleos. Además, a medida que aumenta el número de núcleos por chip, cada fabricante probablemente ofrecerá un número diferente de núcleos por chips a diferentes precios.

Dada esta diversidad creciente, disponer de un modelo sencillo que ofrezca información sobre las prestaciones de diferentes diseños será especialmente útil. No es necesario ser perfectos, solamente se necesita algo intuitivo.

El modelo de las 3Cs del capítulo 5 es una buena analogía. No es un modelo perfecto, porque ignora factores potencialmente importantes como el tamaño de bloque, la política de asignación de bloques y la política de reemplazo de bloques. Además, tiene singularidades. Por ejemplo, un fallo en un diseño puede asociarse a la capacidad y en otra cache del mismo tamaño a conflictos. Aún así, este modelo ha sido popular durante los últimos 20 años, porque ofrece una indicación sobre la conducta de los programas, ayudando a los arquitectos y programadores a mejorar sus diseños en base a la información proporcionada por este modelo.

Para encontrar un modelo de este tipo, tomaremos como punto de partida los 13 patrones de diseño de Berkeley mostrados la figura 7.11. La idea de estos patrones de diseño es que las prestaciones de una aplicación concreta son realmente una suma ponderada de varios núcleos computacionales que implementan estos patrones. Evaluaremos los núcleos computacionales individuales, pero tendremos en mente que las aplicaciones reales son una combinación de muchos núcleos computacionales.

Aunque hay versiones con diferentes tipos de datos, la representación de punto flotante es habitual en varias implementaciones. Así, las prestaciones pico de punto flotante son un límite para la velocidad de los núcleos computacionales en un computador dado. Para sistemas multinúcleo, las prestaciones pico de punto flotante son las prestaciones pico colectivas de todos los núcleos del sistema. Si el sistema tiene varios microprocesadores, debemos multiplicar el pico por chip por el número de chips.

Las exigencias sobre el sistema de memoria pueden estimarse dividiendo esta prestación pico de punto flotante entre el número medio de operaciones punto flotante por byte accedido:

$$\frac{\text{Operaciones punto flotante/seg}}{\text{Operaciones punto flotante/byte}} = \text{Bytes/seg}$$

Intensidad aritmética: relación entre el número de operaciones punto flotante de un programa y el número de bytes de datos accedidos en memoria principal.

Esta relación de operaciones punto flotante por byte de memoria accedido se denomina **intensidad aritmética**, y se puede calcular dividiendo el número total de operaciones de punto flotante de un programa entre el número de bytes de datos transferidos a la memoria principal durante la ejecución. La figura 7.12 muestra la intensidad aritmética de varios patrones de diseño de Berkeley de la figura 7.11.

El modelo Roofline

El modelo sencillo propuesto une en un único grafo bidimensional las prestaciones de punto flotante, la intensidad aritmética y las prestaciones de la memoria [Williams, Patterson, 2008]. Las prestaciones de punto flotante pueden obtenerse utilizando las especificaciones hardware mencionadas anteriormente. El conjunto de trabajo de los núcleos computacionales que se van a considerar en este modelo no cabe en las caches integradas en el mismo circuito integrado del procesador *(on-chip caches)* de este modo, los niveles de la jerarquía de memoria detrás de las caches definen las prestaciones de la memoria. (Véase la *Extensión* en la página 473 del capítulo 5.)

El modelo, que se aplica a un computador en lugar de a cada núcleo computacional, se muestra en la figura 7.13. En el eje Y se representan las prestaciones de punto flotante alcanzables, entre 0.5 y 64.0 GFLOPS/segundo, y en el eje X la intensidad aritmética, que varía entre 1/8 y 16 FLOPS/bytes DRAM accedidos. Observe que se está utilizando una escala logarítmica.

Para un núcleo computacional dado, podemos encontrar un punto en el eje X basado en su intensidad aritmética. Dibujando una línea vertical en este punto, las prestaciones del núcleo computacional en este computador deben estar en alguna parte de esta línea. Podemos dibujar una línea horizontal mostrando las prestaciones de punto flotante pico del computador y obviamente, las prestaciones de punto flotante reales no pueden sobrepasar la línea horizontal, que representa el límite hardware.

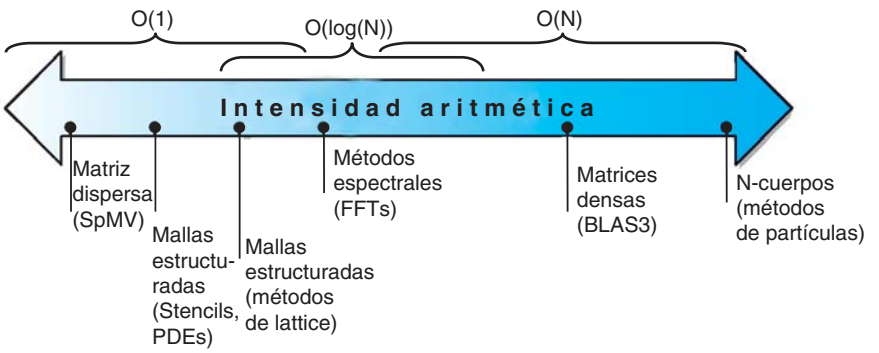


FIGURA 7.12 Intensidad aritmética, especificada como el número de operaciones de punto flotante de un programa dividido por el número de bytes accedidos en la memoria principal [William, Patterson, 2008]. La intensidad aritmética de algunos núcleos computacionales escala con el tamaño del problema, por ejemplo las matrices densas, pero otros núcleos computacionales tienen una intensidad aritmética independiente del tamaño del problema. El escalado débil para los núcleos computacionales de este último tipo puede conducir a resultados diferentes, porque la memoria tiene mucha menor presión.

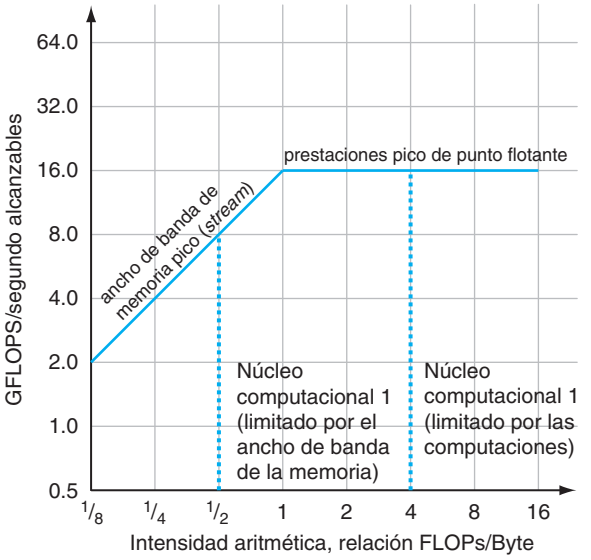


FIGURA 7.13 Modelo Roofline [Williams, Patterson, 2008]. Este ejemplo tiene unas prestaciones pico de punto flotante de 16 GFLOPS/seg y un ancho de banda de memoria pico de 16 GB/seg con el programa de prueba Stream. (Como Stream realmente da cuatro medidas, esta línea es la media de las cuatro.) La línea punteada vertical de la izquierda representa el núcleo computacional 1, que tiene una intensidad aritmética de 0.5 GFLOPs/byte. En este Opteron X2 el ancho de banda de la memoria está limitado a menos de 8 GFLOPs/seg. La línea punteada vertical de la derecha representa el núcleo computacional 2, que tiene una intensidad aritmética de 4 GFLOPs/byte. Está limitado por la computación, a 16 GFLOPs/seg. (Datos basados en el AMD Opteron X2 (revisión F) con dos núcleos a 2 GHz en un sistema de ranura dual.)

¿Cómo se representan las prestaciones de memoria pico? Como el eje X indica FLOPS/byte y el eje Y FLOPs/seg es, simplemente, la diagonal con un ángulo de 45 grados. De este modo, podemos dibujar una tercera línea para las prestaciones máximas de punto flotante que el sistema de memoria del computador puede soportar para una intensidad aritmética dada. Los límites se pueden expresar con la siguiente ecuación:

$$\text{GFLOPS/seg alcanzables} = \text{Min}(\text{Ancho de banda de memoria pico} \times \text{intensidad aritmética}, \text{prestaciones de punto flotante pico})$$

Las líneas horizontal y diagonal dan el nombre al modelo (línea de tejado) e indican su valor. El modelo fija un límite superior en las prestaciones de un núcleo computacional dependiendo de su intensidad aritmética. Si pensamos en la intensidad aritmética como un poste que sostiene un tejado, o bien sostiene la parte plana del tejado, lo que significa que las prestaciones están limitadas por las computaciones (núcleo computacional 2 de la figura 7.13), o bien sostiene la parte inclinada, lo que significa que las prestaciones están limitadas por el ancho de banda de la memoria (núcleo computacional 1 de la figura 7.13). Una vez obtenida la representación de un computador con el modelo Roofline, se puede utilizar repetidamente porque la representación no varía al cambiar los núcleos computacionales.

Observe que el "punto cresta", donde se unen los tejados diagonal y horizontal, proporciona una interesante perspectiva del computador. Si este punto está muy a la derecha en el eje X, entonces sólo los núcleos computacionales con una intensidad aritmética elevada alcanzan las prestaciones máximas del computador. Por el contrario, si está hacia la izquierda, casi todos los núcleos computacionales pueden llegar a las prestaciones máximas. Pronto veremos algunos ejemplos.

Comparación de dos generaciones de procesadores Opteron

El AMD Opteron X4 (Barcelona) con cuatro núcleos es el sucesor del Opteron X2 de dos núcleos. Para simplificar el diseño de la tarjeta, utilizan la misma ranura. En consecuencia, tienen los mismos canales de DRAM y por lo tanto el mismo ancho de banda de memoria pico. Además de duplicar el número de núcleos, el Opteron X4 también ha duplicado las prestaciones pico de punto flotante por núcleo: el Opteron X4 puede emitir dos instrucciones de punto flotante SSE2 en cada ciclo de reloj mientras que el Opteron X2 sólo puede emitir una. Como los dos sistemas que estamos comparando tienen frecuencias de reloj similares (2.2 GHz en el Opteron X2 y 2.3 GHz en el Opteron X4) el Opteron X4 tiene unas prestaciones pico de punto flotante más de cuatro veces mayores que las del Opteron X2 con el mismo ancho de banda de DRAM. Por otra parte, el Opteron X4 tiene una cache L2 de 2 MB que no existe en el Opteron X2.

La figura 7.14 compara el modelo Roofline de ambos sistemas. Como era de esperar, el punto cresta se mueve de 1 en el Opteron X2 a 5 en el Opteron X4. Así, para obtener mejores prestaciones en la siguiente generación, los núcleos computacionales deben tener una intensidad aritmética mayor que 1 o sus conjuntos de trabajo deben caber en las caches del Opteron X4.

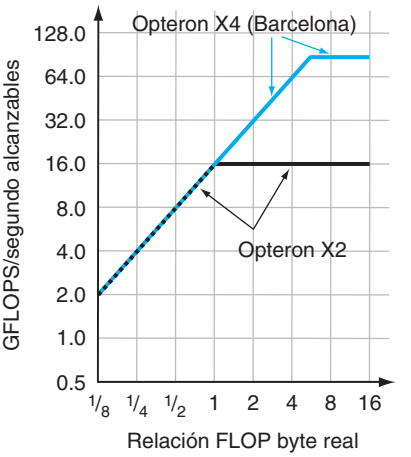


FIGURA 7.14 Modelo Roofline de dos generaciones de Opteron. El Opteron X2, que es el mismo de la figura 7.11, se representa en negro y el Opteron X4 en color. El Opteron X4 tiene un punto cresta mayor, lo que significa que núcleos computacionales limitados por las computaciones en el Opteron X2 podrían estar limitados por las prestaciones de la memoria en el Opteron X4.

El modelo Roofline proporciona un límite superior para las prestaciones. Suponga que su programa está lejos de este límite. ¿Qué optimizaciones se deberían introducir y en qué orden?

Las dos optimizaciones siguientes pueden ayudar a reducir los cuellos de botella computacionales en cualquier núcleo:

1. *Combinación de operaciones punto flotante*. Las prestaciones pico de punto flotante de un computador se alcanzan con un número similar de sumas y multiplicaciones que se ejecutan casi simultáneamente. Este equilibrio es necesario, bien porque el computador soporta instrucciones multiplicación-suma (véase la *Extensión* en la página 268 del capítulo 3) o bien porque la unidad de punto flotante dispone de igual número de sumadores y multiplicadores de punto flotante. Para obtener las mejores prestaciones es necesario también que parte importante de las instrucciones sean operaciones de punto flotante y no instrucciones de enteros.

2. *Mejora del paralelismo a nivel de instrucción y utilización del SIMD*. En las arquitecturas superescalares las prestaciones más elevadas se obtienen cuando se capturan, ejecutan y terminan tres o cuatro instrucciones por ciclo (véase el capítulo 4). El objetivo entonces debe ser mejorar el código proporcionado por el compilador para aumentar el ILP. Una forma de conseguirlo es utilizando desenrollamiento de lazos. En las arquitecturas x86, una instrucción SIMD puede operar sobre parejas de operandos de precisión doble, por lo tanto deben utilizarse cuando sea posible.

Las dos optimizaciones siguientes pueden ayudar a reducir los cuellos de botella de memoria:

1. *Prebúsqueda software*. Habitualmente, para obtener las mayores prestaciones es necesario mantener en ejecución un elevado número de operaciones de memoria, lo que se puede conseguir con mayor facilidad si hace prebúsqueda software de instrucciones en lugar de esperar a que el dato sea demandado por la computación.

2. *Afinidad de memoria*. La mayoría de los procesadores actuales incluyen un controlador de memoria en el mismo chip. Si el sistema tiene varios chips, algunas direcciones harán referencia a la DRAM localizadas en el chip y otras tendrán que acceder, a través de interconexiones entre chips, a DRAMs localizadas en otros chips. En este caso, las prestaciones disminuyen. El objetivo de esta optimización es situar los hilos y los datos sobre los que operan en el mismo par procesador-memoria, para que el procesador sólo tenga que acceder en muy contadas ocasiones a memorias en otros chips.

El modelo Roofline puede ayudar a decidir cuáles de estas optimizaciones se implementan y en qué orden. Podemos pensar en estas optimizaciones como un "techo" debajo de la línea de tejado, de modo que no se puede atravesar el techo sin implementar la optimización correspondiente.

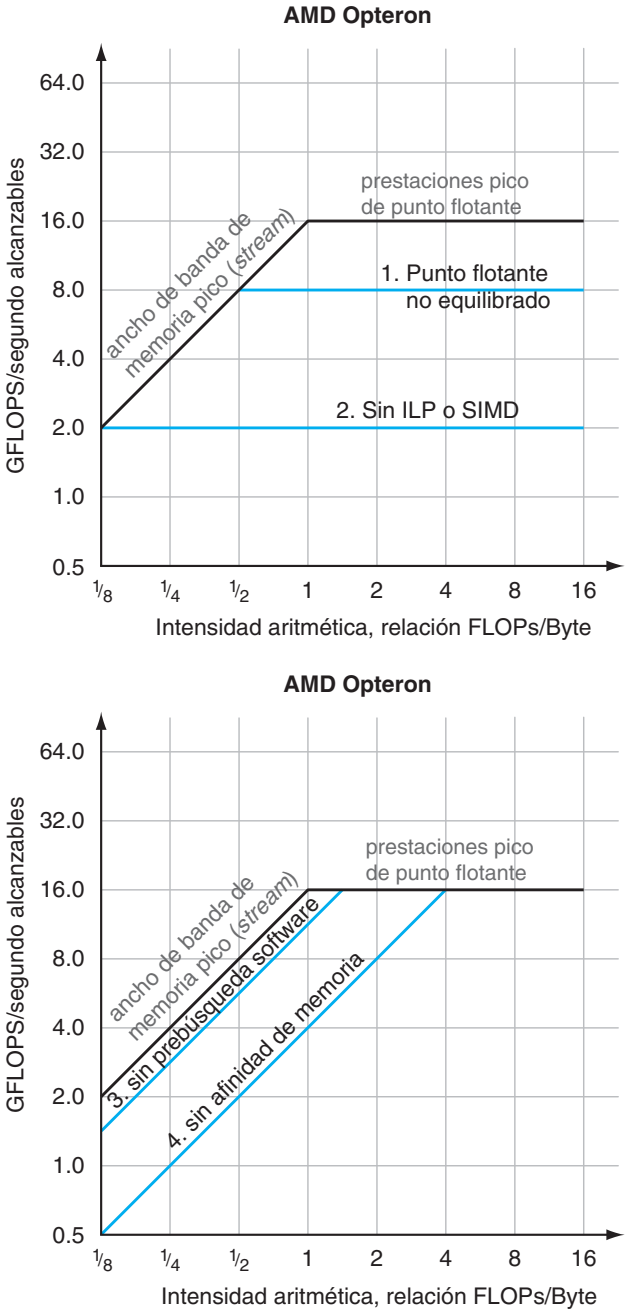


FIGURA 7.15 Modelo Roofline con techos. El gráfico de la parte superior muestra "techos" computacionales de 8 GFLOPs/seg si la combinación de operaciones punto flotante está desequilibrada y de 2 GFLOPs/seg si no están implementadas las optimizaciones para aumentar el ILP y el SIMD. El gráfico de la parte inferior muestra "techos" de ancho de banda de memoria de 11 GB/seg y de 4.8 GB/seg si no están implementadas las optimizaciones de prebúsqueda software y afinidad de memoria, respectivamente.

La línea del tejado computacional puede obtenerse de los manuales, y la línea del tejado de memoria ejecutando programas de prueba. Los techos computacionales, por ejemplo el equilibrio punto flotante, también vienen en los manuales del computador; pero los techos de memoria requieren que se ejecuten varios experimentos en cada computador para ver las diferencias entre ellos. Sin embargo, sólo hay que hacerlo una vez para cada computador, porque una vez que se han caracterizado los techos de un computador, se pueden utilizar en cualquier momento y por cualquiera para priorizar las optimizaciones.

En la figura 7.15 se añaden los techos al modelo Roofline de la figura 7.13, con el techo computacional en la gráfica superior y el techo del ancho de banda de la memoria en la gráfica inferior. Aunque no se indica explícitamente en la figura, los techos más altos corresponden a las optimizaciones; para atravesar los techos más elevados es necesario romper antes los techos inferiores.

El premio por la introducción de una optimización es el ancho de la separación entre un techo y siguiente límite. Así, la figura 7.15 sugiere que la optimización 2, mejora del ILP, afecta de forma muy significativa a la computación en este computador y la optimización 4, mejora de la afinidad de la memoria, beneficia considerablemente al ancho de banda de la memoria.

La combinación de los techos de la figura 7.15 en un único grafo se muestra en la figura 7.16. La intensidad aritmética de un núcleo computacional determina la región de optimización, y sugiere las optimizaciones que merece la pena intentar. Observe que, para varias intensidades aritméticas, las optimizaciones computacionales y las del ancho de banda de la memoria se solapan. En la figura 7.16 se han sombreado de forma diferente tres regiones para indicar diferentes estrategias de optimización. Por ejemplo, el núcleo computacional 2 cae en el trapezoide azul de la derecha, lo que sugiere la conveniencia de aplicar únicamente optimizaciones computacionales. El núcleo computacional 1 cae en el paralelogramo azul-gris del medio, lo que sugiere que es conveniente aplicar optimizaciones de ambos tipos y, además, que es preferible comenzar con las optimizaciones 2 y 4. Observe que las líneas verticales del núcleo computacional 1 están por debajo de la optimización de equilibrado de las operaciones de punto flotante por lo que la optimización 1 no es necesaria. Si un núcleo computacional estuviese en el triángulo gris de la parte inferior izquierda, entonces las únicas optimizaciones útiles serían las optimizaciones de memoria.

Hasta ahora hemos asumido que la intensidad aritmética es fija pero esto no es lo que ocurre en la realidad. En primer lugar, hay núcleos computacionales en los que la intensidad aritmética aumenta con el tamaño del problema, como ocurre en Matriz Densa y N-cuerpos (véase figura 7.12). Efectivamente, esto puede ser una razón que explique por qué los programadores tienen más éxito con el escalado débil que con el escalado fuerte. En segundo lugar, las caches influyen en el número de accesos a la memoria por lo que las optimizaciones que mejoran las prestaciones de las caches también mejoran la intensidad aritmética. Un ejemplo es la utilización del desenrollamiento de lazos y el agrupamiento de sentencias con direcciones similares para mejorar la localidad temporal. Muchos computadores tienen instrucciones especiales de cache para situar datos que se van a sobreescribir en poco tiempo en la cache sin leerlos antes de memoria. Ambas optimizaciones reducen el tráfico de la memoria y mueven el palo de la intensidad aritmética

a la derecha, por ejemplo, en un factor 1.5. Este desplazamiento puede situar al núcleo computacional en una región de optimización diferente.

En la siguiente sección utilizamos el modelo Roofline para mostrar las diferencias entre cuatro microprocesadores multinúcleo recientes con dos núcleos computacionales de aplicaciones reales. Si los ejemplos anteriores han mostrado el modo de usar el modelo para ayudar a los programadores a mejorar las prestaciones, también puede ser utilizado por los arquitectos para decidir donde introducir optimizaciones hardware para mejorar las prestaciones de la ejecución de los núcleos computacionales que van a ser importantes.

Extensión: Los techos están ordenados de forma que los más bajos son los más fáciles de optimizar. Claramente, un programador puede aplicar las optimizaciones en cualquier orden, pero siguiendo esta secuencia se reducen los esfuerzos baldíos en optimizaciones que no tienen beneficios debido a otros condicionantes. Al igual que en el modelo de las tres C, como el modelo Roofline se centra en la visión a alto nivel, pueden aparecer singularidades. Por ejemplo, el modelo supone que la carga está equilibrada entre todos los procesadores.

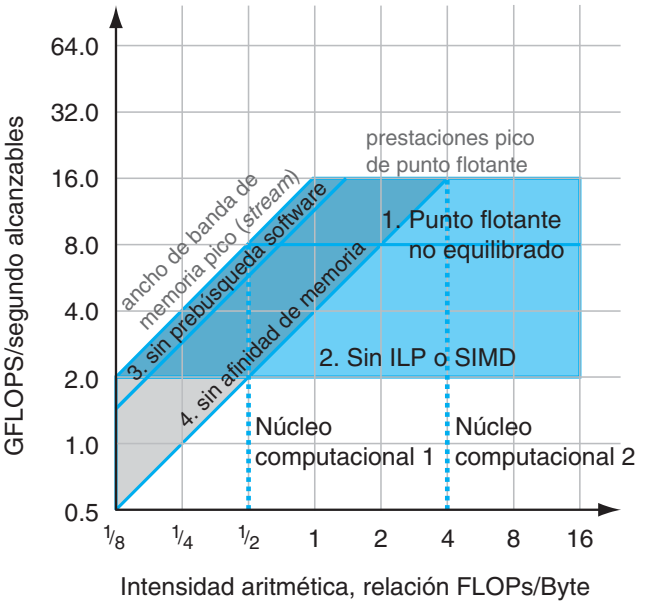


FIGURA 7.16 Modelo Roofline con techos, solapamiento de áreas sombreadas y los dos núcleos computacionales de la figura 7.13. Los núcleos computacionales cuya intensidad aritmética está en el trapezoide azul de la derecha deberían centrarse en optimizaciones de computación, y los núcleos computacionales cuya intensidad aritmética está en el triángulo gris de la parte inferior izquierda deberían centrarse en las optimizaciones de ancho de banda de la memoria. Los que están en el paralelogramo gris-azul del centro, deben preocuparse por ambos tipos de optimizaciones. Como el núcleo computacional 1 está en el paralelogramo del centro deben intentarse las optimizaciones ILP, SIMD, afinidad de memoria y prebúsqueda software. El núcleo computacional 2 está en el trapezoide de la derecha, entonces deben intentarse las optimizaciones ILP, SIMD y equilibrado de las operaciones de punto flotante.

Extensión: Una alternativa al uso del programa de prueba Stream es utilizar directamente el ancho de banda de la DRAM como línea del tejado. Las memorias DRAM fijan un límite difícil y las prestaciones de las memorias actuales están tan alejadas de esta frontera que no es demasiado útil utilizarlo como límite superior. Es decir, ningún programa puede acercarse a este límite. La desventaja de la utilización de Stream es que una programación muy cuidadosa puede superar los resultados de Stream y la línea de tejado de memoria puede no ser un límite tan estricto como la línea de tejado computacional. Sin embargo, como pocos programadores serán capaces de obtener más ancho de banda que Stream, nosotros nos quedamos con Stream.

Extensión: Los dos ejes utilizados anteriormente han sido operaciones punto flotante por segundo e intensidad aritmética de los accesos a memoria. El modelo Roofline podría utilizarse con otros núcleos computacionales y computadores que necesitasen otras métricas diferentes para las prestaciones.

Por ejemplo, si el conjunto de trabajo cabe en la cache L2 del computador, el ancho de banda de la línea de tejado diagonal podría ser el ancho de banda de la cache L2 en lugar del ancho de banda de la memoria principal y la intensidad aritmética del eje X podría estar basada en FLOPs por byte de la cache L2 accedido. La línea diagonal de las prestaciones de la L2 podría moverse hacia arriba y, probablemente, el punto cresta podría desplazarse hacia la izquierda.

Otro ejemplo, si el núcleo computacional fuese una clasificación (sort), podríamos reemplazar las operaciones punto flotante por instrucción en el eje Y por registros ordenados por segundo, y la intensidad aritmética por registros por byte DRAM accedido.

El modelo Roofline podría utilizarse incluso en un núcleo computacional con muchas operaciones de E/S. El eje Y podría ser operaciones de E/S por segundo, el eje X el número medio de instrucciones por operación de E/S y el modelo podría mostrar el ancho de banda de E/S pico.

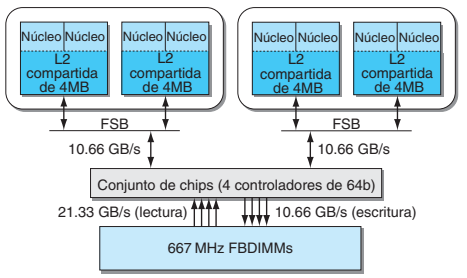
Extensión: Aunque el modelo Roofline está pensado para procesadores multinúcleo, obviamente podría utilizarse también en sistemas monoprocesador.

7.11 Casos reales: evaluación de cuatro multinúcleos con el modelo Roofline

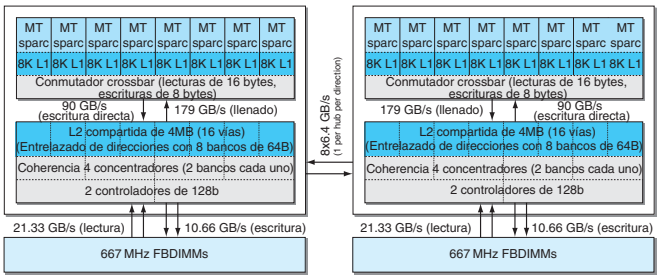
Dada la incertidumbre existente sobre la mejor forma de proceder en esta revolución paralela, no es extraño que veamos tantos diseños diferentes como chips multinúcleo. En esta sección analizaremos cuatro sistema multinúcleo para dos núcleos computacionales incluidos en los patrones de diseño de la figura 7.11: matrices dispersas y mallas estructuradas. (La información de esta sección ha sido extraída de [Williams, Oliker et al., 2007], [Williams, Carter et al., 2008] y [Williams and Patterson, 2008].)

Cuatro sistemas multinúcleo

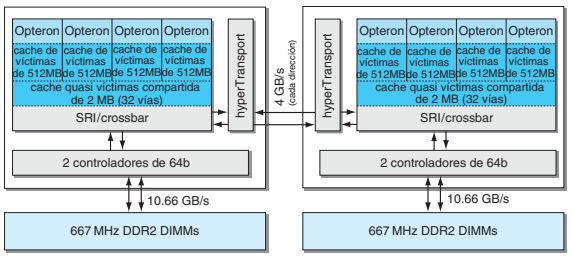
La organización básica de los cuatro sistemas se muestra en la figura 7.17 y las características claves de los ejemplos de esta sección en la figura 7.18. Son sistemas de ranura doble. La figura 7.19 muestra el modelo de prestaciones Roofline para cada sistema.



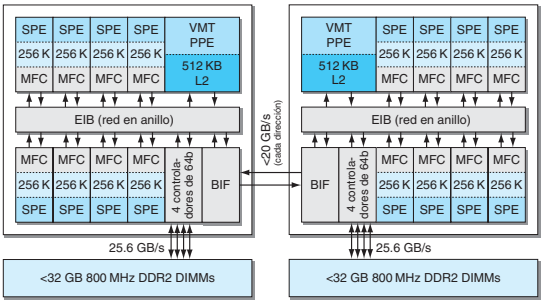
(a) Intel Xeon e5345 (Clovertown)



(c) Sun UltraSPARC T2 5140 (Niagara 2)



(b) AMD Opteron X4 2356 (Barcelona)



(d) IBM Cell QS20

FIGURA 7.17 Cuatro multiprocesadores recientes, cada uno con dos ranuras para procesadores. Comenzando por la esquina superior izquierda, los computadores son (a) Intel Xeon e5345 (Clovertown), (b) AMD Opteron X4 2356 (Barcelona), (c) Sun UltraSPARC T2 5140 (Niagara 2) y (d) IBM Cell QS20. Observe que el Intel Xeon e5345 (Clovertown) tiene un chip puente norte que no existe en los restantes microprocesadores.

MPU	ISA	Nº de hilos	Nº de núcleos	Nº de ranuras	Reloj GHz	GFLOP/s pico	DRAM: GB/s pico, frecuencia de reloj, tipo	
Intel Xeon e5345 (Clovertown)	x86/64	8	8	2	2.33	75	FSB: 2 x 10.6	667 MHz FBDIMM
AMD Opteron X4 2356 (Barcelona)	x86/64	8	8	2	2.30	74	2 x 10.6	667 MHz DDR2
Sun UltraSPARC T2 5140 (Niagara 2)	Sparc	128	16	2	1.17	22	2 x 21.3 (lect) 2 x 10.6 (escr)	667 MHz FBDIMM
IBM Cell QS20	Cell	16	16	2	3.20	29	2 x 25.6	XDR

FIGURA 7.18 Características de los cuatros multinúcleos recientes. Aunque el Xeon e5345 y el Opteron X4 tienen DRAMs de la misma velocidad, el programa de prueba Stream produce un ancho de banda práctico mayor debido a la ineficiencias del bus del sistema del Xeon e5345.

El Intel Xeon e5345 (Clovertown) tiene cuatro núcleos por ranura, con dos chips de doble núcleo en cada ranura. Los dos chips comparten un bus del sistema unido al conjunto de chips del puente norte (véase capítulo 6). El conjunto de chips del puente norte puede soportar hasta dos buses del sistema, y por lo tanto dos ranuras. Incluye el controlador de memoria para DIMMs DRAM con búfer completo (FBDIMM) a 667 MHz. La frecuencia del reloj es 2.33 GHz y tiene las prestaciones pico más elevadas de los cuatro ejemplos: 75 GFLOPS. Sin embargo, el modelo Roofline de la figura 7.19 pone de manifiesto que estas prestaciones pico sólo se alcanzan con una intensidad aritmética de 8 o superior. El motivo son las interferencias entre los buses del sistema, que da como resultado un ancho de banda de memoria relativamente bajo.

El AMD Opteron X4 2356 (Barcelona) tiene cuatro núcleos por chip, con un chip por ranura. En cada chip hay un controlador de memoria integrado y su conexión con una DRAM DRR2 a 667 MHz. Las dos ranura se comunican mediante una interconexión HyperTransport que hace posible construir un sistema multichip. El procesador tiene una frecuencia de reloj de 2.30 GHz y unas prestaciones pico de 74 GFLOPS. La figura 7.19 muestra que el punto cresta del modelo Roofline está a la izquierda del punto cresta del Xeon e5345 (Clovertown), con una intensidad aritmética de aproximadamente 5 FLOPs por byte.

El Sun UltraSPARC T2 5140 (Niagara 2) es bastante diferente de las dos microarquitecturas x86. Tiene ocho núcleos relativamente sencillos por chip y una frecuencia de reloj mucho más baja. Proporciona ejecución multihilo de grano fino con ocho hilos por núcleo. Cada chip tiene cuatro controladores de memoria que pueden controlar cuatro conjuntos de FBDIMMs a 667 MHz. Dos de los cuatro canales de memoria se emplean para unir dos chips UltraSPARC T2, dejando libres dos canales de memoria por chip. Este sistema de doble ranura tiene unas prestaciones pico de aproximadamente 22 GFLOPS y el punto cresta está en una sorprendentemente baja intensidad aritmética de 1/3 FLOPs por byte.

El IBM Cell QS20 es, a su vez, diferente de las dos microarquitecturas x86 y del UltraSPARC T2. Es un sistema heterogéneo con un PowerPC relativamente sencillo y ocho SPEs (*Synergisttic Processing Elements*) que tiene su propio reper-

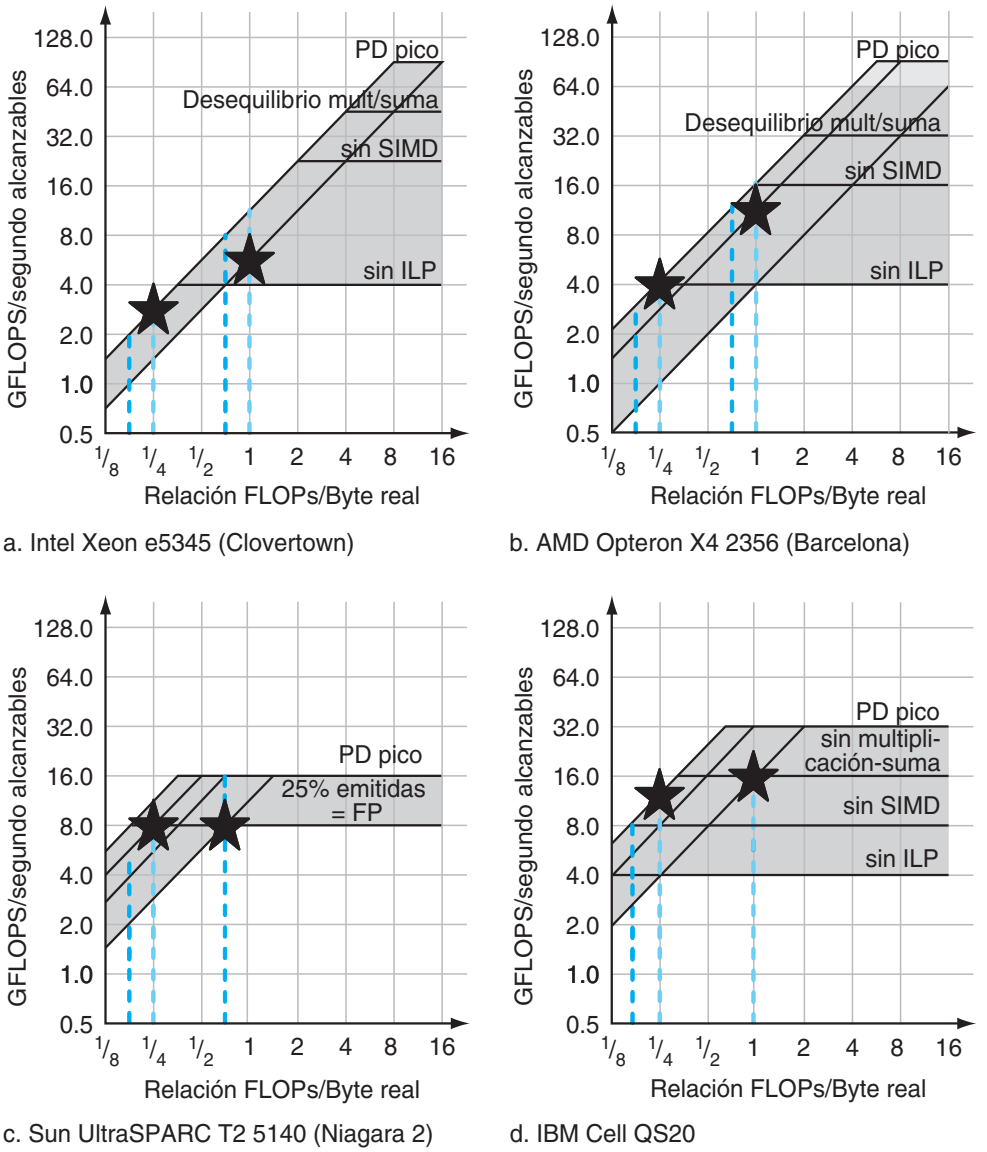


FIGURA 7.19 Modelo Roofline de los multiprocesadores multihilo de la figura 7.15. Los techos son los mismos que los de la figura 7.13. Comenzando por la esquina superior izquierda, los computadores son (a) Intel Xeon e5345 (Clovertown), (b) AMD Opteron X4 2356 (Barcelona), (c) Sun UltraSPARC T2 5140 (Niagara 2) y (d) IBM Cell QS20. Observe que los puntos cresta cruzan el eje X en intensidades aritméticas de 6, 4, 1/4 y 3/4 respectivamente. Las líneas verticales punteadas son para los dos núcleos computacionales de esta sección y el comienzo marca las prestaciones alcanzadas para estos núcleos computacionales después de todas las optimizaciones. El par de líneas verticales punteadas de la izquierda son para SpMV. Hay dos líneas verticales porque se mejoró la intensidad aritmética de 0.166 a 0.255 con optimizaciones de bloques de registros. La línea vertical de la derecha corresponde a LBHMD. Tiene un par de líneas en (a) y (b) porque una optimización que evita llenar el bloque de cache en caso de fallo permite aumentar la intensidad aritmética de 0.7 a 1.07. Hay solo una línea en (c) y (d) porque el UltraSPARC T2 no implementa esta optimización y el Cell tiene un almacenamiento local con DMA y no busca datos innecesarios como en las caches.

torio de instrucciones de estilo SIMD. Cada SPE tiene una memoria local en lugar de cache y debe transferir los datos desde la memoria principal a la memoria local antes de operar sobre ellos y devolverlos a la memoria principal cuando ha terminado. Utiliza DMA, que tiene cierta similitudes con la prebúsqueda software. Las dos ranuras se conectan vía conexiones dedicadas para formar un sistema multichip. La frecuencia del reloj es la más elevada de los cuatro multinúcleos, 3.2 GHz, y utiliza chips DRAM XDR, típicos de las consolas de videojuegos. Tiene un ancho de banda elevado pero poca capacidad. Como la principal aplicación de los Cell son los gráficos, las prestaciones de precisión simple son mucho más elevadas que las de precisión doble. Las prestaciones pico de precisión doble de los SPEs en el sistema de ranura doble son de 29 GFLOPS y el punto cresta está en una intensidad aritmética de 0.75 FLOPs por byte.

A comienzos del 2008, las dos arquitecturas x86 tenían muchos menos núcleos por chip que las arquitecturas de IBM y Sun, y esta diferencia todavía se mantiene en la actualidad. Como la expectativa es que el número de núcleos por chip se duplique con cada generación de la tecnología, va a ser interesante comprobar si las arquitecturas x86 reducirán esta distancia o, por el contrario, IBM y Sun pueden mantener un número de núcleos mayor, teniendo en cuenta que unas arquitecturas se centran en los servidores y otras en los computadores de sobremesa.

Observe que estas arquitecturas adoptan alternativas muy diferentes para el sistema de memoria. El Xeon e5345 utiliza una cache L1 privada convencional y la L2 se comparte entre pares de procesadores, que se conectan a una memoria común, a través de un controlador situado en un chip diferente y dos buses. Por el contrario, el Opteron X4 tiene un controlador de memoria independiente y memoria en cada chip y cada núcleo tiene caches L1 y L2 privadas. El UltraSPARC T2 tiene el controlador de memoria en el chip y cuatro canales DRAM independientes en cada chip; los núcleos comparten una cache L2, con cuatro bancos para mejorar las prestaciones. La ejecución multihilo de grano fino le permite mantener muchos acceso a memoria en ejecución. El diseño más radical es el del Cell. Tiene memorias locales privadas en cada SPE y utiliza DMA para la transferencia de datos entre la DRAM asociada a cada chip y la memoria local. Mantiene muchos accesos a memoria en ejecución gracias a la disponibilidad de muchos núcleos y de muchas transferencias DMA por núcleo.

Veamos ahora como se comportan estos multinúcleos con dos núcleos computacionales.

Matriz dispersa

El primer núcleo computacional que tomaremos como ejemplo, perteneciente al patrón de diseños computacionales Matriz Dispersa, es la Multiplicación Matriz Dispersa–Vector (Sparse Matrix-Vector Multiply, SpMV). SpMV es muy habitual en computación científica, en el modelado económico y en recuperación de la información. Desafortunadamente, las prestaciones de las implementaciones convencionales son bajas, menos del 10% de las prestaciones pico de un monoprocesador. El motivo principal es el patrón irregular de los accesos a memoria, que por

otra parte, es lo que se podría esperar de un núcleo computacional que trabaja con matrices dispersas. La operación es

$$y = A \times x$$

donde A es una matriz dispersa y x e y son vectores densos. Para la evaluación de las prestaciones de SpMV se han utilizado 14 matrices dispersas tomadas de varias aplicaciones reales diferentes, pero los resultados mostrados aquí corresponden a las prestaciones medias. La intensidad aritmética varía entre 0.166 antes de optimizaciones de bloques de registros y 0.25 FLOPS por byte después de implementar la optimizaciones.

El código se paralelizó para utilizar todos los núcleos. Como la intensidad aritmética de SpMV está por debajo de los puntos cresta de los cuatro multinúcleo de la figura 7.19, la mayoría de las optimizaciones afectan al sistema de memoria:

- *Prebúsqueda*. Para obtener las máximas prestaciones del sistema de memoria se utilizó prebúsqueda software y hardware.

- *Afinidad de memoria*. Esta optimización reduce los accesos a la memoria DRAM conectada a las otras ranuras en los tres sistemas con memorias DRAM locales.

- *Compresión de estructuras datos*. Como el ancho de banda de memoria limita las prestaciones, esta optimización utiliza estructuras de datos más pequeñas para aumentar las prestaciones —por ejemplo, utilizando índices de 16 bits en lugar de 32 bits y una representación más eficiente de los ceros en las filas de la matriz dispersa.

La figura 7.20 muestra las prestaciones obtenidas con SpMV frente el número de núcleos en los cuatro sistemas. (Estos mismo resultados se encuentran en la figura 7.19 pero es difícil comparar las prestaciones en una escala logarítmica). Observe que a pesar de tener las prestaciones pico más elevadas en la figura 7.18 y las prestaciones más elevadas con un solo núcleo, el Intel Xeon e5345 obtiene las prestaciones más bajas de los cuatro sistemas, hasta el punto que el Opteron X4 duplica sus prestaciones. El cuello de botella del Intel Xeon e5345 es el bus del sistema dual. El Sun UltraSPARC T2 se comporta mejor que las dos arquitecturas x86, gracias al elevado número de núcleos y a pesar de tener la frecuencia de reloj más baja. El IBM Cell tiene las prestaciones más elevadas de los cuatro sistemas. Observe que todos, excepto el Xeon e5345, escalan bien con el número de núcleos, aunque el Opteron X4 escala más lentamente con cuatro o más núcleos.

Mallas estructuradas

El segundo núcleo computacional está tomado de los patrones de diseño de mallas estructuradas. El núcleo Lattice-Boltzmann Magneto-Hydrodinamics (LBMHD) es habitual en dinámica de fluidos computacional; es un código de malla estructurada con una serie de pasos de tiempo.

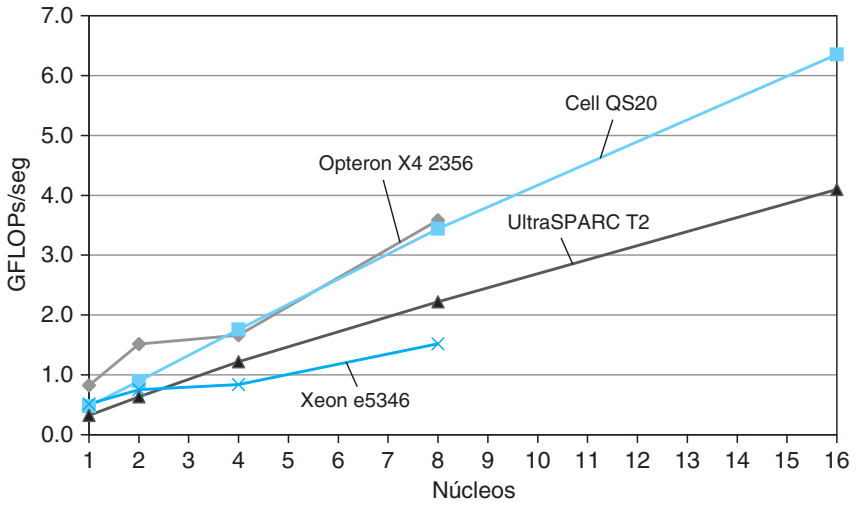


FIGURA 7.20 Prestaciones de SpMV en los cuatro multinúcleos.

El procesamiento de cada punto implica la lectura y escritura de aproximadamente 75 números punto flotante de precisión doble y unas 1300 operaciones de punto flotante. Al igual que SpMV, LBMHD tiende a obtener una pequeña fracción de las prestaciones pico de un monoprocesador debido a la complejidad de las estructuras de datos y a la irregularidad de los patrones de acceso a memoria. La relación FLOPs a bytes de memoria accedidos es mucho mayor que 0.7 frente a menos de 0.25 en SpMV. La intensidad aumenta hasta 1.07 si, en caso de fallo de escritura en cache, no se llena el bloque completo cuando el programa está a punto de sobreescribir el bloque entero. Todos los multinúcleo excepto el UltraSPARC T2 (Niagara 2) implementan esta optimización.

La figura 7.19 muestra que la intensidad aritmética de LBMHD es los suficientemente alta para que tenga sentido implementar tanto las optimizaciones computacionales como las optimizaciones del ancho de banda de memoria en todos los multinúcleo, excepto en el UltraSPARC T2, cuyo punto cresta está por debajo de LBMHD. Por lo tanto, en el UltraSPARC T2 implementamos sólo las optimizaciones computacionales.

Además, se utilizaron las siguientes optimizaciones para paralelizar LBMHD de forma que pueda hacer uso de todos los núcleos:

- *Afinidad de memoria*: Esta optimización es útil otra vez por las razones mencionadas anteriormente.
- *Minimización de fallos de TLB*: Para reducir los fallos de TLB de forma significativa, se utiliza una estructura de vectores y matrices y se combinan algunos lazos en lugar de la alternativa más convencional basada en la utilización de vectores y matrices de estructuras.

- *Desenrollamiento y reordenamiento de lazos* : Para exponer el suficiente paralelismo y mejorar la utilización de la cache, los lazos se han desenrollado y reordenado para agrupar sentencias con direcciones parecidas.
- *"SIMD-zar"*: El compilador de los dos sistemas x86 podría no generar buenos códigos SSE, por lo tanto se escribieron a mano en lenguaje ensamblador.

La figura 7.21 muestra las prestaciones obtenidas con LBMHD frente el número de núcleos en los cuatro sistemas. Al igual que con SpMV, el Intel Xeon e5345 es el que peor escala. Esta vez, los núcleos más potentes del Opteron X4 obtienen mejores prestaciones que los núcleos más sencillos del UltraSPARC T2 a pesar de tener la mitad de núcleos. Una vez más, el sistema más rápido es el IBM Cell. Todos excepto el Intel Xeon e5345 escalan con el número de núcleos, aunque el T2 y el Cell escalan de forma más suave que el Opteron X4.

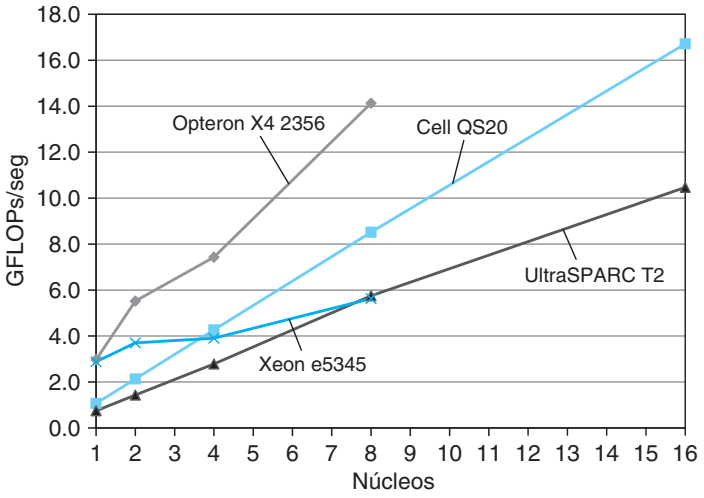


FIGURA 7.21 Prestaciones de LBMHD en los cuatro multinúcleos.

Productividad

Además de las prestaciones, otro factor a tener en cuenta en la revolución del procesamiento paralelo es la productividad, definida como la dificultad para obtener buenas prestaciones. Para entender las diferencias, la figura 7.22 compara las prestaciones básicas (sin optimizaciones) con las prestaciones después de implementar todas las optimizaciones para los cuatro procesadores y los dos núcleos computacionales.

MPU	Núcleo computacional	GFLOPS/s Base	GFLOPs/s optimizado	Base % de optimizado
Intel Xeon e5345 (Clovertown)	SpMV	1.0	1.5	64%
	LBMHD	4.6	5.6	82%
AMF Opteron X4 2356 (Barcelona)	SpMV	1.4	3.6	38%
	LBMHD	7.1	14.1	50%
Sun UltraSPARC T2 (Niagara 2)	SpMV	3.5	4.1	86%
	LBMHD	9.7	10.5	93%
IBM Cell QS20	SpMV	-	6.4	0%
	LBMHD	-	16.7	0%

FIGURA 7.22 Prestaciones básicas frente a las prestaciones de una implementación completamente optimizada para los cuatro procesadores y los dos núcleos computacionales. Observe la elevada fracción de las prestaciones de una implementación completamente optimizada en el Sun UltraSPARC T2 (Niagara 2). No hay datos de prestaciones base para el IBM Cell porque no es posible ejecutar el código en los SPEs, que no tienen caches. Se puede ejecutar el código en el núcleo PowerPC, pero se obtienen unas prestaciones un orden de magnitud menor que en los SPEs, por lo que decidimos ignorar estos datos.

El más fácil es el Sun UltraSPARC T2 (Niagara 2), debido a su elevado ancho de banda de memoria y sus núcleos sencillos. Las conclusiones para el Sun UltraSPARC T2 están claras, intentar obtener un código del compilador con buenas prestaciones y utilizar tantos hilos como sea posible. La única precaución para otros núcleos computacionales es que puede haber un conflicto con el error habitual de la página 545 del capítulo 5, sobre la conveniencia de asegurar que el conjunto de asociatividad sea igual al número de hilos hardware. Cada chip soporta 64 hilos hardware, mientras que la cache L2 es asociativa por conjuntos de 4 vías. Debido a esta desigualdad, podría ser necesario reestructurar algunos lazos para reducir los fallos de conflicto.

El Xeon e5345 es difícil por la dificultad para comprender el comportamiento de la memoria y los buses de sistema duales, para entender el funcionamiento de la prebúsqueda hardware y para obtener un buen código SIMD con el compilador. Su código C y el del Opteron X4 están llenos de instrucciones SIMD para obtener buenas prestaciones.

El Opteron X4 se beneficia de la mayoría de la optimizaciones, por lo tanto necesita un mayor esfuerzo que el Xeon e5345 aunque el comportamiento del sistema de memoria del Opteron X4 es más fácil de entender que el del Xeon e5345.

En el Cell hay dos tipos de retos. Primero, las instrucciones SIMD de los SPEs son difíciles para el compilador, por lo que en ocasiones es necesario ayudar al compilador insertando instrucciones de lenguaje ensamblador en el código C. Segundo, el sistema de memoria es más interesante. Como cada SPE tiene una memoria local y un espacio de direcciones diferenciado, no es posible simplemente ejecutar el código sin más en el SPE. En consecuencia, no hay una columna de prestaciones base para el IBM Cell en la figura 7.22, y es necesario cambiar el programa para introducir órdenes DMA para la transferencia de

información entre la memoria y el almacenamiento local. La parte positiva es que el DMA juega el papel de la prebúsqueda software en las caches y el DMA es más fácil de utilizar y es más sencillo obtener unas buenas prestaciones de memoria. El Cell fue capaz de obtener un 90% del ancho de banda límite del modelo Roofline con estos núcleos computacionales, comparado con el 50% o menos de los otros multinúcleos.

Durante una década los profetas han manifestado su desacuerdo con que la organización de un solo computador haya alcanzado su límites y que realmente se pueden conseguir avances importantes mediante la interconexión de varios computadores de forma que puedan cooperar para encontrar una solución ... Se hacen demostraciones de la validez de un único procesador ...

Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities". Spring Joint Computer Conference, 1967

7.12 Falacias y errores habituales

La evolución del procesamiento paralelo ha destapado muchas numerosas falacias y errores. Aquí analizamos algunas de ellas.

Falacia: La ley de Amdahl no se aplica a los computadores paralelos.

En 1987, el director de una organización de investigación reivindicaba que la ley de Amdahl había sido superada por un sistema multiprocesador. Para comprender las bases de esta afirmación, recordamos la ley de Amdahl [1967, p. 483]:

> *Una conclusión obvia e imparcial que puede hacerse en este punto es que el esfuerzo invertido en alcanzar tasas de procesamiento paralelo elevadas será baldío a menos que se acompañen de logros de la misma magnitud en las tasas de procesamiento secuencial.*

Esta afirmación debe ser todavía cierta; una parte pequeña y olvidada del programa puede limitar las prestaciones. La interpretación de la ley lleva al siguiente lema: en todo programa hay partes que tienen que ser secuenciales, por lo tanto debe haber un límite superior en el número de procesadores, por ejemplo, 100. Este lema se refuta si se obtiene una aceleración lineal con 1000 procesadores, de ahí la afirmación de que la Ley de Amdahl ha sido superada.

El enfoque de los investigadores fue utilizar escalado débil: en lugar de ir 1000 veces más rápido con el mismo conjunto de datos, se realizaba un trabajo 1000 veces mayor en un tiempo comparable. En sus algoritmos, la parte secuencial del programa no cambiaba, independientemente del tamaño de las entradas, y el resto era totalmente paralela, de ahí la aceleración lineal con 1000 procesadores.

Obviamente, la ley de Amdahl se aplica a procesadores paralelos. Lo que se pone de manifiesto es que uno de los usos principales de los computadores más rápidos es ejecutar problemas de mayor tamaño, pero prestando atención a como escala el algoritmo al aumentar el tamaño del problema.

Falacia: Las prestaciones pico siguen a las prestaciones observadas.

Por ejemplo, en la sección 7.11 se muestra que el Intel Xeon e5345, el microprocesador con las mayores prestaciones pico, tiene las prestaciones más baja de los cuatro microprocesadores multinúcleo con los dos núcleos computacionales.

Los fabricantes de supercomputadores utilizaron esta métrica en marketing y los computadores paralelos agravaron esta falacia. Los vendedores no sólo utilizan las casi inalcanzables prestaciones pico de los nodos monoprocesador, sino que además ¡asumen una aceleración perfecta y multiplican estas prestaciones por el número total de procesadores! La ley de Amdahl pone de manifiesto que es difícil llegar a las prestaciones pico, y hacer esa multiplicación multiplica el pecado. El modelo Roofline nos ayuda a poner las prestaciones pico en su lugar.

Error: No desarrollar u optimizar el software para aprovechar las características de una arquitectura multiprocesador.

Tradicionalmente, y desde hace ya bastante tiempo, el software se queda a la zaga de los procesadores paralelos, posiblemente debido a la dificultad de los problemas del software. A continuación, ponemos un ejemplo de la sutileza de la cuestión, pero ¡hay otros muchos ejemplos que podríamos escoger!

Frecuentemente nos encontramos con que software diseñado para un sistema monoprocesador se adapta a un sistema multiprocesador. Por ejemplo, el sistema operativo de SGI originalmente protegía la tabla de página con un bloqueo simple, asumiendo que la asignación de páginas es infrecuente. En un monoprocesador, esto no afecta a las prestaciones. Sin embargo, en un multiprocesador se puede convertir en un cuello de botella importante para algunos programas. Consideremos un programa que utiliza un gran número de páginas que se inicializan al arrancar, algo que en UNIX se hace para las páginas asignadas de forma estática. Supongamos que se paraleliza el programa para que varios procesos asignen las páginas. Como la asignación de páginas necesita usar la tabla de páginas, que se bloquea cuando alguien la está usando, el acceso se serializa si todos los procesos intentan asignar sus páginas al mismo tiempo (que es exactamente lo que se podría esperar que ocurriese en la inicialización), incluso cuando el núcleo del sistemas operativo permite varios hilos.

Esta serialización en el acceso a la tabla de páginas elimina el paralelismo en la inicialización y tiene un impacto importante sobre las prestaciones paralelas del sistema. Este cuello de botella persiste incluso para paralelismo a nivel de trabajos. Por ejemplo, supongamos que se divide el programa de procesamiento paralelo en trabajos separados y que se ejecuta un trabajo en cada procesador, de modo que no hay compartición entre trabajos. (Esto es exactamente lo que hizo un usuario, creyendo razonablemente que el problema era la compartición no intencionada o la interferencia en su aplicación.) Desafortunadamente, el bloqueo aún serializa todos los trabajos y las prestaciones con trabajos independientes siguen siendo bajas.

Todo esto pone de manifiesto la clase de errores, casi imperceptibles pero importantes, con que nos podemos encontrar cuando el software se ejecuta en un sistema multiprocesador. Al igual que otros muchos componentes software clave, los algoritmos y las estructuras de datos del SO deben ser repensados para un sistema multiprocesador. Poniendo los bloqueos en partes más pequeñas de la tabla de páginas, se elimina este problema.

Estamos dedicando todos los desarrollos de nuestros futuros productos al diseño multinúcleo. Creemos que esto es un punto de inflexión clave para la industria ... Esto no es una carrera, es un cambio enorme en computación ...

Paul Otellini, Presidente de Intel, Intel Developers Forum, 2004

7.13 Conclusiones finales

El sueño de construir computadores agregando procesadores ha estado presente desde los primeros tiempos de la computación. Sin embargo, los progresos en la construcción y el uso efectivo y eficiente los procesadores paralelos han sido lentos. El ritmo de progreso ha estado limitado por los problemas del software y por un largo proceso de evolución de la arquitectura de los multiprocesadores para facilitar su uso y mejorar la eficiencia. En este capítulo hemos analizado muchos de los retos del software, incluyendo la dificultad de escribir programas que obtengan buenas aceleraciones debido a la ley de Amdahl. La amplia variedad de soluciones arquitecturales diferentes y el éxito limitado y la corta vida de las arquitecturas paralelas en el pasado han agravado las dificultades del software. La historia del desarrollo de estos multiprocesadores se analiza en la **sección 7.14** en el CD.

Tal como se ha dicho en el capítulo1, a pesar de este largo y accidentado pasado, la industria de la tecnología de la información ha unido ahora su futuro a la computación paralela. Aunque es fácil decir que este esfuerzo también va a fallar, como tantos otros en el pasado, hay razones para ser optimista:

Software como servicio: en lugar de vender software que se instala y ejecuta en los computadores de los clientes, el software se está ejecutando en un servidor remoto y los clientes acceden a él a través de Internet, con una interfaz Web. El coste para los clientes está basado en el uso que se haga del mismo.

- Claramente, el **software como servicio** está ganando cada vez más relevancia y los clústeres han demostrado ser muy valiosos para garantizar tales servicios. Mediante la redundancia al más alto nivel, incluyendo los centros de datos distribuidos geográficamente, tales servicios están disponibles 24 × 7 × 365 (24 horas/día, 7 días/semana, 365 días/año) para clientes en cualquier parte del mundo. Es fácil suponer que tanto el número de servidores por centro como el número de centros de datos continuará creciendo. Seguramente, estos centros de datos adoptarán el diseño multinúcleo porque ya están utilizando miles de procesadores en sus aplicaciones.

- El uso del procesamiento paralelo en el dominio de la computación científica y la ingeniería es ya habitual. Este dominio de aplicación tiene una necesidad casi ilimitada de más capacidad de cómputo. Además, hay muchas aplicaciones en este dominio altamente concurrentes. Otra vez, los clústeres dominan en este área de aplicación. Por ejemplo, basándonos en los informes de Linpack del año 2007, los clústeres representan más del 80% de los 500 computadores más rápidos. No obstante, no ha sido fácil: la programación paralela sigue siendo un reto incluso para estas aplicaciones. Con seguridad, también este grupo adoptará los sistemas multinúcleo, porque también tienen experiencia en la utilización de miles de procesadores.

- Todos los fabricantes de microprocesadores para servidores y computadores personales están implementando multiprocesadores para obtener altas prestaciones, así, a diferencia de lo que ocurrió en el pasado, no hay un

camino fácil para aumentar las prestaciones de las aplicaciones secuenciales. Así, los programadores que necesitan prestaciones más elevadas tienen que paralelizar sus códigos o escribir nuevos programas de procesamiento paralelo.

- Disponer de varios procesadores en el mismo chip permite velocidades de comunicación muy diferentes que los diseños con varios chips, ofreciendo una latencia mucho menor y un ancho de banda mucho mayor. Estas mejoras hacen más fácil alcanzar buenas prestaciones.

- En el pasado, los microprocesadores y los multiprocesadores utilizaban diferentes definiciones para el éxito. En el escalado de las prestaciones de un monoprocesador los arquitectos se sentían satisfechos si las prestaciones de un único hilo aumentaban en un factor igual a la raíz cuadrada del aumento del área. Es decir, se contentaban con unas prestaciones sub-lineales frente a los recursos. En cambio, en los multiprocesadores se solía definir el éxito como una aceleración lineal frente al número de procesadores, aceptando que el coste de compra o de administración de un sistema con *n* procesadores era *n* veces el de un procesador. En la actualidad, con la utilización de los multinúcleo, el paralelismo aparece dentro del chip y, al igual que en los microprocesadores tradicionales, podemos utilizar la mejora sublineal de las prestaciones como definición del éxito.

- El éxito de la *compilación en tiempo de ejecución (just-in-time runtime compilation)* hace posible pensar en la autoadaptación del software para aprovechar el número creciente de núcleos por chip, porque proporciona una flexibilidad que no está disponible en los compiladores estáticos.

- Actualmente, el software libre es una parte importante de la industria del software. El movimiento del software libre es una meritocracia donde las mejores soluciones desde el punto de vista de la ingeniería pueden volverse populares sin importar los problemas legales. Promueve la innovación, invitando a cambiar el software antiguo y dando la bienvenida a nuevos lenguajes y productos software. Esta mentalidad abierta puede ser extremadamente útil en estos tiempos de cambios rápidos.

Esta revolución en la interfaz hardware/software quizás sea el mayor reto que ha tenido que encarar la computación en los últimos 50 años. Proporcionará nuevas oportunidades de investigación y negocio, dentro y fuera del campo de las Tecnologías de la Información, y las empresas que dominen en la era de los multinúcleos no tienen porque ser las mismas que han dominado en la era de los monoprocesadores. Quizás usted sea uno de los innovadores que aprovecharán las oportunidades que sin duda aparecerán en los tiempos de incertidumbre que se avecinan.

7.14 Perspectiva histórica y lecturas recomendadas

Esta sección en el CD resume la rica, y a menudo desastrosa, historia de los sistemas multiprocesador durante los últimos 50 años.

7.15 Ejercicios

Contribución de David Kaeli, Northeastern University

Ejercicio 7.1

En primer lugar, escriba una lista de sus actividades diarias durante un día de semana. Por ejemplo, levantarse de la cama, ducharse, vestirse, desayunar, secar el pelo, cepillarse los dientes, etc. Asegúrese de que haya un mínimo de 10 actividades en su lista.

7.1.1 [5]<7.2> Piense cuál de esas actividades explota algún tipo de paralelismo (por ejemplo, cepillarse varios dientes al mismo tiempo frente a hacerlo de uno en uno, llevar varios libros en la mochila, "en paralelo", frente a llevar los libros de uno en uno). Para cada una de las actividades, analice si ya se hacen en paralelo y en caso contrario indique por qué.

7.1.2 [5]<7.2> Investigue qué actividades podrían hacerse concurrentemente (por ejemplo, desayunar y escuchar las noticias). Indique qué otras acciones podrían emparejarse con cada una de las actividades de su lista.

7.1.3 [5]<7.2> ¿Qué se debería cambiar en los sistemas actuales (duchas, TV, coches) para permitir más tareas en paralelo?

7.1.4 [5]<7.2> Estime cuánto tiempo se ahorraría en esas actividades si se intenta hacer el mayor número posible de tareas en paralelo.

Ejercicio 7.2

Muchas aplicaciones informáticas necesitan realizar búsquedas y ordenamientos en conjuntos de datos y, en consecuencia, para reducir el tiempo de ejecución de estas tareas, se han desarrollado muchos algoritmos de búsquedas y ordenamientos eficientes en grandes conjuntos de datos. En este problema, se considera la paralelización de estas tareas.

7.2.1 [10]<7.2> Considere el siguiente algoritmo de búsqueda binaria (el clásico algoritmo divide-y-vencerás) que busca un valor X en un vector ordenado A de N elementos y devuelve el índice de la entrada que contiene el valor X:

```
BinarySearch(A[0..N-1], X){
     low = 0
     high = N-1
     while (low <= high) {
          mid = (low + high) / 2
          if (A[mid] > X)
               high = mid -1
          else if (A[mid] < X)
               low = mid +1
          else
               return mid // encontrado
     }
     return -1 // no encontrado
}
```

Suponga que dispone de un procesador multinúcleo con Y núcleos, siendo Y<<N, indique el factor de aceleración que se podría obtener para varios valores de Y y N. Represéntelos en una gráfica.

7.2.2 [5]<7.2> Suponga ahora que Y es igual a N. ¿Cómo afectaría esta igualdad a las conclusiones del problema anterior? Si se quisiese obtener la mejor aceleración posible (escalado fuerte), explique qué cambios habría que introducir en el código.

Ejercicio 7.3

Considere el siguiente fragmento de código C:

```
for (j=2; j<1000; j++)
     D[j]=D[j-1]+D[j-2];
```

El código MIPS correspondiente es:

```
     DADDIU   r2, r2, 999
lazo:L.D      f1, -16(f1)
     L.D      f2, -8(f1)
     ADD.D    f3, f1, f2
     S.D      f3, 0(r1)
     DADDIU   r1, r1, 8
     BNE      r1, r2, lazo
```

y las instrucciones tienen las siguientes latencias (en ciclos):

ADD.D	L.D	S.D	DADDIU	BNE
3	5	1	1	3

7.3.1 [10]<7.2> ¿Cuántos ciclos se necesitan para ejecutar una iteración del lazo?

7.3.2 [10]<7.2> Se dice que hay una dependencia a través de lazo cuando una instrucción en una iteración depende de un valor obtenido en una iteración anterior del lazo. Identifique las dependencias a través del lazo en el código anterior y la variable y los registros del código ensamblador dependientes. Ignore la variable `j` de control del lazo.

7.3.3 [10]<7.2> En el capítulo 4 se describe el desenrollamiento de lazos. Aplique desenrollamiento de lazos a este código y suponga que se ejecuta en un sistema de memoria distribuida de paso de mensajes con dos nodos. Suponga que el paso de mensajes que se utiliza es el que se ha descrito en la sección 7.4, donde se introdujeron dos nuevas operaciones: send(x,y), que envía el valor y al nodo x, y receive( ), que espera por un valor. Suponga que se necesita un ciclo para emisión de la instrucción send, es decir, la siguiente instrucción en el mismo nodo puede comenzar en el siguiente ciclo, pero se tardan 4 ciclos en recibirse el valor en el nodo receptor. La instrucción receive detiene la ejecución del nodo en el que se ejecuta hasta que se recibe un mensaje. Obtenga una planificación para los dos nodos suponiendo un factor de desenrollamiento igual a 4 (es decir, el cuerpo del lazo aparece 4 veces en una iteración). Determine el número de ciclos necesarios para la ejecución del lazo en el sistema de paso de mensajes.

7.3.4 [10]<7.2> La latencia de la red de interconexión juega un papel importante en la eficiencia de los sistemas de paso de mensajes. Determine cuál debería ser la velocidad de la red de interconexión del sistema distribuido del problema 7.3.3 para obtener aceleración en la ejecución del código.

Ejercicio 7.4

Considere otro algoritmo divide-y-vencerás clásico, el algoritmo recursivo de ordenamiento por mezcla (*mergesort*), descrito por primera vez en 1945 por John von Neumann. La idea básica es dividir una lista desordenada `x` de `m` elementos en dos sublistas cuyo tamaño es aproximadamente la mitad del tamaño de la lista original; se repite esta operación en cada sublista, y se continúa así hasta tener listas de tamaño 1. Entonces, empezando con la sublistas de tamaño 1, se mezclan dos sublistas en una lista ordenada.

```
Mergesort(m)
     var list left, right, resultados
     if length(m) <= 1
          return m
```

```
        else
            var middle = length(m) / 2
            for each x in m up to middle
                add x to the left
            for each x in m after middle
                add x to the right
            left = Mergesort(left)
            right = Mergesort(right)
            result = Merge(left, right)
            return resultados
```

El código para el mezcaldo (*merge*) es el siguiente:

```
Merge(left,right)
      var list result
      while length(left) >0 and length(right) >0
          if first(left) <= first(right)
              append first(left) to result
              left = rest(left)
          else
              append first(right) to result
              right = rest(right)
      if  length(left) >0
          append rest(left) to result
      if  length(right) >0
          append rest(right) to result
      return result
```

7.4.1 [10]<7.2> Suponga que se dispone de un procesador multinúcleo con Y núcleos. Si Y es mucho menor que `m` indique el factor de aceleración que se podría obtener para varios valores de Y y `m`. Represéntelos en una gráfica.

7.4.2 [10]<7.2> Suponga ahora que Y=N. ¿Cómo afectaría esta igualdad a las conclusiones del problema anterior? Si se quisiese obtener la mejor aceleración posible (escalado fuerte), explique qué cambios habría que introducir en el código.

Ejercicio 7.5

Usted está intentando cocinar tres bizcochos de arándanos con los siguientes ingredientes:

1 taza de mantequilla
1 taza de azúcar

4 huevos grandes
1 cucharadita de extracto de vainilla
1/2 cucharadita de sal
1/4 cucharadita de nuez moscada
1 1/2 tazas de harina
1 taza de arándanos

La receta de un bizcocho es la siguiente:

Precalentar el horno a 160 °C. Untar con aceite y enharinar en el molde.

En el cuenco grande, batir la mantequilla y el azúcar a velocidad media hasta que está esponjoso. Añadir los huevos, la vainilla, la sal y la nuez moscada. Batir hasta que se mezclen. Añadir la harina, de media en media taza, mezclando lentamente.

Incorporar con cuidado los arándanos. Extender uniformemente la masa en el molde y hornear durante 60 minutos.

7.5.1 [5]<7.2> Tiene que cocinar tres bizcochos los más eficientemente posible. Suponga que sólo dispone de un horno en el que cabe un bizcocho, un cuenco grande, un molde y una batidora. Obtener la planificación para hacer tres bizcochos los más rápidamente posible. Identifique los cuellos de botella.

7.5.2 [5]<7.2> Suponga ahora que dispone de tres cuencos grande, tres moldes y tres batidoras. ¿Qué aceleración se obtiene con estos recursos adicionales?

7.5.3 [5]<7.2> Suponga que dos amigos le ayudan a cocinar y que tiene un horno grande donde le caben los tres bizcochos. ¿Cómo afecta esto a la planificación del problema 7.5.1?

7.5.4 [5]<7.2> Compare la tarea de cocinar un bizcocho con la ejecución de tres iteraciones de un lazo en un computador paralelo. Identifique el paralelismo a nivel de datos y el paralelismo a nivel de tareas de lazo "cocinar un bizcocho".

Ejercicio 7.6

La multiplicación de matrices se utiliza frecuentemente en un gran número de aplicaciones. Se pueden multiplicar dos matrices si el número de columnas de la primera matriz es igual al número de filas de la segunda.

Suponga que se tiene una matriz A de dimensión $m \times n$ y se quiere multiplicarla por una matriz B $n \times p$. El resultado es una matriz AB (o $A \cdot B$) de dimensión $m \times p$. Si se define $C = AB$ y $c_{i,j}$ como el elemento en la posición (i,j) de C, entonces

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} b_{r,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \cdots + a_{i,n} b_{n,j}$$

para cada elemento i y j con $1 \leq i \leq m$ y $1 \leq j \leq p$. Se quiere comprobar si es posible paralelizar el cálculo de C. Suponga que las matrices se almacenan en memoria secuencialmente de la siguiente forma: $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$ ... etc.

7.6.1 [10]<7.3> Suponga que se quiere calcular C en un sistema de memoria compartida de un solo núcleo y en un sistema de memoria compartida de cuatro núcleos. Determine la aceleración que se obtiene con el sistema de cuatro núcleos, ignorando las cuestiones de memoria.

7.6.2 [10]<7.3> Repita el problema 7.6.1 suponiendo que se produce un fallo de cache debido a un compartición falsa cuando se actualizan varios elementos consecutivos de una fila (es decir, índice i) de C.

7.6.3 [10]<7.3> ¿Cómo se podría evitar la compartición falsa?

Ejercicio 7.7

Considere los siguientes fragmentos de dos programas diferentes que se ejecutan al mismo tiempo en cuatro procesadores de un multiprocesador simétrico (SMP). Suponga que inicialmente x = 0, y = 0.

Núcleo 1: `x = 2;`

Núcleo 2: `y = 2;`

Núcleo 3: `w = x + y + 1;`

Núcleo 4: `z= x + y;`

7.7.1 [10]<7.3> Determine todos los posibles valores finales de w, x, y, z. Para cada posible resultado, explique cómo se obtienen. Necesitará considerar todos los entrelazamientos posibles de las instrucciones.

7.7.2 [5]<7.3> ¿Cómo se puede conseguir una ejecución más determinística para que sólo se obtenga un resultado?

Ejercicio 7.8

En sistemas de memoria compartida con accesos no uniformes y coherencia cache (CC-NUMA), las CPUs y la memoria física están repartidas en varios nodos computacionales. Cada CPU tiene una cache local. Para mantener la coherencia de la memoria se añaden bits de estado en cada bloque de la cache o se introducen directorios de memoria específicamente dedicados a esta tarea. Con los directorios, cada nodo tiene una tabla hardware para reflejar el estado de cada bloque de la memoria alojado en su cache local. El tamaño de los directorios depende del tamaño del espacio de memoria compartido en el sistema CC-NUMA (hay una

entrada para cada bloque de memoria alojado en la cache local del nodo). Si esta información sobre la coherencia se almacena en la propia cache, se añade esta información en todas las cache del sistema (es decir, las necesidades de espacio de almacenamiento son proporcionales al número de bloques de cache disponibles en todas las caches).

En los siguientes problemas se supone que todos los nodos son iguales, con el mismo número de CPUs y la misma cantidad de memoria (es decir, las CPUs y memoria se han dividido a partes iguales entre los nodos del sistema CC-NUMA).

7.8.1 [15]<7.3> Si se tienen *P* CPUs distribuidas en *T* nodos de un sistema CC-NUMA, y cada CPU tiene *C* bloques de memoria con un byte de información de coherencia en cada bloque, exprese con una ecuación la cantidad de memoria presente en la caches de un único nodo para mantener la coherencia. Tenga en cuenta únicamente el espacio destinado al almacenamiento de la información sobre la coherencia.

7.8.2 [15]<7.3> Si cada entrada del directorio mantiene un byte de información para cada CPU, si el sistema CC-NUMA tiene un total de *S* bloques memoria y *T* nodos, exprese con una ecuación la cantidad de memoria necesaria para cada directorio.

Ejercicio 7.9

Considere el sistema CC-NUMA del ejercicio 7.8 y suponga que el sistema tiene cuatro nodos, cada uno con una CPU de un solo núcleo y caches de datos L1 y L2. La cache L1 es de escritura directa y la cache L2 de escritura retardada. Suponga que la carga de trabajo del sistema es tal que una CPU escribe en la memoria y las otras CPUs leen el dato que se ha escrito, y que la dirección sobre la que se escribe no está inicialmente en ninguna cache local. Además, suponga que, después de la lectura, el bloque actualizado está únicamente en las caches L1 y L2 de la CPU que hace la escritura.

7.9.1 [10]<7.3> Describa el tráfico entre los nodos que se generará en el sistema que mantiene la coherencia con bits de estado de los bloques de cache, si cada uno de los cuatro nodos escribe en una dirección de memoria y los restantes nodos leen cada dirección modificada.

7.9.2 [10]<7.3> Describa el tráfico entre nodos para el mismo patrón de ejecución en el sistema con coherencia basada en directorios.

7.9.3 [20]<7.3> Repita los problemas 7.9.1 y 7.9.2 suponiendo que las CPUs son multinúcleo, con cuatro núcleos por CPU, cache de datos L1 local a cada núcleo y cache de datos L2 compartida entre los cuatro núcleos. Cada núcleo hace una escritura y después los 15 núcleos restantes leen este valor.

7.9.4 [10]<7.3> Suponga ahora que, en el mismo sistema del problema 7.9.3, cada núcleo escribe dos bytes en el mismo bloque de cache. ¿Cómo afecta esto al tráfico del bus? Razone la respuesta.

Ejercicio 7.10

En un sistema CC-NUMA la latencia del acceso a la memoria no local puede limitar la utilización eficiente del multiprocesador. La siguiente tabla muestra las latencias de los accesos a datos en la memoria local y en la memoria no local, junto con la localidad de la aplicación, expresada como el porcentaje de acceso locales.

carga/almacenamiento local (ciclos)	carga/almacenamiento no local (ciclos)	% accesos locales
20	100	50

Suponga que los accesos a memoria están distribuidos equitativamente en la aplicación y que el sistema puede seguir procesando cuando el acceso a memoria está activo (no hay dependencias). Además, suponga que sólo un acceso a memoria puede estar activo durante un ciclo. Establezca todas las suposiciones sobre la ordenación de los accesos a memoria locales y no locales.

7.10.1 [10]<7.3> Si de media se necesita un acceso a memoria cada 75 ciclos, ¿cuál es el impacto sobre la aplicación?

7.10.2 [10]<7.3> Si de media se necesita un acceso a memoria cada 50 ciclos, ¿cuál es el impacto sobre la aplicación?

7.10.3 [10]<7.3> Si de media se necesita un acceso a memoria cada 100 ciclos, ¿qué impacto tiene sobre la aplicación?

Ejercicio 7.11

El problema de la cena de los filósofos es un problema clásico de sincronización y concurrencia. El enunciado general es el siguiente: los filósofos están sentados alrededor de una tabla redonda comiendo o pensando. Cuando están comiendo no piensan, y cuando están pensando no comen. Hay un cuenco de pasta en el centro y un tenedor en medio de cada pareja de filósofos, de modo que cada filósofo tiene un tenedor a su izquierda y otro a a su derecha. Para comer la pasta, cada filósofo necesita dos tenedores y sólo puede utilizar los que tiene a su izquierda o a su derecha. Los filósofos no hablan.

7.11.1 [10]<7.4> Describa el escenario en el que ningún filósofo puede comer (es decir, inanición). ¿Cuál es la secuencia de sucesos que lleva a esta situación?

7.11.2 [10]<7.4> Indique cómo se puede resolver este problema con la introducción del concepto de prioridad. ¿Podemos garantizar que todos los filósofos serán tratados por igual? Razone la respuesta.

Suponga ahora que un camarero se encarga de asignar tenedores a los filósofos. Nadie puede utilizar un tenedor hasta que el camarero, que tiene una visión glo-

bal de todos los tenedores, le da permiso. Además, se puede garantizar que el bloqueo se evita si se impone la condición de que los filósofos siempre pedirán el tenedor que está a su izquierda antes que el que está a su derecha.

7.11.3 [10]<7.4> Las solicitudes al camarero se pueden implementar como una cola de solicitudes o como intentos de solicitud periódicos. Con una cola, las solicitudes se atienden en el orden en que llegan. El problema con la cola es que, debido a la falta de recursos (tenedores), no siempre es posible atender al filósofo cuya solicitud que ocupa la cabeza de la cola. Describa un escenario con cinco filósofos y una cola, pero considerando que no se garantiza el servicio a los filósofos cuya solicitud está más abajo en la cola, aunque haya recursos disponibles.

7.11.4 [10]<7.4> ¿Se soluciona el problema anterior si las solicitudes al camarero se implementan como intentos de solicitud periódicos? Razone la respuesta.

Ejercicio 7.12

Considere las siguientes organizaciones de la CPU:

CPU SS: Un microprocesador superescalar de dos núcleos con emisión fuera de orden, dos unidades funcionales (FU) y un solo hilo en cada núcleo.

CPU MT: Un procesador con ejecución multihilo de grano fino que permite que instrucciones de dos hilos se estén ejecutando concurrentemente (es decir, hay dos unidades funcionales), aunque sólo se puede emitir una instrucción de un hilo en cada ciclo.

CPU SMT: Un procesador SMT que permite que instrucciones de dos hilos se estén ejecutando concurrentemente (es decir, hay dos unidades funcionales), y en cada ciclo se pueden emitir instrucciones de uno o de ambos hilos.

Suponga que se van a ejecutar dos hilos X e Y con las siguientes operaciones:

Hilo X	Hilo Y
A1 – tarda dos ciclos en ejecutarse	B1 – sin dependencias
A2 – depende del resultado de A1	B2 – conflicto con B1 por una unidad funcional
A3 – conflicto con A2 por una unidad funcional	B3 – sin dependencias
A4 – depende del resultado de A2	B4 – depende del resultado de B2

Finalmente, suponga que todas las instrucciones tardan un ciclo a menos que se especifique lo contrario o se ven afectadas por riesgos (dependencia o conflicto).

7.12.1 [10]<7.5> ¿Cuántos ciclos serán necesarios para ejecutar los dos hilos en una CPU SS? ¿Cuántos ciclos se pierden debido a riesgos?

7.12.2 [10]<7.5> ¿Cuántos ciclos serán necesarios para ejecutar los dos hilos en una CPU MT? ¿Cuántos ciclos se pierden debido a riesgos?

7.12.3 [10]<7.5> ¿Cuántos ciclos serán necesarios para ejecutar los dos hilos en una CPU SMT? ¿Cuántos ciclos se pierden debido a riesgos?

Ejercicio 7.13

Actualmente se está utilizando software de virtualización para reducir los costes de administración de los servidores de altas prestaciones. Empresas como VMWare, Microsoft e IBM han desarrollado una amplia gama de productos de virtualización. El concepto general, descrito en el capítulo 5, consiste en introducir una capa hipervisora entre el hardware y el sistema operativo que permite que varios sistemas operativos compartan el mismo hardware. Así, la capa hipervisora es la responsable de la asignación de CPU y de memoria y del manejo de servicios típicamente manejados por el sistema operativo, por ejemplo, la E/S.

La virtualización proporciona al software de aplicaciones y al sistema operativo invitado, una visión abstracta del hardware. Esto requiere repensar como se diseñarán los sistemas multinúcleo y multiprocesador del futuro para poder compartir las CPUs y la memoria entre varios sistemas operativos.

7.13.1 [30]<7.5> Seleccione dos hipervisores entre los disponibles actualmente en el mercado, y compare y contraste como hacen la virtualización y como manejan el hardware (CPU y memoria).

7.13.2 [15]<7.5> Discuta qué cambios pueden ser necesarios en las CPU multinúcleo futuras para atender mejor la demanda de recursos de estos sistemas. Por ejemplo, ¿puede la ejecución multihilo jugar un papel importante para mitigar la competencia por los recursos computacionales?

Ejercicio 7.14

Se quiere ejecutar el siguiente lazo de la forma más eficiente posible. Se dispone de dos computadores diferentes, un MIMD y un SIMD.

```
for (i=0; i<2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

7.14.1 [10]<7.6> Muestre la secuencia de instrucciones MIPS que se ejecutarían en cada CPU de un sistema MIMD con cuatro CPUs. ¿Cuál es la aceleración de este computador MIMD?

7.14.2 [20]<7.6> Escriba un programa ensamblador, con su propia extensión SIMD para MIPS, para ejecutar el lazo en un computador SIMD de ancho 8 (es decir, ocho unidades funcionales paralelas SIMD).

Ejercicio 7.15

Un computador sistólico es un ejemplo de sistema MISD. Está formado por una red segmentada de elementos de procesamiento de datos. Los elementos de procesamiento no necesitan contador de programa, porque la ejecución comienza cuando llegan los datos. Los sistema sistólicos con reloj trabajan de una forma sincronizada, en la que cada procesador alterna fases de computación y de comunicación.

7.15.1 [10]<7.6> Considere implementaciones propuestas de sistemas sistólicos (pueden encontrarse en la web o en publicaciones técnicas) e intente programar el lazo del ejercicio 7.14 con este modelo MISD. Analice las dificultades que encuentre.

7.15.2 [10]<7.6> Discuta las similitudes y diferencias entre sistemas SIMD y MISD en términos de paralelismo de datos.

Ejercicio 7.16

Suponga que se quiere ejecutar el lazo DAXP en ensamblador MIPS de la página 651 en la GPU NVIDIA 8800 GTX descrita en este capítulo. En ese problema se supone que todas las operaciones matemáticas se hacen en punto flotante de precisión simple (renombrando el lazo como SAXP) y que los ciclos necesarios para la ejecución de las instrucciones son los mostrados en la tabla.

Cargas	Almacenamientos	Add.s	Mult.s
4	1	2	5

7.16.1 [20]<7.7> Indique como construiría las tramas para explotar los ocho núcleos disponibles en un único multiprocesador durante la ejecución del lazo SAXP.

Ejercicio 7.17

Descargue las herramientas y el sistema de desarrollo de CUDA *(CUDA Toolkit and SDK)* de www.nvidia.com/object/cuda_get.html. Asegúrese de utilizar la versión "emurelease" (modo emulación) del código (no es necesario disponer de hardware de NVIDIA). Pruebe los ejemplos proporcionados con el SDK y confirme que se ejecutan en el emulador.

7.17.1 [90]<7.7> Utilizando los patrones de los ejemplos del SDK como punto de partida, escriba un programa CUDA para las siguientes operaciones sobre vectores:

1. $a - b$ (resta vector-vector)
2. a · b (producto escalar de vectores)

El producto escalar de dos vectores $a = [a_1, a_2, ..., a_n]$ y $b = [b_1, b_2, ..., b_n]$ se define como:

$$\mathrm{a} \cdot \mathrm{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Haga el código para cada operación y verificar que los resultados son correctos.

7.17.2 [90]<7.7> Si se dispone de una GPU, haga un análisis completo de las prestaciones del programa, examinando el tiempo de cálculo en la GPU y en la CPU para varios de tamaños de vectores diferentes. Explique los resultados observados.

Ejercicio 7.18

AMD ha anunciado recientemente un microprocesador que integra una GPU y varios núcleos x86 en un único chip, aunque con relojes diferentes para cada núcleo. Esto es un ejemplo de un sistema multiprocesador heterogéneo, que será una realidad comercial en un futuro próximo. Uno de los puntos clave del diseño será tener una comunicación rápida entre GPU y CPU. Actualmente, la GPU y la CPU residen en chips diferentes, con comunicaciones lentas, pero esto va a cambiar con la arquitectura *Fusion* de AMD. En este momento se utilizan varios canales PCI express (hasta 16) para facilitar la intercomunicación. También Intel se está introduciendo en este campo con su arquitectura *Larrabee*, y está considerando utilizar su tecnología de interconexión *QuickPath*.

7.18.1 [25]<7.7> Compare el ancho de banda y la latencia de estas dos tecnologías de interconexión.

Ejercicio 7.19

La figura 7.9b muestra una topología de red *n*-cubo de orden 3 que conecta ocho nodos. Una característica atractiva de esta topología de red es su capacidad para proporcionar conectividad aún en presencia de conexiones rotas.

7.19.1 [10]<7.8> Exprese mediante una ecuación cuántas conexiones del *n*-cubo pueden fallar pero garantizando todavía una conexión activa para conectar cualquier par de nodos.

7.19.2 [10]<7.8> Compare la resistencia a los fallos de un topología *n*-cubo y una red completamente conectada con el mismo número de nodos. Dibuje una comparación de la fiabilidad, como una función del número de conexiones que pueden fallar en las dos topologías.

Ejercicio 7.20

La evaluación de las arquitecturas es un campo de estudio que implica identificar cargas de trabajo representativas que se ejecutan en una plataforma específica con el fin de ser capaces de comparar objetivamente las prestaciones de los sistemas. En este ejercicio comparamos dos clases de programas de prueba: Whetstone y PARSEC. Se puede acceder a todos los programa de forma gratuita en internet. Seleccione un programa de PARSEC. Ejecute varias copias de Whetstone frente los programas de prueba PARSEC en cualquiera de los sistemas descritos en la sección 7.11.

7.20.1 [60]<7.8> ¿Cuáles son las diferencias entre las dos cargas de trabajo cuando se ejecutan en esos sistemas multinúcleo?

7.20.2 [60]<7.8> En términos del modelo Roofline, ¿cuál es la dependencia de los resultados obtenidos con la ejecución de estos programas de prueba con la cantidad de sincronización y compartición presentes en la carga de trabajo?

Ejercicio 7.21

En los cálculos con matrices dispersas, la latencia de la jerarquía de memoria es mucho más que un factor. Las matrices dispersas no presentan la localidad en los datos que típicamente se encuentran en otras operaciones sobre matrices. En consecuencia, se han propuesto varias representaciones nuevas para estas matrices.

Una de las primeras representaciones de las matrices dispersas es el formato Yale *(Yale Sparse Matrix Format)*. Una matriz dispersa M de dimensión $m \times n$ se almacena por filas con tres vectores unidimensionales. Sea R el número de elementos de M distintos de cero; se puede construir un vector A de longitud R que contiene todos los elementos de M distintos de cero (ordenados de izquierda a derecha y de arriba a abajo). Se construye también un segundo vector IA de longitud m+1 (un elemento por fila más 1) tal que $IA(i)$ tiene el índice del primer elemento distinto de cero de la fila i. La fila i de la matriz M ocupa desde $A(IA(i))$ a $A(IA(i+1)–1)$. El tercer vector, JA, también de longitud R, contiene el índice de la columna de cada elemento de A.

7.21.1 [15]<7.9> Considere la matriz dispersa X de más abajo y escriba un programa C para almacenar esta matriz en el formato Yale.

```
Fila 1 [0, 0, 0, 0, 0]
Fila 2 [0, 0, 0, 0, 0]
Fila 3 [8, 0, 0, 0, 6]
Fila 4 [0, 1, 8, 7, 0]
Fila 5 [7, 0, 0, 0, 0]
```

7.21.2 [10]<7.9> Suponiendo que los elementos de la matriz X son números punto flotante de precisión simple, calcule el espacio de memoria necesario para almacenar la matriz en formato Yale.

7.21.3 [15]<7.9> Multiplique la matriz X por la matriz Y, mostrada a continuación

```
[9, 8, 7, 199, 2]
```

Obtenga el tiempo de ejecución. Compare el tiempo de ejecución utilizando el formato Yale y sin utilizarlo.

7.21.4 [15]<7.9> ¿Puede encontrar una representación más eficiente (en términos de espacio y sobre coste computacional) para la matriz dispersa?

Ejercicio 7.22

En los sistemas futuros posiblemente se verán plataformas heterogéneas con diferentes CPUS. Ya hay sistemas similares en el campo de los sistemas empotrados, con un DSP de punto flotante y CPUs en un sistema multichip.

Suponga que que se tienen tres clases de CPU:

CPU A. Una CPU multinúcleo de velocidad moderada, con unidad de punto flotante, que puede ejecutar varias instrucciones por ciclo.

CPU B. Una CPU rápida de enteros de un solo núcleo (es decir, sin unidad de punto flotante), que puede ejecutar una instrucción por ciclo.

CPU C. Una CPU vectorial lenta, con unidades punto flotante, que puede ejecutar varias copias de la misma instrucción por ciclo.

Suponga que las siguientes frecuencias de reloj:

CPU A	CPU B	CPU C
1.5 GHz	3 GHz	500 MHz

La CPU A puede ejecutar dos instrucciones por ciclo, la CPU B una instrucción por ciclo y la CPU C ocho instrucciones por ciclo (aunque la misma instrucción). Suponga que todas las operaciones se ejecutan en un ciclo sin dependencias.

Las tres CPUs pueden hacer operaciones con enteros, aunque la CPU B no puede hacer operaciones de punto flotante directamente. El repertorio de instrucciones de las CPUs A y B es similar al repertorio de instrucciones MIPS; la CPU C sólo puede hacer operaciones de suma y resta punto flotante y cargas y almacenamientos de memoria. Las CPUs tienen acceso a una memoria compartida y el coste de sincronización es cero.

La carga de trabajo es la comparación de dos matrices X e Y de 1024 × 1024 números punto flotante, proporcionando como resultado el número de elementos en los que el valor en la matriz X es mayor que el valor en la matriz Y.

7.22.1 [10]<7.11> Indique cómo se puede dividir el problema entre las tres CPUs para obtener las mejores prestaciones.

7.22.2 [10]<7.11> ¿Qué tipo de instrucción se debería añadir al repertorio de instrucciones de la CPU C para mejorar las prestaciones?

Ejercicio 7.23

Suponga un sistema de cuatro núcleos *(quadcore)* que procesa transacciones de una base de datos a un ritmo de solicitudes por segundo mostrado en la tabla. Suponga también que cada transacción tarda, en promedio, una cantidad fija de tiempo en ser procesada. En la tabla se muestran pares de latencia y ritmo de procesamiento.

Latencia media de la transacción	Ritmo máximo de procesamiento de transacciones
1 ms	5000/seg
2 ms	5000/seg
1 ms	10 000/seg
2 ms	10 000/seg

Conteste a las siguientes preguntas para cada par de la tabla.

7.23.1 [10]<7.11> ¿Cuántas solicitudes se procesan de media en un instante de tiempo?

7.23.2 [10]<7.11> En un sistema con 8 núcleos, indique qué ocurre idealmente con la productividad del sistema (transacciones/segundo).

7.23.3 [10]<7.11> Analice por qué esta aceleración se obtiene muy raramente simplemente incrementando el número de núcleos.

Respuestas a las autoevaluaciones

$7.1, página 634. Falso. El paralelismo a nivel de trabajos puede ayudar a las aplicaciones secuenciales y las aplicaciones secuenciales pueden ejecutarse en un hardware paralelo, aunque esto exige más esfuerzo.
$7.2, página 638. Falso. Escalamiento débil puede compensar en una parte serie del programa que, de otro modo limitaría la escalabilidad.
$7.3, página 640. Falso. Dado que la dirección compartida es una dirección *física*, varios trabajos con espacios de direcciones *virtuales* propios pueden ejecutarse correctamente en un multiprocesador de memoria compartida.
$7.4, página 645. 1. Falso. El envío y la recepción de mensajes es una sincronización implícita, al igual que la compartición de datos. 2. Verdadero.
$7.5, página 648. 1. Verdadero. 2. Verdadero
$7.6, página 653. Verdadero.
$7.7, página 660. Falso. El precio de los DIMMs DRAM gráficos depende de su mayor ancho de banda.
$7.9, página 666. Verdadero. Probablemente necesitamos innovación en todos los niveles del hardware y el software para ganar la apuesta de la industria por la computación paralela.

A

APÉNDICE

GPUs para gráficos y cálculo

John Nicolls
Director de Arquitectura
NVIDIA

David Kirk
Científico jefe
NVIDIA

La imaginación es más importante que el conocimiento

Albert Einstein
On Science, 1930s

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

A.1 Introducción

Este apéndice se centra en la **GPU**, la omnipresente **unidad de procesamiento gráfico** (*graphics processing unit*) de cualquier PC, portátil, computador de sobremesa y estación de trabajo. En su concepción más básica, la GPU genera gráficos 2D y 3D, imágenes y vídeo que posibilitan el desarrollo de sistemas operativos basados en ventanas, interfaces gráficas de usuario, videojuegos, aplicaciones de formación y representación de imágenes visuales y vídeo. La GPU moderna que se describe en este apéndice es un multiprocesador con ejecución multihilo, con un número elevado de hilos, altamente paralelo y optimizado para **computación visual**. La GPU es un arquitectura unificada de gráficos y computación, que puede utilizarse como un procesador gráfico programable o un como sistema paralelo escalable, y que proporciona una interacción visual en tiempo real con objetos compuestos de gráficos, imágenes y vídeo. Los PCs y las videoconsolas combinan la GPU con una CPU para formar un **sistema heterogéneo**.

Unidad de procesamiento gráfico (GPU): procesador optimizado para gráficos 2D y 3D, vídeo, computación visual y visualización.

Computación visual: mezcla de procesamiento gráfico y computación que le permite interactuar visualmente con los objetos computados vía gráficos, imágenes y vídeo.

Sistema heterogéneo: sistema que combina diferentes clases de procesadores. Un PC es un sistema heterogéneo CPU-GPU.

Una breve historia de la evolución de la GPU

Hace quince años no había nada parecido a una GPU. El procesamiento de gráficos en un PC se realizaba con el controlador VGA (*Video Graphics Array*). El controlador VGA era simplemente un controlador de memoria y un generador de visualización conectado a una DRAM. En la década de los 90, la tecnología de semiconductores había avanzado lo suficiente para añadir más funciones en el controlador VGA. Así,

en 1997, empiezan a incorporarse algunas funciones de aceleración 3D, incluyendo hardware para configuración y **rasterización** de triángulos (cortar el triángulo en píxeles individuales) y proyección de texturas y sombreado (aplicación de "pegatinas" o patrones a los píxeles y mezclado de colores).

Rasterización: transformación de primitivas 2D y 3D en una imagen representada con un mapa de bits.

En el año 2000, el procesador gráfico, integrado en un único chip, incorporaba casi todas las funciones del sistema gráfico (*graphics pipeline*) tradicional de las estaciones de trabajo de gama alta y se había ganado un nombre que significaba mucho más que un simple controlador VGA. El término GPU se acuñó para dejar claro que el dispositivo gráfico era ya un procesador.

Con el paso del tiempo, y a medida que la lógica especializada en una función fija se reemplazaba por procesadores programables, la GPU fue ganando mayor programabilidad, manteniendo al mismo tiempo la organización básica del pipeline gráfico 3D. Además, los cálculos se hicieron más precisos, pasando de aritmética indexada a enteros y punto fijo, a punto flotante de precisión simple y, recientemente, a punto flotante de precisión doble. Como resultado, las GPUs se transformaron en procesadores masivamente paralelos con cientos de núcleos y miles de hilos.

Recientemente se han añadido instrucciones y hardware de memoria para dar soporte a lenguajes de programación de propósito general, y se han desarrollado entornos para permitir la programación de la GPU con lenguajes familiares, como C y C++. Esta innovación ha convertido a la GPU en un procesador con muchos núcleos, programable y de propósito general, aunque mantiene todavía algunos beneficios y limitaciones específicos.

Tendencias en las GPU

Las GPUs y sus controladores asociados implementan los modelos de procesamiento gráfico OpenGL y DirectX. OpenGL es un estándar libre para programación de gráficos 3D disponible para la mayoría de los computadores. DirectX es un conjunto de interfaces de programación multimedia de Microsoft, incluyendo Direct3D para gráficos 3D. Como estas **interfaces de programación de aplicaciones (*application programming interfaces*, API)** tienen un comportamiento perfectamente definido, es posible construir aceleradores hardware efectivos para las funciones de procesamiento gráfico definidas en las APIs. Esta es una de las razones (además del incremento de la densidad de los dispositivos) que explican el desarrollo de nuevas GPUs cada 12 a 18 meses que doblan las prestaciones de la generación anterior con las aplicaciones existentes.

Interfaz de programación de aplicaciones (API): conjunto de definiciones de funciones y estructuras de datos que proporcionan una interfaz a una biblioteca de funciones.

Este aumento de las prestaciones de la GPU posibilita el desarrollo de nuevas aplicaciones que no eran posibles anteriormente. La intersección del procesamiento gráfico y la computación paralela da lugar a un nuevo paradigma de computación gráfica conocido como computación visual. Con este paradigma, se reemplazan grandes partes del modelo tradicional secuencial del pipeline gráfico hardware por elementos programables para vértices, geometría y píxeles. La computación visual en una GPU moderna combina el procesamiento gráfico y la computación paralela de forma novedosa, permite la implementación de nuevos algoritmos para gráficos y abre la puerta a aplicaciones de procesamiento paralelo totalmente nuevas en GPUs de altas prestaciones.

Sistemas heterogéneos

Aunque podría decirse que la GPU es el procesador más potente y con mayor paralelismo en un PC típico, ciertamente no es el único procesador. La CPU, en la actua-

lidad multinúcleo y en un futuro próximo con un elevado número de núcleos, es un complemento, fundamentalmente serie, de la GPU masivamente paralela con un número elevado de núcleos. Estos dos procesadores juntos configuran un sistema multiprocesador heterogéneo.

Las mejores prestaciones en muchas aplicaciones se obtienen al utilizar la CPU y la GPU. Este apéndice ayudará a comprender como y cuando hacer la mejor partición del trabajo entre estos dos procesadores, que cada vez incorporan mayor paralelismo.

La GPU evoluciona a un procesador paralelo escalable

Las GPUs han evolucionado funcionalmente desde los controladores VGA, de capacidad limitada y totalmente hardware, hasta los procesadores paralelos programables. Esta evolución ha avanzado gracias a cambiar el pipeline gráfico lógico (basado en APIs) para incorporar elementos programables y también gracias a hacer las etapas hardware del pipeline menos especializadas y más programables. Finalmente, se fusionaron elementos programables dispares del pipeline en un conjunto unificado de muchos procesadores programables.

En la serie de GPUs de la generación GeForce 8, el procesamiento de vértices, píxeles y geometría se lleva a cabo en el mismo tipo de procesador. Esta unificación permite una escalabilidad elevada y aumentar la productividad total del sistema. También se mejora el equilibrio efectivo de la carga, porque cualquier función puede utilizar el conjunto de procesadores al completo. En el otro extremo del espectro, un conjunto de procesadores puede actualmente construirse con muy pocos procesadores, porque todas las funciones pueden ejecutarse en el mismo procesador.

¿Por qué CUDA y la computación en una GPU?

Este conjunto uniforme y escalable de procesadores invita al desarrollo de un nuevo modelo de programación para la GPU. La gran potencia de cálculo punto flotante disponible hace que la GPU sea muy atractiva para la resolución de problemas no gráficos. Dado el elevado grado de paralelismo y la escalabilidad del conjunto de procesadores de la GPU, el modelo de programación para computación de propósito general debe expresar de forma directa el paralelismo masivo y permitir una ejecución escalable.

El término acuñado para referirse al uso de la GPU en computación vía un lenguaje y una API de programación paralela, sin utilizar una API tradicional para gráficos y el modelo de pipeline gráfico, es **computación en GPU (*GPU computing*)**. Este uso es diferente del enfoque previo conocido como **computación de propósito general en la GPU (*general purpose computation on GPU, GPGPU*)**, que se basa en la programación de la GPU utilizando APIs gráficas y el pipeline gráfico para tareas no gráficas.

Compute Unified Device Architecture (CUDA) es un modelo de programación paralela escalable y una plataforma software para la GPU y otros procesadores paralelos, que permite programar la GPU en C o C++ sin necesidad de utilizar las APIs e interfaces gráficas de la GPU. El modelo de programación CUDA es SPMD (programa único, varios datos o *Single-Program, Multiple-Data*), en el que el programador escribe un único código que se ejecuta en varios hilos en paralelo en los procesadores de la GPU. De hecho, CUDA proporciona también herramientas para la programación de sistemas con varias CPUs. Es decir, CUDA es un entorno de programación paralela para sistemas heterogéneos.

Computación en GPU: utilización de la GPU para computación vía un lenguaje y una API de programación paralela.

GPGPU: utilización de la GPU para computación de propósito general vía las API tradicionales de gráficos y el pipeline gráfico.

CUDA: modelo de programación escalable y un lenguaje basado en C/C++. Es una plataforma de programación paralela para GPUs y CPUs multinúcleo.

La GPU unifica los gráficos y la computación

Actualmente, y gracias a CUDA y a las capacidades computacionales de la GPU, es posible utilizar la GPU como un procesador gráfico o como un procesador de cálculo al mismo tiempo, y combinar estos usos en aplicaciones de computación visual. La microarquitectura de la GPU puede verse de dos formas diferentes: primero, como la implementación de las APIs gráficas programables, y segundo, como un conjunto de procesadores masivamente paralelo programable en C o C++ con CUDA.

Aunque los procesadores de la GPU están unificados, esto no quiere decir que todos los hilos de programa SPMD tengan que ser iguales. La GPU puede ejecutar programas de sombreado gráfico de geometrías, vértices y píxeles y, al mismo tiempo, ejecutar hilos de programa en CUDA.

Así, la GPU es realmente una arquitectura multiprocesador versátil, capaz de ejecutar una gran variedad de tareas. Es excelente en computación visual y gráfica, porque ha sido específicamente diseñada para estas aplicaciones; pero también es excelente en muchas otras aplicaciones de propósito general, que son "primas hermanas" de las aplicaciones gráficas en el sentido de que tienen un gran paralelismo y una estructura regular del problema. En general, es adecuada para aplicaciones con paralelismo de datos (véase capítulo 7), en particular con grandes conjuntos de datos, y menos adecuadas para aplicaciones más pequeñas y menos regulares.

Aplicaciones de computación visual en GPU

La computación visual incluye las aplicaciones gráficas tradicionales y muchas aplicaciones nuevas. El ámbito original de la GPU ha pasado de ser "cualquier cosa con píxeles" a incluir ahora muchos problemas sin píxeles pero con regularidad en las computaciones y/o en los datos. Las GPUs son efectivas en gráficos 2D y 3D, que es para lo que fueron diseñadas, y cualquier fallo a la hora de obtener buenas prestaciones con estas aplicaciones sería fatal. En estos casos se utiliza la GPU en su "modo gráfico", accediendo a través de APIs gráficas, OpenGL y DirectX. Un ejemplo de estas aplicaciones son los juegos, que aprovechan la capacidad de procesamiento gráfico 3D.

Otras aplicaciones importantes para la GPU, más allá de los gráficos 2D y 3D, son el procesamiento de imágenes y el procesamiento de vídeo. Estas aplicaciones pueden implementarse utilizando las APIs gráficas o como programas de cálculo, utilizando CUDA para programar la GPU en modo computación. De este modo, el procesamiento de imágenes es simplemente otro conjunto de programas con paralelismo de datos. En la medida en que el acceso a los datos es regular y existe localidad, el programa será eficiente y, en la práctica, es una buena aplicación para las GPUs. El procesamiento de vídeo, especialmente la codificación y la decodificación (compresión y descompresión de acuerdo con algunos algoritmos estándar) es también bastante eficiente.

La gran oportunidad para las aplicaciones de computación visual en la GPU está en "romper el pipeline gráfico". Las primeras GPUs implementaban sólo algunas APIs gráficas, aunque con prestaciones muy elevadas. Esto sería maravilloso si la API soportase las operaciones que queremos hacer; en caso contrario, la GPU no serviría para acelerar nuestra tarea porque la funcionalidad no se podía cambiar. Ahora, con la llegada de la computación en GPU y CUDA, estas GPUs pueden programarse para implementar un pipeline virtual diferente, simplemente escribiendo un programa en CUDA que describa

el flujos de datos y de computaciones deseado. De este modo, ahora todas las aplicaciones son posibles, lo que estimulará nuevos enfoques en computación visual.

A.2 Arquitecturas del sistema de la GPU

En esta sección revisamos las arquitecturas del sistema de la GPU más utilizadas en la actualidad. Discutiremos las configuraciones del sistema, las funciones y servicios de la GPU, las interfaces de programación estándares y la arquitectura básica interna de la GPU.

Arquitectura del sistema heterogéneo CPU–GPU

La arquitectura de un sistema de computación heterogéneo formado por una GPU y una CPU puede describirse a alto nivel con dos características primarias: primero, cuántos subsistemas funcionales o chips se utilizan y cuáles son las tecnologías y las topologías de interconexión; segundo, qué subsistemas de memoria están disponibles para estos subsistemas funcionales. Véanse en el capítulo 6 los sistemas de E/S y los conjuntos de chips para PC.

El PC histórico (hacia 1990)

La figura A.2.1 muestra un diagrama de bloques de alto nivel de un PC clásico hacia 1990. El puente norte (véase capítulo 6) contiene interfaces de alto ancho de banda que conectan la CPU, la memoria y el bus PCI. El puente sur consta de interfaces y dispositivos clásicos: bus ISA (audio, LAN), controlador de interrupciones, contro-

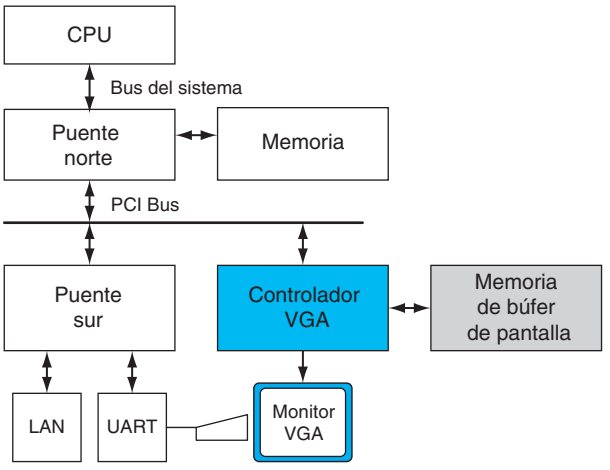


FIGURA A.2.1 PC histórico. El controlador VGA se encarga de la visualización de los gráficos de la memoria de búfer de pantalla.

lador de DMA y temporizador. En este sistema, la visualización está controlada por un subsistema de búfer de pantalla, adjunto al bus PCI, llamado VGA (*video graphics array*). Los subsistemas gráficos con elementos de procesamiento (GPU) no existían en el panorama de los PC de 1990.

PCI Expres (PCI-e): interconexión de E/S de sistemas estándar que utiliza conexiones punto-a-punto. El número de canales y el ancho de banda de las conexiones es configurable.

La figura A.2.2 ilustra dos configuraciones todavía en uso actualmente. Se caracterizan por disponer de GPU y CPU separadas, en chips distintos, con sus respectivos subsistemas de memoria. En la figura A.2.2a, con una CPU Intel, la GPU está conectada con una conexión **PCI Express** 2.0 de 16 canales con un ritmo de transferencia pico de 16 GB/s (pico de 8 GB/s en cada dirección). De forma similar, en la figura A.2.2b, con una CPU AMD, la GPU está asociada al conjunto de chips, también con una conexión PCI Express del mismo ancho de banda. En ambos casos, la GPU puede

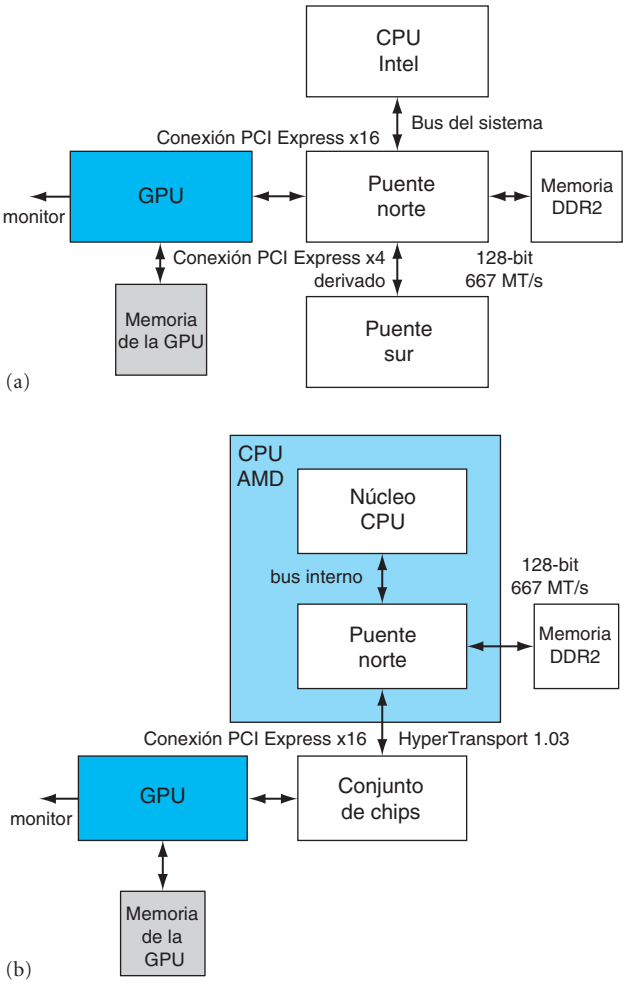


FIGURA A.2.2 PCs actuales con CPUs Intel y AMD. Véase en el capítulo 6 la explicación de los componentes e interconexiones de la figura.

acceder a la memoria de la CPU y viceversa, aunque con un ancho de banda menor que el acceso a su memoria. En el sistema AMD, el puente norte o controlador de memoria, está integrado con la CPU.

Un sistema de **arquitectura de memoria unificada(UMA)** es una variación de bajo coste de estos sistemas. Utiliza sólo la memoria de la CPU, eliminando la memoria de la GPU. Incorporan GPUs de prestaciones bajas, puesto que sus prestaciones están limitadas por el ancho de banda del sistema de memoria y la latencia de los accesos a memoria, mientras que la memoria de la GPU proporciona un elevado ancho de banda y baja latencia.

Arquitectura de memoria unificada (UMA): Arquitectura de un sistema en el que la CPU y la GPU comparten un sistema de memoria común.

Una variación de altas prestaciones es un sistema con varias GPUs, normalmente dos o cuatro, trabajando en paralelo, con sus monitores conectados en cadena *(daisy-chain)*. Un ejemplo es el sistema multi-GPU SLI (*scalable link interconnect*) de NVIDIA, diseñado para juegos y estaciones de trabajo de altas prestaciones.

En la siguiente categoría, los sistemas integran la GPU con el puente norte (Intel) o el conjunto de chips (AMD) con o sin memoria gráfica.

El mantenimiento de la coherencia en un sistema con un espacio de direcciones compartido y caches se explica en el capítulo 5. En un sistema con CPUs y GPUs hay varios espacios de direcciones. Las GPUs acceden a sus propia memoria local física y a la memoria física del sistema de la CPU mediante la utilización de direcciones virtuales que se traducen en una MMU de la GPU. La tabla de páginas de la GPU está gestionada por el núcleo del sistema operativo. El acceso a una página física del sistema se realiza con una transacción PCI Express coherente o no coherente, en función de un atributo en la tabla de páginas de la GPU. La CPU accede a la memoria local de la GPU utilizando un rango de direcciones (también llamado apertura) del espacio de direcciones PCI Express.

Consolas de videojuegos

La arquitectura de las consolas, tales como la Sony PlayStation 3 y la Xbox 360 de Microsoft, se parece mucho a las arquitecturas de PC que acabamos de describir. Están diseñadas para que sus prestaciones y funcionalidades no cambien durante su período de vida, que puede durar cinco años o más. Durante este tiempo, el sistema puede ser reimplementado muchas veces para explotar las ventajas de procesos de fabricación de dispositivos electrónicos más avanzados y proporcionar, así, las mismas capacidades a un coste cada vez más reducido. No necesitan que sus subsistemas se expandan y actualicen en la forma en que lo hacen los PCs y, en consecuencia, los principales buses de sistema internos están hechos a media en lugar de utilizar alguno de los estándares existentes.

Interfaces y controladores de la GPU

Actualmente, las GPUs se conectan a la CPU mediante una PCI Express, pero en generaciones anteriores utilizaban **AGP**. Las aplicaciones gráficas hacen llamadas a funciones de la API de OpenGL [Segal y Akeley, 2006] o de DirectX [Microsoft DirectX specifications] que utilizan la GPU como un coprocesador. Las APIs envían órdenes, programas y datos a la GPU a través de controladores de dispositivos gráficos optimizados para una GPU particular.

AGP: versión extendida del bus original PCI de E/S, que proporciona hasta ocho veces el ancho de banda del bus PCI original en una ranura. Principalmente se utilizaba para conectar subsistemas gráficos en PCs.

El pipeline lógico para gráficos

El pipeline lógico para gráficos se describe en la sección A.3. La figura A.2.3 muestra las principales etapas de procesamiento. Las etapas programables (etapas de sombreado de vértices, geometría y píxeles) se han resaltado.

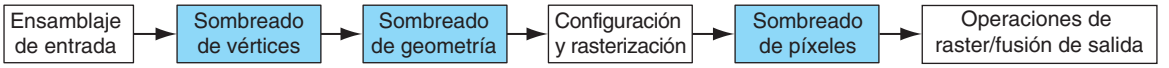


FIGURA A.2.3 El pipeline lógico para gráficos. Las etapas de sombreado gráfico programables están en azul y los bloques con función fija en blanco.

Proyección del pipeline gráfico en los procesadores GPU unificados

La figura A.2.4 muestra como se proyecta el pipeline gráfico, formado por etapas programables independientes, en un conjunto distribuido de procesadores.

Arquitectura básica de la GPU unificada

La GPU unificada está compuesta por un conjunto paralelo de muchos procesadores programables. Se unifica el procesamiento del sombreado de vértices, geometría y píxeles con la computación paralela en el mismo procesador, a diferencia de las GPUs anteriores que tenían procesadores separados y especializados para cada tipo de procesamiento. El conjunto de procesadores programables está altamente integrado con procesadores de propósito específico (o función fija) para filtrado de texturas, rasterización, operaciones de raster, anti-aliasing, compresión, descompresión, visualización, decodificación de vídeo y procesamiento de vídeo de alta definición. Aunque los procesadores de aplicación específica superan de forma significativa a los procesadores programables en términos de prestaciones absolutas cuando hay limitaciones de área, coste o potencia, nos centraremos en los procesadores programables.

Si las comparamos con las CPUs multinúcleo, las GPUs con muchos núcleos tienen un punto de partida diferente para el diseño de la arquitectura, se centran en la ejecución

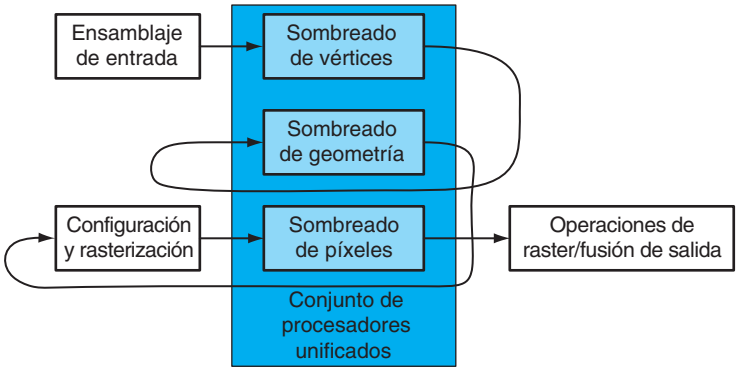


FIGURA A.2.4 Proyección del pipeline lógico en un conjunto de procesadores. Las etapas de sombreado programable se ejecutan en el conjunto de procesadores unificados y los datos del pipeline lógico para gráficos recirculan entre los procesadores.

de muchos hilos paralelos en muchos núcleos. Como consecuencia de la utilización de muchos núcleos más sencillos y de las optimizaciones para aprovechar el paralelismo de datos entre los grupos de hilos, la mayor parte de los transistores del chip están dedicados a cálculo, en detrimento de las caches integradas en el mismo chip y otros componentes.

Conjunto de procesadores

Un conjunto de procesadores GPU unificados consta de muchos núcleos organizados en multiprocesadores con ejecución multihilo. La figura A.2.5 muestra una GPU con un conjunto de 112 procesadores de *streaming* (SP) organizados en 14 multiprocesadores de *streaming* con ejecución multihilo (SM). Cada núcleo SP puede gestionar en hardware 96 hilos concurrentes y sus estados correspondientes. El procesador está conectado con cuatro particiones DRAM de 64 bits de ancho a través de una red de interconexión. Cada SM tiene ocho núcleos SP, dos unidades de funciones especiales (SFU), caches para instrucciones y constantes, una unidad de emisión multihilo y una memoria compartida. Ésta es la arquitectura Tesla básica implementada en la GeForce 8800 de NVIDIA. Es una arquitectura unificada en la que los programas tradicionales de gráficos para sombreado de vértices, geometría y píxeles se ejecutan en los SM unificados y sus núcleos SP, al igual que los programas de computación.

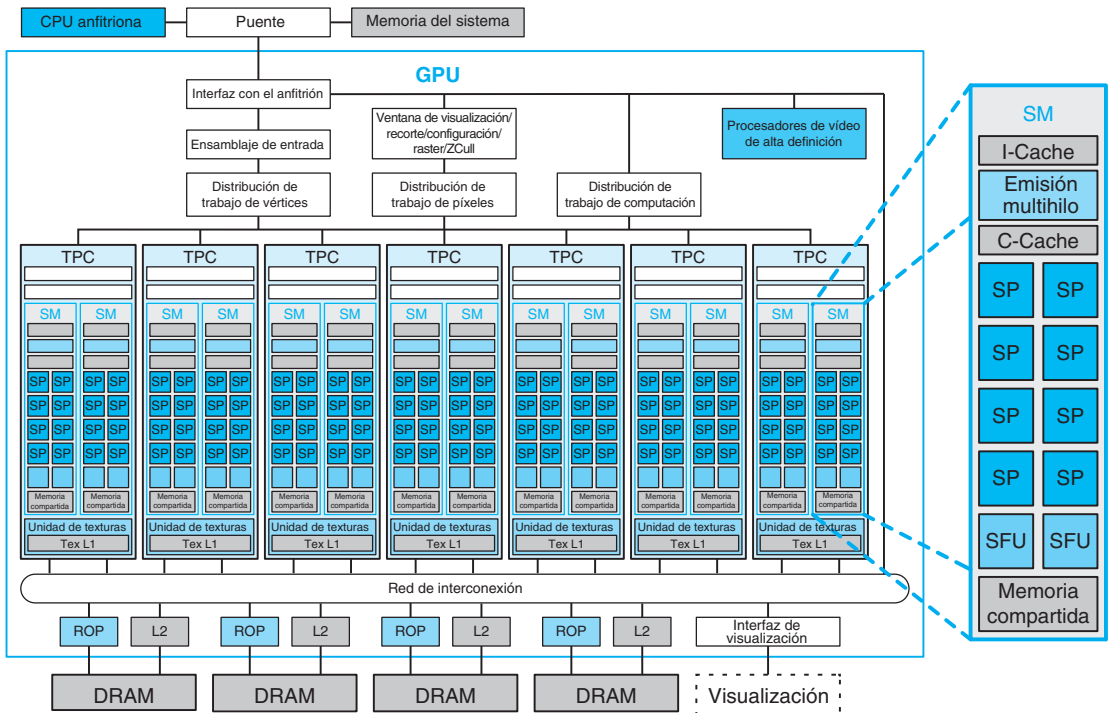


FIGURA A.2.5 Arquitectura básica de la GPU unificada. Ejemplo de GPU con 112 procesadores de *streaming* (SP) organizados en 14 multiprocesadores de *streaming* (SM); los núcleos tienen ejecución multihilo. Es la arquitectura Tesla básica de una GeForce 8800 de NVIDIA. Los procesadores se conectan con cuatro particiones DRAM de 64 bits de ancho a través de una red de interconexión. Cada SM tiene ocho núcleos SP, dos unidades de funciones especiales (SFU), caches para instrucciones y constantes, una unidad de emisión multihilo y una memoria compartida.

La arquitectura es escalable a configuraciones de GPU más pequeñas o más grandes mediante el escalado del número de multiprocesadores y de particiones de memoria. La figura A.2.5 muestra siete clústeres de dos SMs que comparten una unidad y una cache L1 de texturas. Dado un conjunto de coordenadas en el mapa de texturas, la unidad de texturas devuelve resultados filtrados a los SM. Como las regiones de filtrado de texturas de solicitudes consecutivas a menudo se solapan, se añade una pequeña cache L1 de texturas para reducir el número de solicitudes al sistema de memoria. El conjunto de procesadores está conectado a través de una red de interconexión de ancho de GPU *(GPU-wide)* con Procesadores de Operaciones de Raster (*Raster Operations Processor*, ROP), caches L2 de texturas, memorias DRAM externas y sistema de memoria. El número de procesadores y el número de memorias puede variarse para obtener GPUs adaptadas a diferentes niveles de prestaciones y diversos segmentos de mercado.

A.3 Programación de las GPU

La programación de las GPUs multiprocesador es cualitativamente diferente de la programación de otros multiprocesadores como las CPU multinúcleo. El número de hilos y el paralelismo a nivel de datos de la GPU es dos o tres órdenes de magnitud mayor que en las CPUs, escalando en 2008 a cientos de núcleos y decenas de miles de hilos concurrentes. Las GPUs siguen aumentando su paralelismo, doblándolo cada 12 o 18 meses, impulsado por el aumento de la densidad de integración de los circuitos de acuerdo con la ley de Moore [1965] y las mejoras de la eficiencia de la arquitectura. Para abarcar el amplio espectro de precios y prestaciones de los diferentes segmentos de mercado, se implementan diferentes GPUs con un número variable de procesadores e hilos. Aún así, los usuarios esperan que los juegos, los gráficos, el procesamiento de imágenes y las aplicaciones de computación funcionen en cualquier GPU, sin importar cuantos hilos paralelos se ejecutan o cuantos procesadores paralelos tiene, y esperan que las aplicaciones sean más rápidas una GPU más cara (con más hilos y núcleos). En consecuencia, los modelos de programación de la GPU y los programas de aplicaciones se diseñan para que escalen de forma transparente al usuario en un amplio espectro del paralelismo.

La fuerza motriz detrás del elevado número de hilos y núcleos paralelos en una GPU son las prestaciones de los gráficos en tiempo real, la necesidad de sintetizar escenas 3D complejas con alta resolución a un ritmo de al menos 60 cuadros *(frames)* por segundo. Por lo tanto, los modelos de programación escalables de los lenguajes de sombreado de gráficos, Cg (*C for graphics*) y HLSL (*high-level shading lenguage*), se han diseñado para explotar el elevado paralelismo inherente a estas aplicaciones mediante hilos paralelos independientes y el escalado a cualquier número de núcleos. De forma similar, el modelo de programación escalable de CUDA facilita que las aplicaciones paralelas de cómputo general generen un elevado número de hilos y escalen a cualquier número de núcleos paralelos de forma transparente a la aplicación.

En estos modelos de programación escalables, el programador escribe el código para un único hilo y la GPU ejecuta un gran número de hilos con este código en paralelo. De este modo, los programas escalan de forma transparente en un

amplio rango de paralelismo hardware. Este paradigma, que surgió de las APIs gráficas y de los lenguajes de sombreado que describen como sombrear un vértice o un píxel, se ha mantenido como un paradigma efectivo con el aumento del paralelismo y las prestaciones de las GPUs desde el final de la década de los 90.

En esta sección se describe de forma breve la programación de las GPUs para aplicaciones gráficas de tiempo real utilizando APIs gráficas y lenguajes de programación. A continuación se describe como programar las GPUs para aplicaciones de computación visual y de computación paralela general utilizando el lenguaje C y el modelo de programación CUDA.

Programación de gráficos en tiempo real

Las APIs han jugado un papel importante en el rápido y exitoso desarrollo de las GPUs y los procesadores gráficos. Hay principalmente dos APIs gráficas estándar: **OpenGL** y **Direct3D**, esta última es una de las interfaces de programación multimedia de DirectX de Microsoft. OpenGL, un estándar libre, fue propuesto y definido originalmente por Silicon Graphics, pero su desarrollo posterior y la extensión al estándar OpenGL [Segal y Akeley, 2006], [Kessenich, 2006] fue liderado por el consorcio industrial Kronos. Por otra parte, la definición y evolución de Direct3D, un estándar de facto, ha sido liderada por Microsoft con algunos otros socios. La estructura de OpenGL y Direct3D es similar, y ambos continúan su rápida evolución para adaptarse a los avances del hardware. Definen un pipeline lógico de procesamiento gráfico, que se proyecta en el hardware o en un procesador de la GPU, y modelos de programación y lenguajes para las etapas programables del pipeline.

OpenGL: API gráfica libre estándar.

Direct3D: API gráfica de Microsoft y sus asociados.

Pipeline lógico para gráficos

La figura A.3.1 muestra el pipeline lógico de Direct3D. El pipeline de OpenGL es similar. La API y el pipeline lógico proporcionan una estructura de flujo de datos continuo para las etapas de sombreado programables, mostradas en azul. La aplicación 3D envía una secuencia de vértices agrupados en primitivas geométricas —puntos, líneas, triángulos y polígonos— a la GPU. La etapa de ensamblaje de entrada recoge los vértices y las primitivas. La etapa de sombreado de vértices lleva a cabo el procesamiento por

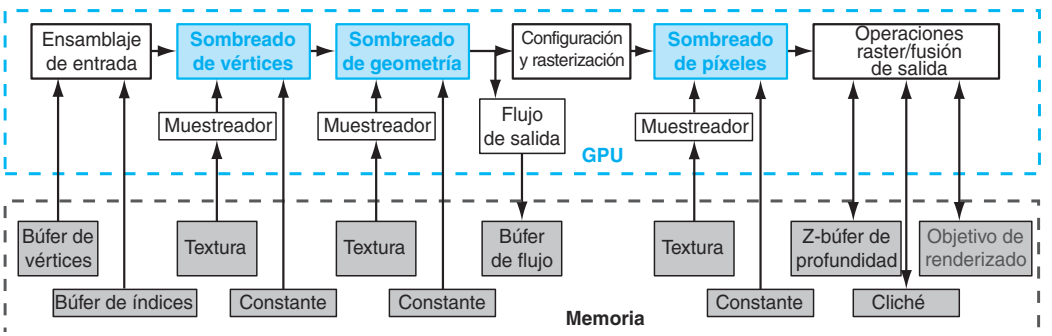


FIGURA A.3.1 Pipeline gráfico del Direct3D. Cada etapa del pipeline lógico se proyecta en el hardware de la GPU o en un procesador de la GPU. Las etapas de sombreado programables están coloreadas en azul, los bloques de función fija en blanco y los objetos de memoria en gris. Una etapa procesa un vértice, una primitiva geométrica o un píxel con un flujo de datos continuo.

vértice, incluyendo la transformación de la posición 3D en una posición de la pantalla y la iluminación del vértice para determinar su color. En la etapa de sombreado de geometría se realiza el procesamiento por primitiva y se añaden o eliminan primitivas. La unidad de configuración y rasterización genera fragmentos de píxeles (los fragmentos son contribuciones potenciales a un píxel) cubiertos por una primitiva geométrica. El programa de sombreado de píxeles realiza el procesamiento de los fragmentos, incluyendo interpolación de parámetros por fragmento, proyección de texturas y coloreado. Esta etapa hace un uso exhaustivo de búsquedas muestreadas y filtradas en grandes conjuntos 1D, 2D y 3D llamados **texturas**, utilizando coordenadas punto flotante interpoladas. Los programas de sombreado utilizan los accesos a texturas para planos, funciones, "pegatinas", imágenes y datos. La etapa de procesamiento de operaciónes raster (o fusión de salida) hace los chequeos de profundidad del Z-búfer y cliché, que pueden eliminar los fragmentos ocultos o reemplazar la profundidad del píxel por la profundidad del fragmento, y la operación de mezcla de colores, que combina el color del fragmento con el color del píxel y colorea el píxel con el color obtenido en la mezcla.

Textura: conjunto de datos 1D, 2D o 3D que permite búsquedas muestreadas y filtradas con coordenadas interpoladas.

Las APIs gráficas y el pipeline gráfico proporcionan la entrada, salida, objetos de memoria e infraestructura a los programas de sombreado que procesan los vértices, primitivas y fragmentos de píxeles.

Programas de sombreado de gráficos

Las aplicaciones de gráficos en tiempo real utilizan varios **programas de sombreado** para modelar la interacción de la luz con diferentes materiales y para renderizar escenas con iluminación y sombras complejas. Los **lenguajes de sombreado** se basan en un modelo de programación de flujo de datos o de *streaming*, en correspondencia con el pipeline gráfico lógico. Los programas de sombreado de vértices proyectan la posición de los vértices del triángulo en pantalla, modificando su posición, color y orientación. Típicamente, un hilo de sombreado de vértices recibe como entradas la posición del vértice (x, y, z, w) en representación punto flotante y calcula la posición en pantalla (x, y, z), también en punto flotante. Los programas de sombreado de geometría operan sobre primitivas geométricas (por ejemplo, líneas y triángulos) definidas por varios vértices, cambiándolas o generando primitivas adicionales. Los programas de sombreado de fragmentos de píxeles "sombrean" cada píxel, calculando su contribución roja, verde, azul y alfa (RGBA), en representación punto flotante, al color de la imagen renderizada. Los programas de sombreado (y las GPUs) utilizan aritmética punto flotante en todos los cálculos relacionados con el color del píxel para eliminar los efectos no deseados *(artifacts)* visibles al calcular el rango de valores de contribución de los píxeles en el renderizado de escenas con iluminación y sombras complejas y elevado rango dinámico. En los tres tipos de sombreado pueden ejecutarse en paralelo muchas instancias *(instances)* del programa, como hilos paralelos independientes, porque cada una opera sobre datos independientes, produce resultados independientes y no tiene efectos sobre las otras instancias. Además, esta independencia entre vértices, primitivas y píxeles posibilita que el mismo programa pueda ser ejecutado en GPUs de diferentes tamaños que procesan un número diferente de vértices, primitivas y píxeles en paralelo. De este modo, los programas gráficos escalan de forma transparente a GPUs con diferentes prestaciones y paralelismo.

Programa de sombreado: programa que opera con datos gráficos, como vértices o fragmentos de píxeles.

Lenguaje de sombreado: lenguaje de renderizado gráfico que habitualmente tiene un modelo de programación basado en flujo de datos o de *streaming*.

Los usuarios programan los tres tipos de sombreado con un mismo lenguaje de alto nivel. Los más utilizados son HLSL (*High-level shading language*) y Cg (*C for graphics*).

Tienen una sintaxis tipo C y disponen de un rico conjunto de funciones de biblioteca para operaciones matriciales, trigonométricas, de interpolación y de acceso y filtrado de texturas, pero no son lenguajes de computación general: actualmente no disponen de acceso a memoria general, punteros, ficheros de E/S y recursividad. HLSL y Cg suponen que el programa vive dentro de un pipeline gráfico lógico y, así, la E/S está implícita. Por ejemplo, el sombreado de un fragmento de píxeles asume que la normal geométrica y las coordenadas de texturas se han obtenido por interpolación de los valores en los vértices en etapas de función fija anteriores y que simplemente tiene que asignar un valor al parámetro de salida COLOR y pasarlo a la siguiente etapa donde se mezclará con un píxel en una posición (*x*, *y*) implícita.

La GPU crea un nuevo hilo independiente para el sombreado de cada vértice, cada primitiva y cada fragmento de píxeles. En videojuegos, la mayor parte de los hilos corresponden a programas de sombreado de píxel, porque típicamente hay entre 10 y 20 veces más fragmentos de píxeles que vértices; la iluminación y el sombreado complejos necesitan incluso mayores ratios entre hilos de sombreado de píxel y de vértices. El modelo de programación de sombreado de gráficos motivó que la arquitectura de la GPU sea capaz de ejecutar de forma eficiente miles de hilos de grano fino independientes en muchos núcleos paralelos.

Ejemplo de sombreado de píxel

Consideremos el siguiente programa de sombreado de píxel, escrito en Cg, que implementa la técnica de renderizado "*environment mapping*". Cada hilo de píxel recibe cinco parámetros, incluyendo coordenadas de la imagen de textura 2D en punto flotante, necesarias para determinar el color de la superficie, y un vector punto flotante 3D con la **reflexión de la dirección en la que se encuentra el observador en la superficie**. Los otros parámetros "uniformes" no cambian de un hilo a otro. El programa busca el color en dos imágenes de textura: un acceso a una textura 2D para el color de la superficie y un acceso a una textura 3D en un cubo (seis imágenes que corresponden a las caras del cubo) para obtener el color del mundo exterior correspondiente a la dirección de reflexión. Entonces, el color final de cuatro componentes (rojo, verde, azul, alfa) punto flotante se obtiene con una media ponderada, llamada "lerp", o mediante interpolación lineal.

```
void reflection(
  float 2                 texCoord          : TEXCOORD0,
  float 3                 reflection_dir    : TEXCOORD1,
  out float 4             color             : COLOR,
  uniform float           shiny,
  uniform sampler2D       surfaceMap,
  uniform samplerCUBE     envMap)
{
// Obtener el color de la superficie de una textura
   float4 surfaceColor = tex2D(surfaceMap, texCoord);
// Obtener color reflejado muestreando el cubo
   float4 reflectedColor = texCUBE(environmentMap,reflection_dir);
// La salida es la media ponderada de los dos colores
   color = lerp(surfaceColor, reflectedColor, shiny);
}
```

Aunque este programa sólo tiene tres líneas, activa mucho hardware de la GPU. Para cada acceso a la textura, el subsistema de texturas de las GPU hace varios accesos a memoria para analizar los colores de la imagen en la región vecina a las coordenadas de la muestra e interpolar el resultado final con aritmética punto flotante. La GPU con ejecución multihilo ejecuta miles de estos hilos ligeros de sombreado de píxeles, entrelazándolos para ocultar la latencia de acceso a la textura y a la memoria.

La visión del programador en Cg se centra en un único vértice, primitiva o píxel, que la GPU implementa con un hilo; el programa de sombreado escala de forma transparente para explotar el paralelismo a nivel de hilos en los procesadores disponibles. Aún siendo un lenguaje de aplicación específica, Cg proporciona un variado conjunto de tipos de datos, funciones de biblioteca y construcciones del lenguaje para implementar las técnicas de renderizado.

La figura A.3.2 muestra el resultado obtenido al renderizar la piel de un rostro con un programa de sombreado de píxel. La piel auténtica parece bastante diferente de la pintura color carne porque la luz rebota de aquí para allá muchas veces antes de reaparecer. En este sombreado complejo se han modelado tres capas de piel, cada una con un comportamiento de dispersión *(scattering)* bajo la superficie diferente, para dar a la piel profundidad visual y un aspecto traslúcido. La dispersión se modela con una convolución para suavizado de imágenes (*blurring convolution)* en un espacio de "texturas" plano, con el rojo más suavizado que el verde y éste más que el azul.

FIGURA A.3.2 Imagen renderizada en una GPU. Para dar profundidad visual y un aspecto traslúcido a la piel, el programa de sombreado de píxel modela tres capas de piel separadas, cada una con un comportamiento diferente de la dispersión (*scattering*) bajo la superficie. Se ejecutan 1400 instrucciones para renderizar los componentes rojo, verde, azul y alfa del color de cada píxel de la piel.

El programa de sombreado de Cg compilado ejecuta 1400 instrucciones para calcular el color de un píxel de la piel.

A medida que las GPUs han evolucionado para mejorar las prestaciones de los gráficos en tiempo real con la utilización de aritmética punto flotante y un ancho de banda de memoria muy elevado, han sido capaces de abordar otras aplicaciones paralelas además de la computación gráfica tradicional. Al principio, el acceso a esta potencia computacional era sólo posible formulando la aplicación como un algoritmo de renderizado de gráficos, pero esta alternativa de GPGPU a menudo era incómoda y limitante. Desde hace poco, el modelo de programación CUDA proporciona una vía mucho más sencilla para la explotación de las altas prestaciones escalables de punto flotante y el ancho de banda de las GPUs con el lenguaje de programación C.

Programación de aplicaciones de computación paralela

Algunas interfaces de programación de GPUs que se centran en la computación con paralelismo de datos en lugar de en los gráficos son CUDA, Brooks y CAL. CAL (*Compute Abstraction Layer*) es una interfaz de lenguaje ensamblador de bajo nivel para las GPUs de AMD. Brook es un lenguaje de *streaming* adaptado a las GPUs por Buck et al. [2004]. CUDA, desarrollado por NVIDIA [2007], es una extensión de C y C++ para programación paralela escalable de GPUs con muchos núcleos y CPUs multinúcleo. El modelo de programación de CUDA se describe a continuación; esta descripción está extraída de un artículo de Nickolls, Buck, Garland y Skadron [2008].

Con este nuevo modelo, la GPU destaca en la computación con paralelismo de datos y alta productividad, ejecutando tanto aplicaciones de computación de altas prestaciones como aplicaciones gráficas.

Descomposición de problemas con paralelismo de datos

Para proyectar grandes problemas de computación de forma efectiva en arquitecturas altamente paralelas, el programador o el compilador tiene que descomponer el problema en muchos problemas más pequeños que puedan ser resueltos en paralelo. Por ejemplo, el programador puede dividir un gran conjunto de datos, resultado de una computación, en bloques y cada bloque en elementos, de forma que los resultados incluidos en un bloque se puedan obtener de modo independiente y en paralelo con otros bloques y, además, los elementos de cada bloque puedan obtenerse en paralelo. La figura A.3.3 muestra la descomposición de un conjunto de datos en una malla 3×2 de bloques, donde cada bloque se ha descompuesto a su vez en un conjunto de 5×3 elementos. Esta descomposición paralela de dos niveles se proyecta de forma natural en la arquitectura de la GPU: los multiprocesadores paralelos calculan los bloques de resultados y los hilos paralelos calculan los elementos del resultado.

El programador escribe un programa que calcula una secuencia de mallas de resultados, dividiendo las mallas en bloques de resultados de grano grueso que pueden calcularse de forma independiente y en paralelo. El programa obtiene cada bloque con un conjunto de hilos paralelos de grano fino, dividiendo el trabajo entre los hilos de forma que cada hilo calcula uno o más elementos del resultado.

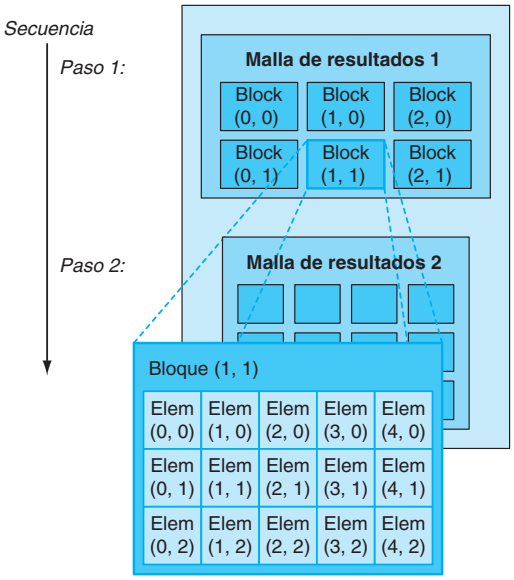


FIGURA A.3.3 Descomposición de los datos obtenidos como resultado de una computación en bloques de elementos que se van a procesar en paralelo.

Programación paralela escalable con CUDA

El modelo de programación paralela escalable de CUDA extiende los lenguajes C y C++ para explotar niveles elevados de paralelismo en aplicaciones generales en sistemas multiprocesador altamente paralelos, particularmente en GPUs. Desde los primeros experimentos se ha demostrado que muchos programas sofisticados pueden expresarse sin dificultad con CUDA utilizando abstracciones fácilmente comprensibles. Desde que NVIDIA hizo público CUDA en 2007, los programadores han desarrollado en poco tiempo programas paralelos escalables para un amplio rango de aplicaciones, incluyendo procesamiento de datos sísmicos, química computacional, álgebra lineal, resolución de matrices dispersas, ordenación, búsqueda, modelos de física y computación visual. Estas aplicaciones escalan de forma transparente a cientos de núcleos y miles de hilos concurrentes. Las GPUs de NVIDIA, con la arquitectura Tesla unificada para gráficos y computación (descrita en la secciones A.4 y A.7), pueden ejecutar programas C en CUDA y están ampliamente extendidas en portátiles, PCs, estaciones de trabajo y servidores. El modelo de programación CUDA se puede aplicar también a otras arquitecturas de procesamiento paralelo de memoria compartida, incluyendo las CPUs multinúcleo [Stratton, 2008].

En CUDA hay tres abstracciones clave —una *jerarquía de grupos de hilos*, *memorias compartidas* y *sincronización de barrera*— que dan una estructura paralela clara a un código C convencional para un hilo de la jerarquía. Estas tres abstracciones proporcionan paralelismo de datos de grano fino y paralelismo a nivel de hilos, anidado con paralelismo de datos de grano grueso y paralelismo a nivel de tareas. Las abstracciones guían al programador en la división del problema en subproblemas "gruesos" que pueden resolverse de forma independiente en paralelo y estos subproblemas se dividen en piezas más "finas" que pueden resolverse también en paralelo. El modelo de programación escala de forma

transparente a un número elevado de núcleos de procesamiento: un programa CUDA compilado se puede ejecutar en cualquier número de procesadores y únicamente el entorno de tiempo de ejecución necesita conocer el número de procesadores del sistema.

El paradigma CUDA

CUDA es una extensión mínima de los lenguajes de programación C y C++ en la que el programador escribe un programa serie que llama a **núcleos computacionales paralelos**. Estos núcleos computacionales pueden ser funciones simples o programas completos. El núcleo computacional se ejecuta mediante un conjunto de hilos paralelos. El programador organiza estos hilos en una jerarquía de bloques de hilos y mallas de bloques de hilos. Un **bloque de hilos** es un conjunto de hilos concurrentes que cooperan y se sincronizan mediante barreras se sincronización y accesos compartidos a un espacio de direcciones privado del bloque. Una **malla** es un conjunto de bloques de hilos que pueden ejecutarse de forma independiente y, por lo tanto, en paralelo.

Núcleo computacional: programa o función para un hilo y diseñado para ejecutarse con muchos hilos.

Bloque de hilos: conjunto de hilos concurrentes que ejecutan el mismo programa de hilo y pueden cooperan para obtener un resultado.

Malla: conjunto de bloques de hilos que ejecutan el mismo núcleo computacional.

Cuando se invoca un núcleo computacional, el programador especifica el número de hilos por bloque y el número de bloques que conforman la malla. A cada hilo se le asocia un número de *identificación de hilo* `threadIdx` en su bloque de hilos y a cada bloque, numerados `0,1,2,...`, `blockDim-1`, se le asocia un número de *identificación de bloque* `blockIdx` dentro de la malla. CUDA permite hasta 512 hilos por bloque. Según convenga, los bloques y las mallas pueden tener 1, 2 o 3 dimensiones a las que se accede con los campos índice `.x`, `.y`, `.z`.

Mostramos ahora un ejemplo sencillo de programación paralela: dados dos vectores punto flotante x e y de dimensión n, queremos calcular $y = ax + y$ para un cierto escalar a. Este es el núcleo computacional `SAXPY` de la biblioteca de álgebra lineal BLAS. La figura A.3.4 muestra el código C para SAXPY en un procesador serie y en paralelo con CUDA.

La declaración `__global__` indica que este procedimiento es el comienzo de un núcleo computacional. Los programas CUDA lanzan núcleos computacionales paralelos con la siguiente llamada, que tiene una sintaxis parecida a la de una función C, pero extendida:

```
kernel<<<dimGrid, dimBlock>>>(... lista de parámetros ...);
```

donde `dimGrid` y `dimBlock` son vectores de tres componentes de tipo `dim3` que especifican las dimensiones de la malla en bloques y las dimensiones de los bloques en hilos, respectivamente. Si no se especifican las dimensiones, se toma por defecto dimensión igual a uno.

En la figura A.3.4 se lanza una malla de n hilos que asocia un hilo a cada elemento de los vectores y sitúa 256 hilos en cada bloque. Cada hilo calcula un índice de elemento utilizando sus identificadores de hilo y bloque y, a continuación, hace los cálculos deseados en los elementos correspondientes de los vectores. Si comparamos la implementación serie y la paralela, veremos que son bastante parecidas y siguen un patrón común. El código serie consta de un lazo con iteraciones independientes y este tipo de lazos se pueden transformar de forma mecánica en núcleos computacionales paralelos: cada iteración del lazo se convierte en un hilo. Asignando un hilo a cada elemento de salida se evita la sincronización entre hilos en la escritura de los resultados en memoria.

C Iculo de y = ax + y con un lazo serie

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
   for (int i = 0; i < n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invocación del núcleo computacional SAXPY serie
saxpy_serial(n, 2.0, x, y);
```

C Iculo de y = ax + y en paralelo con CUDA

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   if (i<n)   y[i] = alpha*x[i] + y[i];
}

// Invocación del núcleo computacional SAXPY paralelo
(256 hilos por bloque)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURA A.3.4 Código C secuencial (arriba) frente a código paralelo en CUDA (abajo) para SAXPY (véase el capítulo 7). Los hilos paralelos de CUDA reemplazan al lazo serie; cada hilo calcula el mismo resultado que una iteración del lazo. El código paralelo calcula *n* resultados con *n* hilos en bloques de 256 hilos.

El código de un núcleo computacional CUDA es simplemente una función C para un hilo secuencial. Así, en general es muy sencillo escribir el código CUDA y típicamente es más sencillo que escribir código paralelo para operaciones vectoriales. El paralelismo se determina de forma clara y explícita especificando las dimensiones de la malla y de los bloques de hilos.

La ejecución paralela y la gestión de hilos es automática. La creación, planificación y terminación de hilos la realiza el sistema. Efectivamente, la GPU con arquitectura Tesla gestiona los hilos directamente en hardware. Los hilos de un bloque se ejecutan concurrentemente y se sincronizan con **barreras de sincronización** (llamada a `__syncthreads()`). Así se garantiza que ningún hilo del bloque puede continuar hasta que todos los hilos han llegado a la barrera. Una vez pasada la barrera, los hilos pueden ver las escrituras hechas por otros hilos del bloque antes de la barrera de sincronización. Así, los hilos del bloque se comunican con escrituras y lecturas en la memoria compartida por el bloque.

Barrera de sincronización: hilos esperan en la barrera de sincronización hasta que todos los hilos del bloque llegan a la barrera.

Los hilos de un bloque residen en el mismo procesador o multiprocesador físico, porque comparten la memoria y se sincronizan con barreras. Sin embargo, el número de bloques de hilos puede ser mucho mayor que el número de procesadores. El modelo de programación de hilos CUDA virtualiza el procesador y le da al programador la fle-

xibilidad de paralelizar con la granularidad más conveniente. La virtualización en hilos y bloques de hilos permite una descomposición intuitiva del problema, ya que el número de bloques puede estar determinado por el tamaño de los datos que se procesan en lugar de por el número de procesadores del sistema. Además, permite que el mismo programa CUDA escale a un gran número de núcleos de procesamiento.

Para gestionar la virtualización de los elementos de procesamiento y proporcionar escalabilidad, es necesario que los bloques de hilos de CUDA sean capaces de ejecutarse de forma independiente; los bloques deben poder ejecutarse en cualquier orden, en serie o en paralelo. Bloques diferentes no pueden comunicarse de forma directa, aunque pueden coordinarse mediante **operaciones de memoria atómicas** en la memoria global visibles a todos los hilos; por ejemplo, aumentando atómicamente los punteros de cola. Esta independencia permite que los bloques se planifiquen en cualquier orden y en cualquier número de núcleos, haciendo que el modelo CUDA sea escalable a un número arbitrario de núcleos y a varias arquitecturas paralelas. Además, contribuye a evitar la posibilidad de un bloqueo *(deadlock)*. Una aplicación puede ejecutar varias mallas, bien de forma independiente, bien de forma dependiente. Las mallas independientes se pueden ejecutar concurrentemente si hay suficientes recursos hardware disponibles. Las mallas dependientes se ejecutan secuencialmente, con una barrera entre núcleos computacionales implícita, garantizando que todos los bloques de la primera malla se completan antes de que comiencen su ejecución los bloques de la segunda malla dependiente.

Operación de memoria atómica: Una secuencia de lectura, modificación y escritura en memoria que se completa sin que haya ningún otro acceso a esta posición de memoria.

Los hilos pueden acceder a datos en varios espacios de memoria diferentes. Cada hilo en CUDA tiene una **memoria local** privada, que se utiliza para el almacenamiento de las variables privadas del hilo, que no tienen sitio en los registros del hilo, y también para los cuadros de la pila y el volcado de registos *(register spilling)*. Cada bloque de hilos tiene una **memoria compartida**, accesible por todos los hilos del bloque, que tiene la misma vida que el bloque. Finalmente, todos los hilos tienen acceso a una **memoria global**. Los programas declaran las variables en la memoria compartida y global con los especificadores `__shared__` y `__device__`. En una GPU con arquitectura Tesla, estos espacios de memoria corresponden a memorias físicamente separadas: la memoria compartida del bloque es una RAM de baja latencia integrada en el mismo chip del procesador, mientras que la memoria global reside en una DRAM rápida de la tarjeta gráfica.

Memoria local: Memoria privada local de cada hilo.

Memoria compartida: Memoria asociada a los bloques y compartida por todos los hilos del bloque.

Memoria global: memoria asociada a un aplicación y compartida por todos los hilos.

La memoria compartida es una memoria de baja latencia cercana al procesador, casi como una cache L1. Proporciona comunicaciones y compartición de datos de altas prestaciones entre los hilos del bloque. Como tiene la misma vida que su bloque de hilos correspondiente, típicamente el código inicializa los datos en variables compartidas, hace los cálculos con variables compartidas y copia los resultados a la memoria global. Los bloques de hilos de mallas secuenciales dependientes se comunican a través de la memoria global, mediante la lectura de entradas y la escritura de resultados.

La figura A.3.5 muestra un diagrama de niveles anidados de hilos, bloques de hilos y mallas de bloques de hilos. Además, muestra los niveles de compartición de memoria correspondientes: memorias locales, compartidas y globales para la compartición de datos en cada hilo, bloque o aplicación.

Un programa gestiona el espacio de memoria global visible a todos los núcleos computacionales mediante las llamadas al entorno de tiempo de ejecución de CUDA, `cudaMalloc()` y `cudaFree()`. Los núcleos computacionales pueden ejecutarse en un dispositivo físicamente separado, como ocurre cuando se ejecutan núcleos compu-

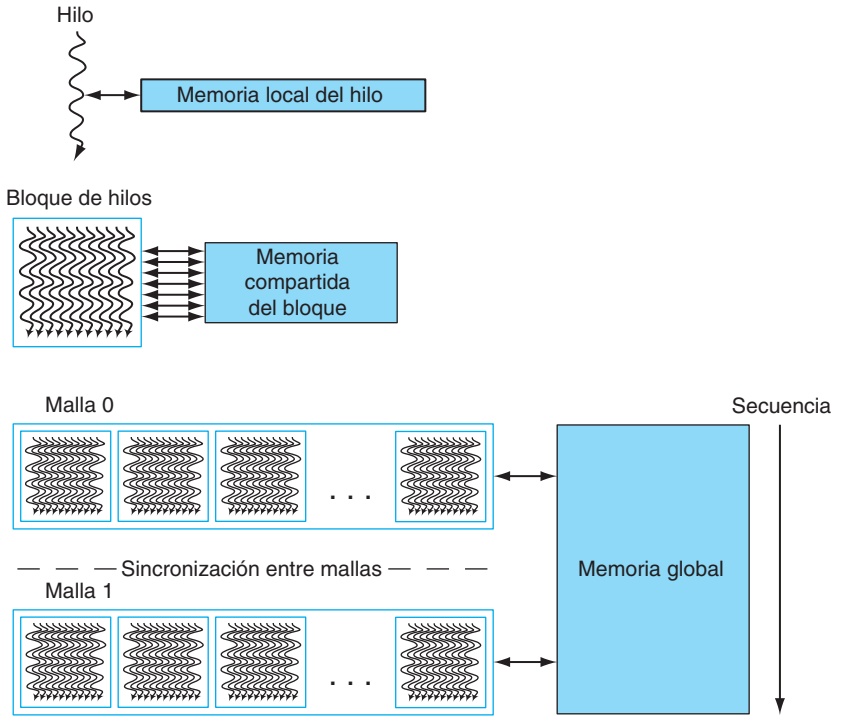


FIGURA A.3.5 Los niveles de granularidad anidados —hilo, bloque y malla— tienen niveles de compartición de memoria asociados: local, compartida y global. La memoria local del hilo es privada al hilo. Los hilos de un bloque comparten la memoria compartida del bloque. Todos los hilos comparten la memoria global de la aplicación.

tacionales en la GPU. Consecuentemente, la aplicación debe usar `cudaMemcpy()` para copiar datos entre el espacio del dispositivo y el sistema de memoria del anfitrión.

Programa único y múltiples datos (SPMD): estilo de programación paralela en el que todos los hilos ejecutan el mismo programa. Los hilos SPMD se coordinan con barreras de sincronización.

El modelo de programación CUDA tiene un estilo similar al modelo de programación de **programa único y múltiples datos (*Single-Program, Multiple-Data, SPMD*)**; expresa el paralelismo explícitamente y cada núcleo computacional se ejecuta en un número fijo de hilos. Sin embargo CUDA es más flexible que la mayoría de las implementaciones SPMD, porque cada llamada a un núcleo computacional crea de forma dinámica una nueva malla con el número adecuado de bloques de hilos e hilos para esta etapa de la aplicación. El programador puede usar el grado de paralelismo más conveniente para cada núcleo computacional, en lugar de utilizar el mismo número de hilos para todas las etapas de la computación. La figura A.3.6 muestra un ejemplo de una secuencia de código CUDA SPMD. Primero se inicia una instancia de `kernelF` en una malla 2D de 3 × 2 bloques donde cada bloque 2D consta de 5 × 3 hilos. A continuación se inicia una instancia de `kernelG` en una malla 1D de cuatro bloques 1D con seis hilos cada uno. Como `kernelG` depende de los resultados de `kernelF`, están separados por una barrera de sincronización entre núcleos computacionales.

Los hilos concurrentes de un bloque expresan el paralelismo de datos de grano fino y el paralelismo a nivel de hilos. Los bloques independientes de una malla

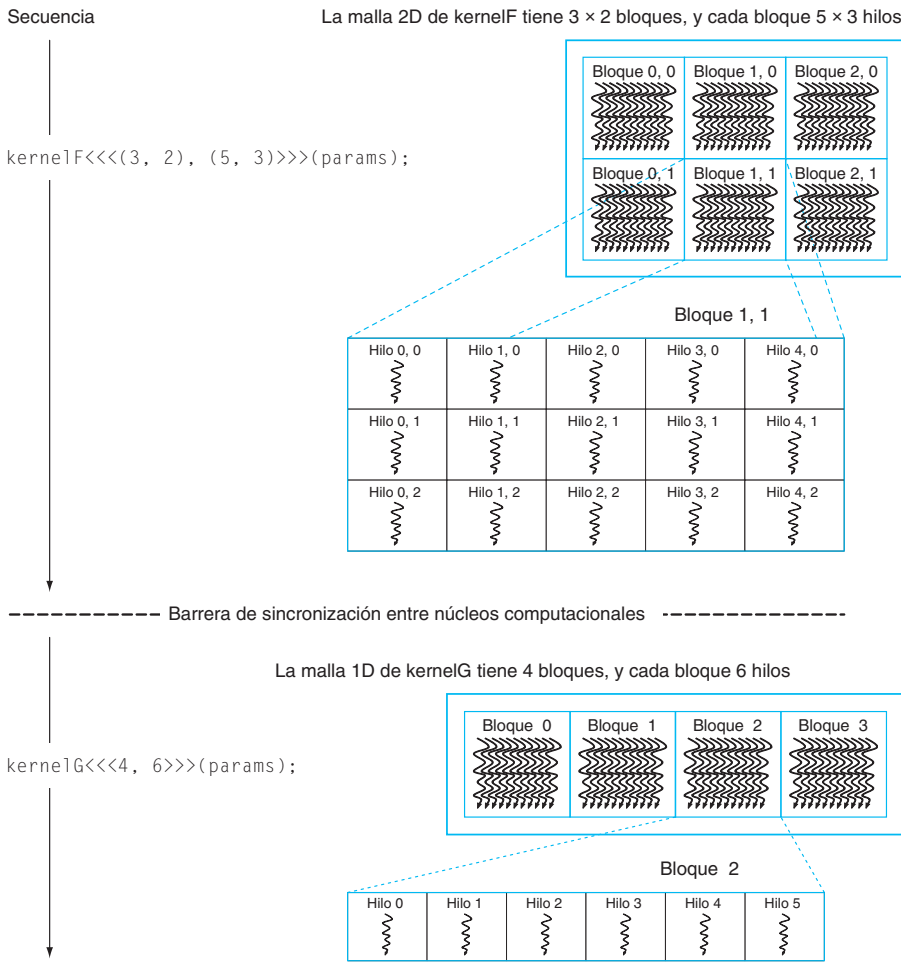


FIGURA A.3.6 Secuencia de inicio de una instancia un núcleo computacional *F* en una malla 2D de bloques de hilos 2D, una barrera de sincronización entre núcleos computacionales, seguido de un núcleo computacional G en una malla 1D de bloques de hilos 1D.

expresan el paralelismo de datos de grano grueso. Las mallas independientes expresan el paralelismo a nivel de tareas de grano grueso. Un núcleo computacional es simplemente un código C para un hilo en la jerarquía.

Restricciones

Por motivos de eficiencia y para simplificar su implementación, el modelo de programación CUDA tiene algunas restricciones. Los hilos y los bloques de hilos sólo se pueden crear invocando un núcleo computacional paralelo, no desde dentro de un núcleo computacional paralelo. Esto, junto con la necesaria independencia de los bloques de hilos, hace posible ejecutar programas CUDA con un planificador senci-

llo que introduce un sobrecoste en tiempo de ejecución mínimo. De hecho, la GPU Tesla implementa la gestión y planificación de hilos y bloques de hilos en hardware.

El paralelismo a nivel de tareas puede expresarse a nivel de bloques de hilos pero es difícil expresarlo dentro de un bloque porque la barrera de sincronización de hilos opera sobre todos los hilos del bloque. Para facilitar que los programas CUDA se puedan ejecutar en un número arbitrario de procesadores, no se permiten las dependencias entre bloques de la misma malla, los bloques deben ejecutarse de forma independiente. Como CUDA exige que los bloques sean independientes y permite que se ejecuten en cualquier orden, para hacer la combinación de resultados generados por varios bloques se debe lanzar un segundo núcleo computacional en una nueva malla de bloques (aunque los bloques pueden *coordinar* su actividad mediante operaciones de memoria atómicas en la memoria global visible a todos los hilos, por ejemplo, incrementando de forma atómica los punteros de cola).

Actualmente, las llamadas recursivas a funciones no están permitidas en CUDA. La recursividad es poco atractiva en núcleos computacionales masivamente paralelos, porque disponer de espacio de pila para las decenas de miles de hilos que pueden estar activos podría necesitar ingentes cantidades de memoria. Los algoritmos serie que normalmente se formulan mediante recursividad, como algunos algoritmos de ordenación, pueden ser implementados también con paralelismo de datos anidado, en lugar de con recursividad explícita.

Para dar soporte a un sistema heterogéneo que combina una CPU y una GPU, cada uno con su sistema de memoria propio, los programas CUDA deben copiar datos y resultados entre la memoria del anfitrión y la memoria del dispositivo. El sobrecoste de la interacción CPU-GPU y de la transferencia de datos se minimiza mediante la utilización de DMA para la transferencia de bloques e interconexiones rápidas. Los grandes problemas computacionalmente intensivos amortizan el sobrecoste mejor que problemas pequeños.

Implicaciones para la arquitectura

Los modelos de programación paralela para gráficos y computación han llevado a que la arquitectura de la GPU sea diferente de la arquitectura de la CPU. Las características de los programas de la GPU que más han influido sobre la arquitectura son los siguientes:

- *Uso masivo del paralelismo de datos de grano fino*: Los programas de sombreado describen como procesar un único píxel o vértice y los programas CUDA describen como calcular un resultado individual.
- *Modelo de programación con ejecución multihilo de muchos hilos*: Un hilo de sombreado procesa un píxel o un vértice y un hilo de un programa CUDA puede generar un único resultado. Una GPU tiene que crear y ejecutar millones de estos hilos por cuadro, con 60 cuadros por segundo.
- *Escalabilidad*: Los programas deben aumentar automáticamente sus prestaciones cuando se dispone de procesadores adicionales, sin necesidad de recompilarlos.
- *Computaciones punto flotante (o de enteros) intensivas*.
- *Soporte para computación de alta productividad*.

A.4 Arquitectura multiprocesador con ejecución multihilo

Las GPUs actuales son multiprocesadores compuestos de multiprocesadores y para dirigirse a los diferentes segmentos del mercado, se implementan con un número escalable de multiprocesadores. Además, cada multiprocesador soporta ejecución multihilo con un número elevado de hilos, lo que permite una ejecución eficiente de muchos hilos de grano fino de sombreado de píxeles y vértices. Una GPU de calidad básica dispone de dos a cuatro multiprocesadores, mientras que las GPU para los entusiastas de los videojuegos y para las plataformas de computación pueden tener docenas de multiprocesadores. En esta sección analizamos la arquitectura de uno de estos multiprocesadores con ejecución multihilo, una versión simplificada del multiprocesador de *streaming* (SM) Tesla de NVIDIA que se describe en la sección A.7.

¿Por qué utilizar un multiprocesador en lugar de varios procesadores independientes? El paralelismo de cada multiprocesador proporciona altas prestaciones y soporta ejecución multihilo a gran escala para los modelos de programación paralela de grano fino que se han descrito en la sección A.3. Los hilos de cada bloque se ejecutan juntos en el mismo multiprocesador para compartir datos. La arquitectura multiprocesador multihilo que describimos en esta sección consta de ocho núcleos de procesamiento escalares en un arquitectura altamente acoplada que pueden ejecutar un máximo de 512 hilos (el SM que se describe en la sección A.7 puede ejecutar un máximo de 768 hilos). Para tener una mayor eficiencia en área y potencia, algunas unidades complejas están compartidas entre los ocho núcleos, por ejemplo, la cache de instrucciones, la unidad de emisión de instrucciones y la memoria RAM compartida.

Ejecución multhilo a gran escala

Los procesadores de la GPU incorporan ejecución multhilo a gran escala para alcanzar los siguientes objetivos:

- Ocultar la latencia de las cargas de memoria y de la búsqueda de texturas en la DRAM.
- Dar soporte de paralelismo de grano fino a los modelos de programación de sombreado gráfico.
- Dar soporte de paralelismo de grano fino a los modelos de programación de computación.
- Virtualizar los procesadores físicos como hilos y bloques de hilos para proporcionar escalabilidad transparente.
- Simplificar el modelo de programación paralela para tener que escribir sólo un programa serie para un hilo.

La latencia de la búsqueda de texturas y de los accesos a memoria puede ser de varios cientos de ciclos del procesador, porque las GPUs tienen caches de datos pequeñas en lugar de las grandes caches para almacenar el conjunto de trabajo de las CPUs. Generalmente, una búsqueda requiere un acceso a la DRAM, con una latencia igual a la latencia de la DRAM más la latencia del circuito de interconexión y la latencia del búfer. La ejecución multihilo ayuda a ocultar esta latencia haciendo computación útil —mientras un hilo está esperando a que termine una carga o una búsqueda de una textura, el procesa-

dor está ejecutando otro hilo. Los modelos de programación de grano fino proporcionan literalmente miles de hilos independientes que pueden mantener el procesador ocupado a pesar de las largas latencias de los accesos a memoria de los hilos individuales.

Un programa de sombreado de vértices o píxeles es un programa para un único hilo que procesa un vértice o un píxel. De forma similar, un programa CUDA es un programa C para un único hilo que calcula un resultado. Los programas de gráficos y computación lanzan muchos hilos paralelos para hacer el renderizado de imágenes complejas o para el cálculo de conjuntos grandes de resultados. Para equilibrar dinámicamente la carga de trabajo de los hilos de sombreado de vértices y píxeles, cada multiprocesador ejecuta concurrentemente varios hilos diferentes y varios tipos de programas de sombreado.

Cada hilo en la GPU tiene sus propios registros, una memoria local privada, un contador de programas, un estado de la ejecución del hilo y puede ejecutar una vía del código de forma independiente. De este modo, admite el modelo de programación de vértices, primitivas y píxeles independientes de los lenguajes de sombreado gráfico y el modelo de programación de CUDA con C/C++. Para ejecutar de forma eficiente cientos de hilos "ligeros", la GPU multiprocesador incluye soporte hardware para ejecución multihilo: gestiona y ejecuta cientos de hilos concurrentes en hardware sin sobrecoste de planificación. Por otra parte, los hilos concurrentes de los bloques de hilos se sincronizan en las barreras con una sola instrucción. En resumen, la creación de hilos ligeros, la planificación de hilos con sobrecoste cero y la sincronización de barrera con una instrucción, proporcionan un soporte eficiente para el paralelismo de grano muy fino.

Arquitectura multiprocesador

Un multiprocesador unificado de gráficos y computación ejecuta programas de sombreado de vértices, geometrías y píxeles, y programas de computación paralela. Tal y como se muestra en la figura A.4.1, el multiprocesador que estamos tomando como ejemplo consta de ocho núcleos de procesamiento escalares (SP), cada uno con un gran banco de registros multihilo (*Register File*, RF), dos unidades para funciones especiales (*Special Functions Units*, SFU), una unidad de emisión de instrucciones multihilo, una cache de instrucciones, una cache de sólo lectura para constantes y una memoria compartida.

La memoria compartida de 16KB almacena los búferes de datos gráficos y los datos de computación compartidos. Las variables CUDA que residen en la memoria compartida se definen como `__shared__`. Los hilos de vértices, geometría y píxeles tienen búferes de entrada y salida independientes, de forma que las cargas de trabajo entran y salen independientemente de la ejecución del hilo; esto permite ejecutar la carga de trabajo del pipeline gráfico lógico en el multiprocesador varias veces, tal como se ha mostrado en la sección A.2.

Los núcleos SP tienen unidades escalares de aritmética punto flotante y de enteros, que ejecutan la mayoría de las instrucciones, y soporte hardware para la ejecución multihilo de hasta 64 hilos. Los SP están segmentados y ejecutan una instrucción escalar por hilo y por ciclo de reloj, que tiene una frecuencia entre 1.2 GHz y 1.6 GHz dependiendo de la GPU. Además, tienen un gran banco de 1024 registros de propósito general de 32 bits (RF), que se reparten entre los hilos asignados a un SP. Normalmente, un programa necesita entre 16 y 64 registros por hilo. Así, el SP puede ejecutar concurrentemente muchos hilos que usan pocos registros o menos hilos que utilizan muchos registros. El

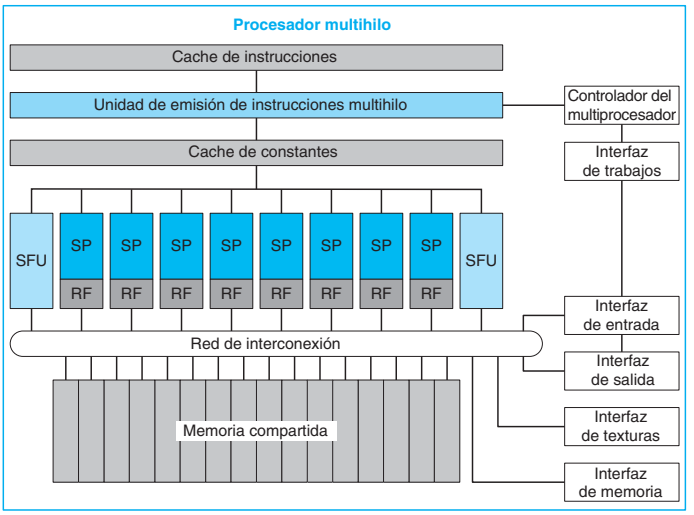


FIGURA A.4.1 Multiprocesador multihilo con ocho núcleos de procesamiento escalar (SP). Los SP tienen un gran banco de registros multihilo (RF) y comparten la unidad de emisión de instrucciones multihilo, la cache de constantes, dos unidades de funciones especiales (SFU), la red de interconexión y una memoria compartida multibanco.

compilador optimiza la asignación de registros para equilibrar el coste del volcado de registros (*register spilling*) frente al coste de tener pocos hilos. Los programas de sombreado de píxeles utilizan normalmente 16 o menos registros, por lo tanto un SP puede estar ejecutando hasta 64 hilos de sombreado de píxeles para ocultar las elevadas latencias asociadas a la búsqueda de la texturas. Por otra parte, habitualmente un programa CUDA compilado necesita 32 registros por hilo, limitando la capacidad de cada SP a 32 hilos y, por lo tanto, en este ejemplo de multiprocesador, un núcleo computacional sólo podría tener 256 hilos por bloque en lugar de su máximo de 512 hilos.

Las SFUs, que también están segmentadas, ejecutan instrucciones de cálculo de funciones especiales y realizan la interpolación de los atributos del píxel a partir de los atributos de los vértices de la primitiva. Estas instrucciones pueden ejecutarse concurrentemente con instrucciones de los SPs. Las SFU se describen más adelante.

Las instrucciones de búsqueda de texturas se ejecutan en la unidad de texturas, a la que se accede a través de la interfaz de texturas. Por otra parte, las instrucciones de carga, almacenamiento y acceso atómico a la memoria utilizan la interfaz de memoria. Estas instrucciones pueden ejecutarse concurrentemente con las instrucciones de los SPs. Los accesos a la memoria compartida hacen uso de la red de interconexión de baja latencia situada entre ésta y los procesadores SP.

Instrucción única y múltiples hilos (*Single-Instruction, Multiple-Threads*, SIMT)

El multiprocesador utiliza un arquitectura **Instrucción única y múltiples hilos (*Single-Instruction, Multiple-Threads*, SIMT)** para gestionar y ejecutar de forma eficiente cientos de hilos de varios programas diferentes. Crea, gestiona, planifica y ejecuta hilos concurrentes en grupos de hilos paralelos llamados *tramas (warps)*. El término **trama**

Instrucción única y múltiples hilos (*Single-Instruction, Multiple-Threads*, SIMT): arquitectura de procesador que envía una instrucción a varios hilos independientes en paralelo.

Trama: conjunto de hilos paralelos que ejecutan la misma instrucción en una arquitectura SIMT.

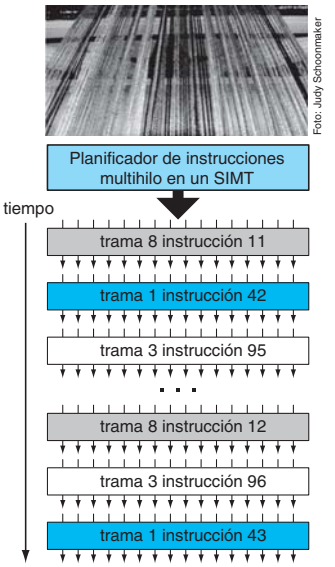


FIGURA A.4.2 Planificación de tramas multihilo SIMT. El planificador selecciona una trama lista para ejecutarse y emite una instrucción de forma sincronizada con los hilos que componen la trama. Como las tramas son independientes, el planificador puede seleccionar una trama diferente cada vez.

procede de los tejidos, la primera tecnología de hilos paralelos. La figura A.4.2 muestra una malla de hilos paralelos saliendo de un telar. El multiprocesador que usamos como ejemplo utiliza tramas de 32 hilos y ejecuta cuatro hilos en cada uno de los ocho núcleo SP en cuatro ciclos. El multiprocesador SM Tesla que se describe en la sección A.7 tiene también tramas de 32 hilos paralelos y ejecuta cuatro hilos por núcleo SP para alcanzar buenos rendimiento con abundantes hilos de píxel y de computación. Los bloques de hilos constan de una o más tramas.

Este multiprocesador SIMT de ejemplo gestiona un conjunto de 16 tramas, un total de 512 hilos. Los hilos paralelos individuales que componen una trama son todos del mismo tipo y comienzan en la misma dirección de programa pero, por otra parte, pueden ejecutar de forma diferente las instrucciones de salto y seguir por vías del programa diferentes. Cada vez que se emite una instrucción, la unidad de emisión de instrucciones multihilo SIMT selecciona una trama que tenga la siguiente instrucción lista para ejecutarse y emite esta instrucción en los hilos activos de esta trama. Por lo tanto, la instrucción SIMT se envía de forma síncrona a los hilos paralelos activos de la trama; algunos hilos pueden estar inactivos debido a la ejecución independiente de los saltos o a la predicación. Cada procesador escalar SP de este multiprocesador ejecuta una instrucción para cuatro hilos de la trama utilizando cuatro ciclos de reloj, obteniendo así la relación 4:1 entre los hilos de la trama y los núcleos.

La arquitectura del procesador SIMT es similar a los diseños SIMD, que envían una instrucción a varios canales de datos, pero difieren en que el SIMT envía una instrucción a varios hilos paralelos independientes, en lugar de simplemente a varios canales de datos. Una instrucción de un procesador SIMD controla un vector de varios canales de datos conjuntamente, mientras que una instrucción de un procesador SIMT controla un hilo

individual y la unidad de emisión de instrucciones SIMT emite la instrucción a una trama de hilos paralelos independientes. El procesador SIMT encuentra el paralelismo de datos entre hilos en tiempo de ejecución, de forma análoga a como un procesador superescalar encuentra el paralelismo a nivel de instrucciones también en tiempo de ejecución.

La eficiencia y las prestaciones de un procesador SIMT son máximas cuando todos los hilos de la malla siguen la misma vía de ejecución. Si los hilos de una malla toman vías diferentes en los saltos condicionales que dependen de los datos, se serializa la ejecución de cada vía de salida del salto y, una vez que terminan todos las vías, los hilos convergen otra vez a la misma vía de ejecución. Cuando las vías son de la misma longitud, la eficiencia de un bloque de código if-else es del 50%. El multiprocesador utiliza una pila de sincronización de saltos para gestionar la divergencia y convergencia de los hilos. Las tramas se ejecutan de forma independiente a su velocidad máxima sin importar si están ejecutando vías de código comunes o disjuntas. Como resultado, la eficiencia y flexibilidad de las GPUs SIMT en los saltos es mucho mayor que en las GPUs previas, y sus mallas son mucho más estrechas que el ancho de los SIMD de las GPUs anteriores.

En contraste con las arquitecturas vectoriales SIMD, la arquitectura SIMT permite que el programador escriba el código con paralelismo a nivel de hilos para un hilo individual, así como código con paralelismo de datos para muchos hilos coordinados. Desde el punto de la corrección del programa, el programador puede ignorar los atributos de ejecución SIMT de las mallas; sin embargo, teniendo en cuenta que los hilos de una trama raramente divergen, se pueden obtener mejoras sustanciales de las prestaciones. En la práctica, esto es análogo al papel de las líneas en la caches tradicionales: el tamaño de la línea de la cache se puede ignorar a la hora del diseño para un funcionamiento correcto, pero debe tomarse en cuenta si se quieren obtener unas prestaciones pico.

Ejecución de tramas SIMT y divergencia

El enfoque SIMT para la planificación de tramas independientes es más flexible que la planificación implementada en anteriores arquitecturas de la GPU. Una trama consta de varios hilos paralelos independientes del mismo tipo: vértices, geometrías, píxeles o computación. La unidad básica del sombreado de fragmentos de píxeles es un cuadrado de 2×2 píxeles, que se implementa con cuatro hilos de sombreado de píxel. El controlador del multiprocesador empaqueta el cuadrado en una trama y de forma similar agrupa los vértices y primitivas en tramas y empaqueta los hilos de computación en una trama. Un bloque de hilos consta de una o más tramas. La arquitectura SIMT comparte la búsqueda de instrucciones y la unidad de emisión entre los hilos paralelos de una trama, pero para maximizar las prestaciones se necesita una trama de hilos activos completa.

Este multiprocesador unificado planifica y ejecuta varios tipos de tramas simultáneamente, permitiendo procesar de forma concurrente tramas de vértices y píxeles. El planificador trabaja a una frecuencia menor que la frecuencia del reloj, porque hay cuatro canales de hilos por núcleo de procesamiento. Durante un ciclo de planificación, se selecciona una trama para ejecutar una instrucción SIMT, tal como se muestra en la figura A.4.2. La instrucción de la trama emitida se ejecuta como cuatro conjuntos de ocho hilos en cuatro ciclos de procesador. El procesador segmentado tiene una latencia de varios ciclos de reloj para completar cada instrucción. Pero si el número de tramas activas por el número de ciclos de reloj por trama es mayor que la latencia del procesador segmentado, el programador no tiene que preocuparse de la latencia del procesador.

De este modo y para este multiprocesador, la planificación de turno rotatorio (*round-robin*) de ocho tramas tiene un período de 32 ciclos entre instrucciones consecutivas de la misma trama. Si el programa puede mantener 256 hilos activos por multiprocesador, se pueden ocultar latencias de hasta 32 ciclos. Sin embargo, si hay pocas tramas activas, la profundidad del procesador segmentado puede causar paradas en el procesador.

Un reto desafiante es la implementación de la planificación de tramas con sobrecoste cero para una mezcla dinámica de diferentes tramas y tipos de programas. El planificador debe seleccionar una trama cada cuatro ciclos para emitir una instrucción por ciclo de reloj y por hilo, equivalente a un IPC igual a 1 en cada núcleo de procesamiento. Como las tramas son independientes, las únicas dependencias posibles están entre instrucciones secuenciales de la misma trama. El planificador utiliza la técnica del marcador de dependencias en registros para seleccionar las tramas con hilos activos y preparadas para ejecutar una instrucción. Realiza una priorización de estas tramas y selecciona la de mayor prioridad para la emisión. La priorización tiene en cuenta el tipo de trama, el tipo de instrucción y el deseo de ser imparcial con todas las tramas activas.

Gestión de hilos y bloques de hilos

El controlador del multiprocesador y la unidad de emisión de instrucciones gestionan los hilos y los bloques de hilos. El controlador arbitra los accesos a los recursos compartidos, incluyendo la unidad de texturas, la memoria y la E/S. Para cargas de trabajo de gráficos, crea y gestiona tres tipos de hilos gráficos de forma concurrente: vértices, geometrías y píxeles. Cada uno de estos tipos tienen vías de entrada y salida independientes. El controlador acumula y empaqueta cada una de estos tipos de entradas en una trama SIMT de hilos paralelos que ejecutan el mismo programa. Es decir, asigna una trama libre, asigna registros a los hilos de la trama y lanza la ejecución de la trama en el multiprocesador. Cada programa tiene que declarar sus necesidades registros por hilo y el controlador lanza una trama sólo cuando puede asignar los registros demandados por los hilos de la trama. Cuando todos los hilos de la trama terminan, el controlador desempaqueta los resultados y libera los recursos y registros asociados a la trama.

Conjunto de hilos cooperativos (*cooperative threads array*, CTA): conjunto de hilos concurrentes que ejecutan el mismo programa de hilo y cooperan para obtener un resultado. Un CTA de la GPU implementa un bloque de hilos de CUDA.

El controlador crea **conjuntos de hilos cooperativos (*cooperative threads array*, CTA)**; cada conjunto implementa un bloque de hilos de CUDA como una o más tramas de hilos paralelos. El CTA se crea sólo cuando el controlador puede crear todos las tramas y asignar todos los recursos del CTA. Además de hilos y registros, el CTA necesita que se le asigne memoria compartida y barreras de sincronización. El programa declara las capacidades que necesita y el controlador espera a tener disponibles todos los recursos solicitados antes de lanzar el CTA. Sólo entonces crea las trama del CTA, y las crea a la velocidad de la planificación de tramas, de modo que el programa del CTA comienza a ejecutarse de forma inmediata aprovechando las prestaciones máximas del multiprocesador. El controlador vigila cuando terminan todos los hilos del CTA para liberar los recursos compartidos del CTA y los recursos de las tramas.

Instrucciones de hilos

Un procesador SP de hilos ejecuta instrucciones escalares de hilos individuales, a diferencia de las arquitecturas vectoriales de GPUs anteriores que, en los programas de sombreado de vértices y píxeles, ejecutaban instrucciones sobre vectores de cuatro componentes. Efectivamente, los programas de procesamiento de vértices calculan las cua-

tro componentes (x, y, z, w) de los vectores y los programas de sombreado de píxeles calculan las cuatro componentes (rojo, verde, azul, alfa) del vector de color. Sin embargo, los programas de sombreado son cada vez más largos y más escalares y cada vez es más difícil ocupar completamente incluso dos de los cuatro componentes de la arquitectura vectorial de las GPUs anteriores. En efecto, la arquitectura SIMT hace la paralelización teniendo en cuenta 32 hilos de píxel independientes, en lugar de paralelizar los cuatro componentes del vector para cada píxel. Como resultado, el código de los programas C/C++ de CUDA para cada hilo es predominantemente escalar. Las GPUs anteriores utilizaban empaquetamiento de vectores (por ejemplo, combinación de subvectores para aumentar la eficiencia) pero esto complicaba sobremanera la planificación hardware y el compilador. Las instrucciones escalares son más sencillas y más fáciles para el compilador. Las instrucciones de texturas siguen siendo vectoriales, la entrada es el vector de coordenadas de la fuente y el resultado es un vector de color filtrado.

Los compiladores de lenguajes de alto nivel de computación y de gráficos generan un código intermedio con instrucciones ensamblador (por ejemplo, las instrucciones vectoriales de Direct3D o las instrucciones escalares de PTX) que se optimizan y se traducen a microinstrucciones binarias de la GPU, soportando así GPUs diferentes con microinstrucciones con formatos binarios distintos. El repertorio de instrucciones PTX (*parallel thread execution*) de NVIDIA [2007] proporciona una ISA estable a los compiladores y compatibilidad entre varias generaciones de GPUs con arquitecturas del repertorio de instrucciones parecidas, que han cambiado sólo por evolución. El optimizador expande fácilmente las instrucciones vectoriales de Direct3D a varias microinstrucciones binarias escalares. Por el contrario, las instrucciones escalares de PTX se traducen casi uno-a-uno con microinstrucciones binarias escalares, aunque algunas instrucciones PTX pueden necesitar expandirse en varias microinstrucciones y varias instrucciones PTX pueden juntarse en una microinstrucción binaria. Como las instrucciones ensamblador intermedias utilizan registros virtuales, el optimizador analiza las dependencias de datos y asigna los registros reales; además, elimina el código redundante, junta instrucciones cuando es posible y optimiza los puntos de divergencia y convergencia de los saltos SIMT.

Arquitectura del repertorio de instrucciones (ISA)

La ISA (*Instruction Set Architecture*) que se describe a continuación es una versión simplificada de la ISA PTX de la arquitectura Tesla, un repertorio de instrucciones escalar basado en registros con operaciones punto flotante, de enteros, lógicas, conversiones, funciones especiales, control de flujo, acceso a memoria e instrucciones de texturas. La figura A.4.3. muestra una lista de las instrucciones de hilo de la GPU PTX básica; para más detalles, véanse las especificaciones de la PTX de NVIDIA [2007]. El formato de las instrucciones es:

```
opcode.type d, a, b, c;
```

siendo `d` el destino del resultado, `a`, `b`, `c` los operandos y `.type` es uno de los siguientes:

Tipo	Especificador .type
Grupos de bits sin tipo de 8, 16, 32 y 64 bits	.b8, .b16, .b32, .b64
Enteros sin signo de 8, 16, 32 y 64 bits	.u8, .u16, .u32, .u64
Enteros con signo de 8, 16, 32 y 64 bits	.s8, .s16, .s32, .s64
Punto flotante de 16, 32 y 64 bits	.f16, .f32, .f64

Instrucciones de hilo de la GPU PTX básica

Grupo	Instrucción	Ejemplo	Significado	Comentarios
Funciones aritméticas	arithmetic.*type*= .s32, u.32, .f32, s.64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a - b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	multiplicación-suma
	div.*type*	div.f32 d, a, b	d = a / b;	varias microinstrucciones
	rem.*type*	rem.u32 d, a, b	d = a % b;	resto entero
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 - a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	Punto flotante selecciona no-NAN
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	Punto flotante selecciona no-NAN
	stp.*cmp.type*	setp.lt.f32 p, a, b	p = (a < b);	compara y da valor al predicado
	numeric.*cmp*=eq, en, lt, le, gt, ge; unordered *cmp*=equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a	mover
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	Seleccionar con predicado
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	Conversión de tipo
Funciones especiales	Special.*type*= .f32 (algunas .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	recíproco
	sqrt.*type*	sqrt.f32 d, a	d = srqt(a);	raíz cuadrada
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	inverso raíz cuadrada
	sin.*type*	sin.f32 d, a	d = sin(a);	seno
	cos.*type*	cos.f32 d, a	d = cos(a);	coseno
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2);	logaritmo base 2
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	Exponencial binaria
Funciones lógicas	logic.*type*=.pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a	d = ~a;	complemento a 1
	cnot.*type*	cnot.b32 d, a	d = (a==0)? 1:0;	negación lógica complementaria
	shl.*type*	shl.b32 d, a, b	d = a << b;	Desplazamiento a la izquierda
	shr.*type*	shr.b32 d, a, b	d = a >> b;	Desplazamiento a la derecha
Accesos a memoria	memory.*space*=.global, .shared, .local, .const; *.type*=.b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space.type*	ld.global.b32 d, [a+off]	d = *(a+off);	carga desde memoria
	st.*space.type*	st.shared.b32 [d+off], a	*(d+off) = a;	almacenamiento en memoria
	tex.*nd.dtyp.btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	búsqueda de textura
	atom.*spc.op.type*	atom.global.add,u32 d, [a], b atom.global.cas,b32 d, [a], b, c	atomic { d = *a; *a = op(a, b); }	operación lectura-modificación-escritura atómica
	Atom.*op*= and, or, xor, add, min, max, exch, cas; *.spc*=.global; .type=.b32			
Control de flujo	branch	@p bra target	If (p) goto target;	salto condicional
	call	call (ret), func, (params)	ret =func (params)	llamada a función
	ret	ret	return;	retorno de función
	bar.sync	bar.sync d	esperar por hilos	barrera de sincronización
	exit	exit	exit;	termina la ejecución del hilo

FIGURA A.4.3 Instrucciones de hilo de la GPU PTX básica.

Los operandos son valores escalares de 32 o 64 bits almacenados en registros, un valor inmediato o una constante; los predicados son valores booleanos de 1 bit. El destino de los resultados es siempre un registro excepto en los almacenamientos en memoria. Las instrucciones pueden ser predicadas si se pone el prefijo `@p` o `@!p`, siendo `p` el registro de predicado. Las instrucciones de memoria o de texturas transfieren escalares o vectores de 2 a 4 componentes, hasta un máximo de 128 bits. Las instrucciones PTX especifican el comportamiento de un hilo.

Las instrucciones aritméticas operan con números punto flotante de 32 y 64 bits, enteros con signo y enteros sin signo. Algunas GPUs recientes pueden operar también con números punto flotante de 64 bits de precisión doble; véase la sección A.6. En las GPUs actuales, las instrucciones PTX lógicas o de enteros con operandos de 64 bits se traducen a dos o más microinstrucciones binarias con operandos 32 bits. Las instrucciones de funciones especiales están limitadas a punto flotante de 32 bits. Las instrucciones de control del flujo de datos del hilo son el salto condicional, la llamada y el retorno de una función, finalización del hilo y sincronización de barrera. El salto condicional `@p bra target` utiliza un registro de predicado, que ha tomado su valor en la ejecución previa de una instrucción `setp`, para decidir si el se toma o no el salto. Otras instrucciones pueden estar también predicadas con un registro predicado que toma el valor verdadero o falso.

Instrucciones de acceso a memoria

La instrucción `tex` busca y filtra muestras de texturas en conjuntos de texturas 1D, 2D y 3D almacenados en memoria a través de subsistema de texturas. En la búsqueda de texturas generalmente se utilizan coordenadas punto flotante interpoladas para el direccionamiento. Una vez que el hilo de sombreado de píxel calcula el color del fragmento de píxeles, el procesador de operaciones de raster lo mezcla con el color asignando a la posición (x, y) del píxel y lo escribe en la memoria.

Las instrucciones de carga/almacenamiento de memoria de la ISA PTX de Tesla se han implementado para incluir las necesidades de la computación y del lenguaje C/C++. Se utiliza un direccionamiento por byte y la dirección de memoria se obtiene a partir de un valor almacenado en un registro más un desplazamiento, con el objetivo de facilitar las optimizaciones de código de los compiladores tradicionales. Estas instrucciones son habituales en todos los procesadores, pero es un capacidad nueva e importante de las GPUs con arquitectura Tesla, ya que las GPUs anteriores sólo proporcionaban los accesos a texturas y píxeles que se necesitaban en las APIs gráficas.

En aplicaciones de computación, las instrucciones de carga/almacenamiento pueden especificar accesos a tres espacios de memoria de lectura/escritura diferentes, que forman los espacios de memoria de CUDA de la sección A.3:

- Una memoria local para datos temporales privados de cada hilo (implementada en una memoria DRAM externa).
- Una memoria compartida de baja latencia para accesos a datos compartidos entre hilos cooperativos del mismo CTA/bloque (implementada en una SDRAM integrada en el mimos chip que el multiprocesador).
- Una memoria global para grandes conjuntos de datos compartidos por todos los hilos de una aplicación de computación (implementada en una memoria DRAM externa).

Las instrucciones de acceso a los espacios de memoria global, compartido y local son `ld.global`, `st.global`, `ld.shared`, `st.shared`, `ld.local` y `st.local`. En estas aplicaciones se utiliza la instrucción `bar.sync` para la sincronización de los hilos de un CTA/bloque. Estos hilos se comunican con otros hilos a través de las memorias global y compartida.

Para mejorar el ancho de banda y reducir el sobrecoste, las instrucciones de carga/almacenamiento locales y globales de varios hilos paralelos de la misma trama SITM que hacen referencia al mismo bloque y cumplen unos criterios de alineamiento, se unen en una única solicitud de bloque de memoria. La fusión de solicitudes de memoria mejora significativamente las prestaciones en comparación con solicitudes separadas de varios hilos individuales. El elevado número de hilos del multiprocesador y el soporte para el procesamiento de una gran cantidad de solicitudes de carga pendientes, ayudan a ocultar la latencia de los accesos a las memorias local y global, que se implementan en una DRAM externa.

La arquitectura Tesla más reciente también proporciona un procesamiento eficiente de las operaciones de memoria atómicas mediante las instrucciones `atom.`*op*`.u32`, que incluyen las operaciones de enteros `add`, `min`, `max`, `and`, `or`, `xor`, `exchange` y `cas` (compare-and-swap), y facilitan la reducción paralela y la gestión de estructuras de datos paralelas.

Sincronización de barrera para comunicación entre hilos

La sincronización rápida de barrera permite que los programas CUDA se comuniquen frecuentemente a través de la memoria compartida y de la memoria global simplemente llamando a la función `__syncthreads()` como parte de la etapa de comunicación entre hilos. Esta función genera una sola instrucción `bar.sync`. Sin embargo, la sincronización entre un número elevado de hilos, hasta un máximo de 512 hilos por bloque de CUDA, es todo un reto.

El agrupamiento de los hilos en una trama SIMT de 32 hilos reduce la complejidad de la sincronización en un factor 32. Mientras los hilos esperan en un barrera del planificador de hilos SIMT no consumen ciclos del procesador. Cuando un hilo ejecuta la instrucción `bar.sync` se incrementa el contador de hilos esperando en la barrera y el planificador marca al hilo como "hilo esperando". Una vez que el contador de hilos en espera de la barrera comprueba que ya han llegado todos los hilos del CTA, el planificador libera los hilos que estaban en espera y reanuda su ejecución.

Procesador de *streaming* (SP)

El procesador de *streaming* (SP) con ejecución multihilo es el procesador principal del multiprocesador. Su banco de registros (RF) dispone de 1024 registros escalares de 32 bits que se reparten entre un máximo de 64 hilos. Ejecuta todas operaciones punto flotante fundamentales, entre las que se incluyen `add.f32`, `mult.f32`, `mad.f32` (multiplicación-suma punto flotante), `min.f32`, `max.f32` y `setp.f32` (comparación y asignación de un valor al predicado). Las instrucciones de multiplicación y suma punto flotante siguen el estándar IEEE 754 para operaciones punto flotante de precisión simple, incluyendo las representaciones de NaN (*Not-a-Number*) e infinito. El SP implementa también todas las operaciones de enteros de 32 y 64 bits y las instrucciones de comparación, conversión y lógicas del repertorio PTX de la figura A.4.3.

Las instrucciones `add` y `mul` implementan el modo de redondeo al más próximo par definido en el estándar IEEE 754. La instrucción de multiplicación-suma punto flotante `mad.f32` trunca el resultado de la multiplicación y redondea al más próximo par el resultado de la suma. Los operandos denormales son reemplazados por el valor cero con signo, para preservar el signo del operando. Cuando se produce un desbordamiento a cero *(underflow)* en un resultado, éste se hace igual a cero con signo.

Unidad de funciones especiales (SFU)

Algunas instrucciones tienen que ejecutarse en la SFU, concurrentemente con las instrucciones que se ejecutan en los SPs. La SFU implementa las instrucciones de funciones especiales de la figura A.4.3, que básicamente calculan aproximaciones punto flotante de 32 bits al recíproco, inverso de la raíz cuadrada y algunas funciones trascendentales clave. Además, en esta unidad se calcula la interpolación de atributos planares punto flotante de 32 bits de los programas de sombreado de píxel, proporcionando una interpolación precisa de atributos como el color, profundidad y coordenadas de textura.

Cada SFU segmentada genera un resultado punto flotante de 32 bits por ciclo; las dos SFUs del multiprocesador operan a un cuarto del ritmo de procesamiento de instrucciones de los ocho SPs. Por otra parte, la SFU puede ejecutar además la instrucción `mul.f32` concurrentemente con los ocho SPs y, de esta forma, aumenta el ritmo de computación pico en un 50% en hilos con una mezcla adecuada de instrucciones.

La evaluación de las funciones recíproco, inverso de la raíz cuadrada, $\log_2 x$, 2^x, seno y coseno se realiza con una interpolación cuadrática basada en un aproximación minimax optimizada, con una exactitud comprendida entre 22 y 24 bits de la mantisa. Véase la sección A.6 para un descripción más detallada de la SFU.

Comparación con otros multiprocesadores

Si lo comparamos con arquitecturas vectoriales SIMD, como por ejemplo el SSE de los procesadores x86, el multiprocesador SIMT puede ejecutar los hilos individuales de forma independiente, en lugar de ejecutarlos siempre de forma conjunta en grupos síncronos. El hardware SIMT encuentra el paralelismo de datos entre hilos independientes, mientras que el hardware SIMD necesita que el software exprese de forma explícita el paralelismo de datos de cada instrucción vectorial. Una arquitectura SIMT ejecuta una trama de 32 hilos de forma síncrona cuando los hilos siguen la misma vía de ejecución, pero si siguen vías divergentes, ejecuta cada hilo de forma independiente. La ventaja de este tipo de ejecución es significativa, porque las instrucciones y los programas SIMT describen simplemente el comportamiento de un solo hilo independiente, en vez de un vector de datos SIMD de cuatro o más canales de datos. Sin embargo, el multiprocesador SIMT tiene una eficiencia tipo SIMD, ya que distribuye el área y el coste de una unidad de emisión de instrucciones entre los 32 hilos de una trama y los ocho procesadores de *streaming*. Es decir, la arquitectura SIMT proporciona las prestaciones de una arquitectura SIMD junto con la productividad de un procesador con ejecución multihilo y evita la necesidad de codificar explícitamente las condiciones de borde y las divergencias parciales.

El sobrecoste impuesto por la arquitectura SIMT es pequeño, porque dispone de soporte hardware para la sincronización de barrera y la ejecución multihilo. Esto permite que los programas de sombreado gráfico y los hilos de CUDA exploten el paralelismo de grano muy fino. Los programas de gráficos y CUDA expresan el paralelismo de datos de

grano fino mediante hilos, en lugar de obligar al programador a expresarlo explícitamente con instrucciones vectoriales SIMD. La principal ventaja para el programador es que es más simple y más productivo desarrollar código escalar para un solo hilo que código vectorial y el multiprocesador SIMT ejecuta el código con eficiencia SIMD.

El acoplamiento de ocho procesadores de *streaming* en un multiprocesador y la implementación de un sistema con número variable de tales multiprocesadores, da como resultado un multiprocesador con dos niveles de multiprocesadores. El modelo de programación CUDA explota esta jerarquía de niveles proporcionando hilos para computación paralela de grano fino y mallas de bloques de hilos para computación paralela de grano grueso. Por lo tanto, el programa de hilo proporciona tanto paralelismo de grano fino como paralelismo de grano grueso. Por el contrario, las CPUs con instrucciones vectoriales SIMD tienen que utilizar dos modelos de programación diferentes para explotar el paralelismo de grano fino y el paralelismo de grano grueso: hilos paralelos de grano grueso en núcleos diferentes e instrucciones vectoriales SIMD para el paralelismo de datos de grano fino.

Conclusiones sobre los multiprocesadores con ejecución multihilo

La GPU que hemos tomado como ejemplo, basada en la arquitectura Tesla, implementa la ejecución multihilo a gran escala, explotando el paralelismo de grano fino de los programas de sombreado de píxel y de los hilos de CUDA con un elevado número de hilos ligeros, hasta un máximo de 512, que se ejecutan concurrentemente. Utiliza una variación de la arquitectura SIMD y la ejecución multihilo, llamada SIMT (instrucción única, múltiples hilos o *single-instruction, multiple-thread*) para emitir de forma eficiente una instrucción a una trama de 32 hilos paralelos permitiendo, al mismo tiempo, que cada hilo tome una vía de ejecución diferente en los saltos condicionales. Cada hilo ejecuta sus instrucciones en uno de los ocho procesadores de *streaming* (SP), que soportan ejecución multihilo con un máximo de 64 hilos.

La ISA PTX es una ISA escalar de carga/almacenamiento basada en registros que describe la ejecución de un hilo. Las instrucciones PTX se optimizan y se traducen a las microinstrucciones binarias de una GPU específica; esto permite que las instrucciones puedan evolucionar de forma rápida sin afectar a los compiladores y herramientas software que generan instrucciones PTX.

A.5 Sistema de memoria paralelo

Aunque es externo a la GPU propiamente dicha, el subsistema de memoria es el componente más importante a la hora de determinar las prestaciones del sistema gráfico. Las cargas de trabajo gráficas son muy exigentes respecto a los ritmos de transferencia de datos a la memoria y desde ella. La mayor parte del tráfico con memoria se debe a las operaciones de escritura y mezclado de píxeles (lectura-modificación-escritura), lectura y escritura en el búfer de profundidad, lectura de mapas de texturas y lectura de órdenes y de los atributos y vértices.

Tal como se ha mostrado en la figura A.2.5, las GPUs actuales son altamente paralelas. Por ejemplo, la GeForce 8800 puede procesar 32 píxeles por ciclo a una frecuencia de 600 MHz. Para procesar cada píxel es necesario, típicamente, realizar la lectura y escritura del color y una lectura y escritura de la profundidad de un píxel de 4 bytes. Normalmente, se necesita leer una media de dos o tres téxeles de cuatro bytes para obtener el color del píxel. Así, en un caso típico, se necesitarán 28 bytes por 32 píxeles = 896 bytes por ciclo. Claramente, el ancho de banda necesario en el sistema de memoria es enorme.

Para atender estos requerimientos, el sistema de memoria de la GPU tienen las siguiente características:

- Es ancho, en el sentido de que hay un gran número de pines para transmitir datos entre la GPU y la memoria, y está compuesto de varios chips DRAM para completar el ancho del bus de datos.
- Es rápido, en el sentido de que se utilizan técnicas de señalización agresivas para maximizar el ritmo de datos (bits/segundo) por pin.
- Las GPUs intentan aprovechar cada ciclo disponible para transferir datos a la memoria o desde ella. Para ello, las GPUs no intentan minimizar la latencia de los accesos a la memoria, porque la alta productividad (eficiencia de la utilización) suele estar reñida con la baja latencia.
- Se utilizan técnicas de compresión con pérdidas, de lo cual tiene que ser consciente el programador, y sin pérdidas, que son invisibles a la aplicación y oportunistas.
- Se utilizan estructuras de fusión de cache y trabajo para reducir el tráfico hacia fuera del chip y para asegurar que los ciclos dedicados a la transferencia de datos se utilizan tanto como sea posible.

Consideraciones sobre la DRAM

Las GPUs deben tener en cuenta las características singulares de las DRAM. Internamente, las DRAM está organizadas en varios bancos (habitualmente entre cuatro y ocho), donde el número de filas del banco es una potencia de dos (típicamente 16 384) y cada fila tiene un número de bits que también es una potencia de dos (típicamente 8192). Imponen ciertos requerimientos temporales a los procesadores. Por ejemplo, se necesitan docenas de ciclos para activar una fila, pero una vez activada, se puede acceder a los bits de la fila de forma aleatoria direccionando una columna cada cuatro ciclos. Las DRAM síncronas de ritmo de transferencia de datos doble (*Double-Data Rate*, DDR) transfieren datos en el flanco de subida y en el flanco de bajada del reloj (véase capítulo 5). De este modo, el ritmo de transferencia de datos de una DRAM DDR con un reloj de 1 GHz es 2 gigabits por segundo por pin. Como las DRAM DDR gráficas tienen habitualmente 32 pines de datos bidireccionales, se pueden leer o escribir ocho bytes por ciclo.

En las GPUs hay varios generadores de tráfico con memoria. Hay diferentes etapas del pipeline gráfico que tienen sus propios flujos de solicitudes: búsqueda de órdenes y atributos de vértices, búsqueda y carga/almacenamiento de texturas de sombreado y lectura-escritura de la profundidad y el color del píxel. En cada etapa lógica, a menudo hay varias unidades independientes que buscan productividad paralela y cada una de éstas es

un solicitante independiente para la memoria. Desde el punto de vista de la memoria, hay una cantidad enorme de solicitudes no correlacionadas en ejecución, y esto no encaja bien con patrón de referencias preferido por la DRAM. Una solución es que el controlador de la memoria tenga pilas de tráfico separadas para cada banco y espere a que haya bastante solicitudes pendientes para una fila de la DRAM antes de activar esta fila y transferir todo el tráfico de golpe. Observe que la acumulación de solicitudes pendientes, aunque es buena para la localidad de las filas de la DRAM y significa un uso eficiente del bus de datos, lleva a latencias medias más altas para el solicitante, que ve que sus solicitudes están paradas esperando por otras solicitudes. El diseño debe asegurar que ninguna solicitud tiene que esperar demasiado tiempo, en caso contrario algunas unidades de procesamiento podrían morir de inanición esperando por los datos y, en última instancia, causar que procesadores vecinos se queden inactivos.

El subsistema de memoria de la GPU se organiza en varias *particiones de memoria*, cada una de las cuales consta de un controlador de memoria independiente y uno o dos dispositivos DRAM que pertenecen completamente y de forma exclusiva a esta partición. Para alcanzar el mejor equilibrio de la carga y aprovechar, por lo tanto, las prestaciones teóricas de n particiones, las direcciones deben estar entrelazadas de forma fina y uniforme entre todas las particiones de la memoria. El desplazamiento del entrelazado es típicamente un bloque de unos pocos cientos de bytes. El número de particiones se diseña para equilibrar el número de procesadores y los otros solicitantes de memoria.

Caches

Las cargas de trabajo de la GPU suelen tener grandes conjuntos de trabajo —del orden de cientos de megabytes para generar un único cuadro gráfico. Al contrario que en la CPUs, no es práctico tener integradas en el mismo chip del procesador caches lo suficientemente grandes como para mantener la mayor parte posible del conjunto de trabajo de una aplicación gráfica. Mientras la CPUs tienen índices de acierto muy elevados (99% o mayor), el índice de acierto de la GPU está alrededor del 90% y por lo tanto debe sobrellevar este elevado número de fallos. Si la CPU puede diseñarse de forma que se pare mientras espera por un poco habitual fallo de cache, la GPU necesita continuar, combinando fallos y aciertos. A este tipo de cache le llamamos *arquitectura de cache de streaming*.

La caches de la GPU deben tener anchos de banda muy elevados. Consideremos el caso de una cache de texturas. Una unidad de texturas típicamente puede evaluar dos interpolaciones bilineales para cada cuatro píxeles en cada ciclo de reloj, y una GPU puede tener varias unidades de texturas operando independientemente. Cada interpolación bilineal necesita cuatro téxeles y cada téxel puede ser un valor de 64 bits. Son típicos cuatro componentes de 16 bits. Por lo tanto, el ancho de banda es $2 \times 4 \times 4 \times 64 = 2048$ bits por ciclo. Cada téxel de 64 bits se direcciona de forma separada, por lo tanto, la cache necesita gestionar 32 direcciones por ciclo. Esto favorece de forma natural la organización de los conjuntos de SRAM en varios bancos y/o con varios puertos.

MMU

Las GPU actuales son capaces de hacer la traducción de dirección virtual a dirección física. En la GeForce 8800 todas las unidades generan direcciones de 40 bits en el espacio de direcciones virtuales. En computación, las instrucciones de carga y almacenamiento utilizan direcciones de byte de 32 bits que, al añadirle un desplazamiento, se extienden a

direcciones virtuales de 40 bits. Una unidad de gestión de memoria *(Memory management Unit, MMU)* realiza la traducción de la dirección de memoria virtual a dirección de memoria física; el hardware lee las tablas de páginas de memoria local en respuesta a fallos en la jerarquía de TLBs (*translation lookaside buffer*) distribuidas entre los procesadores y los motores de renderizado. Las entradas de la tabla de páginas de la GPU especifican además el algoritmo de compresión para cada página. Los tamaños de página varían entre 4 y 128 kilobytes.

Espacios de memoria

Tal como se ha introducido en la sección A.3, en CUDA hay diferentes espacios de memoria que permiten al programador almacenar los datos en la forma más adecuada para optimizar las prestaciones. En la siguiente discusión, ponemos como ejemplo la GPU con arquitectura Tesla de NIVIDIA.

Memoria global

La memoria global se implementa en una DRAM externa; no es local a ningún multiprocesador de *streaming* (SM) porque se utiliza para la comunicación entre diferentes CTAs (bloques de hilos) en mallas diferentes. De hecho, los CTAs que referencian una posición de la memoria global no tienen por qué estar ejecutándose en la GPU al mismo tiempo; en CUDA el programador no conoce el orden relativo de ejecución de los CTAs. Como el espacio de direcciones está distribuido uniformemente entre todas las particiones de memoria, debe haber una vía de lectura/escritura desde un multiprocesador de *streaming* a cualquier partición de la DRAM.

No está garantizada la coherencia secuencial en el acceso a la memoria global por parte de hilos diferentes (y procesadores diferentes): los programas de hilos ven un modelo de ordenación de accesos a memoria relajado. Es decir, dentro del hilo, se mantiene el orden de las lecturas y escrituras a una misma posición de memoria, pero el orden de los accesos a diferentes posiciones puede no mantenerse. Por otra parte, las solicitudes de lecturas y escrituras de memoria provenientes de hilos diferentes no están ordenadas; sin embargo, en un CTA se puede utilizar la instrucción de sincronización de barrera `bar.sync` para obtener una ordenación estricta de los accesos a memoria por parte de los hilos del CTA. La instrucción `membar` proporciona una barrera de memoria que termina los anteriores accesos a memoria y los hace visibles a los demás hilos antes de seguir. Finalmente, los hilos también pueden utilizar las operaciones atómicas de memoria, descritas en la sección A.4, para coordinarse en la memoria compartida.

Memoria compartida

La memoria compartida de cada CTA es visible sólo a los hilos del CTA en cuestión y sólo existe desde que se crea el CTA hasta que termina. Por lo tanto, la memoria compartida puede residir en el chip del multiprocesador. Este enfoque tiene muchos beneficios. En primer lugar, el tráfico de la memoria compartida no necesita competir con el tráfico de la memoria global por el ancho de banda limitado de la comunicación con la memoria externa. Segundo, es ventajoso disponer de memorias de alto ancho de banda integradas en el mismo chip que soporten las demandas de lectura/escritura de los multiprocesadores de *streaming*. De hecho, la memoria compartida está altamente acoplada al multiprocesador de *streaming*.

Cada multiprocesador de *streaming* consta de ocho procesadores de hilo. Durante un ciclo de reloj de memoria compartida cada procesador de hilo puede procesar el equivalente a dos instrucciones, así en cada ciclo se pueden procesar el equivalente a 16 solicitudes de memoria compartida. Como cada hilo genera sus propias direcciones y éstas típicamente son únicas, la memoria compartida consta de 16 bancos SRAM direccionables independientemente. Normalmente, los 16 bancos son suficientes para mantener la productividad con los patrones de acceso habituales, pero siempre es posible que haya casos patológicos; por ejemplo, puede ocurrir que los 16 hilos quieran acceder a direcciones del mismo banco SRAM. El circuito de interconexión debe hacer posible dirigir la solicitud de cualquier hilo a cualquier banco de memoria, por lo tanto se necesita un circuito de interconexión 16-por-16.

Memoria local

La memoria local de cada hilo es privada y visible sólo por este hilo. Es mayor que el banco de registros del hilo y los programas pueden operar con sus direcciones. Como se puede necesitar mucho espacio de memoria local (recordemos que el espacio total es el espacio por hilo multiplicado por el número de hilos activos), la memoria local se implementa en una DRAM externa.

Aunque la memoria global y local residen en memorias externas, son aptas para tener utilizarse en conjunción con una cache en el chip.

Memoria de constantes

Es una memoria de sólo lectura para un programa que se ejecuta en el SM (se puede modificar mediante órdenes a la GPU). Reside en una DRAM externa y tiene una cache en el SM. Como habitualmente la mayoría, por no decir todos, de los hilos de una trama SIMT leen la misma dirección de la memoria de constantes, es suficiente con hacer la búsqueda de una sola dirección por ciclo. La cache de constantes se diseña para emitir valores escalares a los hilos de la trama.

Memoria de texturas

La memoria de texturas almacena grandes conjuntos de datos de solo lectura. Las texturas para computación tienen los mismos atributos y capacidades que las utilizadas en gráficos 3D. Aunque habitualmente las texturas son imágenes bidimensionales (conjuntos 2D de valores de píxel) también se utilizan texturas 1D (lineales) y 3D (volumétricas).

Un programa de cómputo utiliza la instrucción `tex` para referencias una textura. Los operandos incluyen un identificador de la textura y 1, 2 o 3 coordenadas en función de la dimensión de la textura. Las coordenadas punto flotante tienen un parte fraccional que especifica una posición entre posiciones de téxeles y utilizan una interpolación bilineal ponderada de los cuatro valores más próximos (para una textura 2D) para obtener el resultado que se devuelve al programa.

Las recuperaciones de texturas se almacenan en una jerarquía de caches diseñada para optimizar la productividad de la recuperación de texturas desde miles de hilos concurrentes. Algunos programas utilizan las recuperaciones de texturas para disponer de una cache de la memoria global.

Superficies

Superficie es un término genérico para designar conjuntos unidimensionales, bidimensionales o tridimensionales de pixeles con un formato asociado. Se han definido varios formatos; por ejemplo, un píxel puede definirse como cuatro componentes RGBA enteros de 8 bits, o como cuatro componentes punto flotante de 16 bits. El programa no necesita conocer el tipo de superficie, con la instrucción `tex` puede transformar los resultados a punto flotante, dependiendo del formato de la superficie.

Carga/almacenamiento

Las instrucciones de carga/almacenamiento con direccionamiento a nivel de bytes facilitan la escritura y compilación de programas en lenguajes convencionales tipo C y C++. Los programas CUDA utilizan estas instrucciones para el acceso a la memoria.

Para mejorar el ancho de banda y reducir el sobrecoste, las instrucciones de carga/almacenamiento locales y globales unen las solicitudes de hilos paralelos de la misma trama, que hacen referencia al mismo bloque y cumplen unos criterios de alineamiento en una única solicitud de bloque de memoria. Esta fusión de solicitudes de memoria individuales en bloques mejora significativamente las prestaciones en comparación con las solicitudes separadas. El elevado número de hilos del multiprocesador y el soporte para el procesamiento de una gran cantidad de solicitudes de carga pendientes, ayudan a ocultar la latencia de los accesos a las memorias local y global, que se implementan en una DRAM externa.

ROP

Como se muestra en la figura A.2.5, la GPU Tesla de NIVIDIA tiene un conjunto de procesadores de *streaming* (SP), que realizan todos los cálculos programables de la GPU y un sistema de memoria escalable formado por el control de la DRAM externa y el procesador de función fija Procesador de Operaciones de Raster (ROP), que realiza las operaciones color y profundidad del búfer de cuadros directamente en memoria. Cada ROP está asociado con una partición de memoria específica y recibe datos del SM a través de una red de interconexión. Es responsable de los tests y actualizaciones de profundidad y cliché y de la mezcla de colores. El ROP y el controlador de memoria cooperan para implementar compresiones sin pérdidas de color y profundidad (hasta compresiones 8:1) para reducir las necesidades de ancho de banda con memoria externa. Realiza también operaciones atómicas de memoria.

A.6 Aritmética punto flotante

Las GPUs actuales realizan la mayor parte de las operaciones aritméticas en el núcleo de procesamiento programable en una representación de precisión simple de 32 bits compatible con el estándar IEEE 754 (véase el capítulo 3). La aritmética punto flotante de 16, 24 y 32 bits reemplazó a la aritmética punto fijo de las primeras GPUs, y posteriormente se introdujo la aritmética punto flotante de 32 bits compatible con el estándar IEEE 754.

Aún así, parte de la lógica de función fija de la GPU, como por ejemplo el hardware para filtrado de texturas, continúa utilizando formatos numéricos no estándares. Algunas GPUs recientes tienen también instrucciones de punto flotante de precisión doble de 64 bits compatible con el estándar IEEE 754.

Formatos permitidos

El estándar IEEE 754 para aritmética punto flotante [2008] define formatos básicos y formatos de almacenamiento. La GPU utiliza dos de los formatos básicos para cálculo, los formatos de precisión simple y de precisión doble, con 32 y 64 bits respectivamente. El estándar incluye además un formato de 16 bits para almacenamiento, el **formato de precisión mitad**. Las GPUs y el lenguaje Cg de sombreado utilizan este formato estrecho para tener un almacenamiento y movimiento de datos eficiente, manteniendo al mismo tiempo un alto rango dinámico. La GPU realiza muchas operaciones de filtrado de texturas y mezclado de píxeles en las unidades de filtrado de texturas y de operaciones de raster con precisión mitad. Este formato mitad es exactamente igual al formato OpenEXR de Industrial Light and Magic [2003] para ficheros de imágenes con elevado rango dinámico, utilizado para los componentes del color en aplicaciones de movimiento y procesado de de imágenes.

Formato de precisión mitad: formato punto flotante binario de 16 bits con 1 bit de signo, 5 bits de exponente, 10 bits fraccionales y un bit entero implícito.

Aritmética básica

Las operaciones punto flotante habituales de los núcleos programables de la GPU incluyen la suma, multiplicación, **multiplicación-suma**, cálculo del mínimo, del máximo, comparación, asignación de valor a un predicado y conversiones de formatos en números enteros y en punto flotante. Además, habitualmente las instrucciones de punto flotante incluyen modificadores de operandos para complementación y valor absoluto.

Las operaciones de suma y multiplicación punto flotante de la mayoría de la GPUs actuales son compatibles con el estándar IEEE 754 de precisión simple, incluyendo los valores *not-a-number* (NaN) e infinito, e implementan por defecto el redondeo al más próximo par. Para aumentar la productividad de las instrucciones punto flotante, las GPUs utilizan a menudo la instrucción compuesta multiplicación-suma (`mad`). Esta instrucción hace una multiplicación con truncamiento seguida de una suma con redondeo al más próximo par. Por lo tanto, emite dos operaciones punto flotante en un ciclo, sin que el planificador tenga que emitir dos instrucciones separadas, pero el cálculo no está mezclado (*fused*), porque el resultado de la multiplicación se trunca antes de hacer la suma. Esta es la diferencia con la multiplicación-suma mezclada (*fused multiply-add*) que se discutió en el capítulo 3 y que se discutirá más adelante en esta sección. Por otra parte, la GPU reemplaza los valores denormales y los resultados con desbordamiento a cero por cero con signo, para mantener el signo del operando o el resultado.

Multiplicación-suma (MAD): instrucción punto flotante que realiza una operación compuesta: multiplicación seguida de suma.

Aritmética especializada

Las GPUs disponen de hardware para la aceleración del cálculo de funciones especiales, interpolación de atributos y filtrado de texturas. Entre las instrucciones de funciones especiales podemos encontrar el coseno, seno, exponencial binaria, logaritmo binario, recíproco e inverso de la raíz cuadrada. Por otra parte, las ins-

trucciones de interpolación de atributos proporcionan una generación eficiente de los atributos de los píxeles, a partir de la evaluación de funciones del plano. La **unidad de funciones especiales (SFU)**, que se ha introducido en la sección A.4, realiza el cálculo de las funciones especiales de la interpolación de atributos planares [Oberman y Siu, 2005].

Unidad de funciones especiales (SFU): unidad hardware que calcula funciones especiales e interpola atributos planares.

Hay varios métodos para la evaluación de las funciones especiales en hardware. Un método que se ha mostrado muy eficiente para la evaluación de estas funciones en hardware, incluyendo recíproco, inverso de la raíz cuadrada, $\log_2 x$, 2^x, seno y coseno, es la interpolación cuadrática basada en aproximaciones minimax mejoradas.

A continuación resumimos este método. El significando de los n bits del operando binario X se divide en dos partes: una parte más significativa de m bits, X_u, y una parte menos significativa de n-m bits, X_l. Con los m bits más significativos X_u se accede a un conjunto de tres tablas de las que se obtienen tres coeficientes C_0, C_1 y C_2. Este conjunto de tablas es específico para cada función. Con estos coeficientes se aproxima la función dada f(X) en el rango $X_u <= X < X_u + 2^{-m}$ mediante la expresión:

$$f(X) = C_0 + C_1 X_l + C_2 X_l^2$$

La exactitud de la estimación de las funciones está entre 22 y 24 bits del significando. Las estadísticas de algunas fucniones evaluadas con este método se muestran en la figura A.6.1.

El estándar IEEE 754 especifica los requisitos para el redondeo exacto de la división y la raíz cuadrada; sin embargo, para muchas aplicaciones de las GPUs, el redondeo exacto no es necesario. Para estas aplicaciones es más importante la productividad computacional que la exactitud del último bit. Para la funciones especiales de la SFU, la biblioteca matemática de CUDA proporciona una implementación exacta o rápida de las instrucciones SFU.

Otra operación aritmética especializada de la GPU es la interpolación de atributos. Los *atributos* clave se especifican habitualmente para los vértices de las primitivas que componen la escena que se va a renderizar. Algunos ejemplos de atributos

Función	Intervalo de entrada	Exactitud (bits correctos)	Error en ULP*	% redondeadas exactamente	Monotónica
$1/x$	[1.2]	24.02	0.98	87	sí
$1/\text{sqrt}(x)$	[1.4]	23.40	1.52	78	sí
2^x	[0.1]	22.51	1.41	74	sí
$\log_2 x$	[1.2]	22.57	N/A**	N/A	sí
sin/cos	$[0, \pi/2)$	22.47	N/A	N/A	no

*ULP: unidad en la última posición. **N/A: no aplicable.

FIGURA A.6.1 Estadísticas de aproximaciones de funciones especiales. Para la unidad de funciones especiales (SFU) de la NVIDIA GeForce 8800.

son el color, la profundidad y las coordenadas de textura. Para obtener el valor de los atributos en cada posición de píxel, es necesario interpolar estos atributos en el espacio de visualización (x, y). El valor de un atributo *U* en el plano (x, y) se expresa con la siguiente ecuación planar:

$$U(x,y) = A_u x + B_u y + C_u$$

donde A, B y C son los parámetros de interpolación asociados al atributo U, representados como números punto flotante de precisión simple.

Una vez establecida la necesidad de disponer de un evaluador de funciones y un interpolador de atributos en el procesador de sombreado de píxeles, es posible diseñar un SFU que implemente ambas funciones de forma eficiente. Ambas funciones utilizan una suma de productos para la interpolación de resultados con un número de términos muy similar en ambos casos.

Operaciones de texturas

La proyección y filtrado de texturas es otro conjunto de operaciones punto flotante especializadas de la GPU. La proyección de texturas incluye:

1. Recibir la dirección de la textura (s, t) para el píxel de la pantalla que se está procesando, donde s y t son números punto flotante de precisión simple.
2. Calcular el nivel de detalle para identificar el nivel **MIP-map** correcto de la textura.
3. Calcular la interpolación trilineal.
4. Escalar la dirección (s, t) de la textura al nivel MIP-map seleccionado.
5. Acceder a memoria para obtener los téxeles (*texture element*) necesarios.
6. Realizar la operación de filtrado en los téxeles.

MIP-map: frase en latín, *multum in parvo*, que significa mucho en un espacio pequeño. Un MIP-map consta de imágenes precalculadas a diferentes resoluciones que se utilizan para acelerar el renderizado y reducir imperfecciones.

La proyección de texturas requiere una cantidad significativa de operaciones punto flotante, la mayoría de las cuales se pueden realizar con precisión mitad de 16 bits. Por ejemplo, la GeForce Ultra proporciona, adicionalmente a las instrucciones punto flotante de precisión simple convencionales del estándar IEEE 754, operaciones punto flotante de formato propietario para las instrucciones de proyección de texturas con unas prestaciones pico de 500 GFLOPS. Detalles adicionales sobre la proyección y filtrado de texturas se pueden encontrar en Foley y Dam [1995].

Prestaciones

El hardware de suma y multiplicación punto flotante está totalmente segmentado y la latencia se ha optimizado para equilibrar el área y el retraso. Por otra parte y aunque también está segmentada, la productividad de la unidad de funciones especiales es menor que la de la operaciones de suma y multiplicación. En las GPUs modernas, con una SFU compartida por cuatro núcleos SP, su productividad es un cuarto de la productividad de los SP. En contraste, las CPUs tienen una productividad sensiblemente menor en funciones similares, como división y raíz cuadrada, aunque la exactitud de los resultados es mayor. El hardware de interpolación de atributos está también completamente segmentado para disponer de sombreadores de píxeles rápidos.

Precisión doble

Las GPUs más modernas, como la Tesla T10P, tienen también soporte hardware para las operaciones punto flotate de doble precisión de 64 bits, suma, multiplicación y conversión entre formatos punto flotante y de enteros. El estándar IEEE 754 de 2008 incluye las especificaciones para la operación FMA (*fused multiply-add*), que se ha discutido en el capítulo 3. Esta operación es una multiplicación seguida de una suma con un único redondeo, manteniendo una exactitud completa en las operaciones intermedias. Esto permite un cálculo más exacto de las computaciones punto flotante con acumulación de productos, por ejemplo, el producto escalar de dos vectores, la multiplicación de matrices y la evaluación de polinomios. La instrucción FMA permite además tener implementaciones software eficientes para la división y la raíz cuadrada con redondeo exacto, sin necesidad de unidades hardware específicas.

La unidad FMA de precisión doble implementa las operaciones de suma y multiplicación, la conversión de formatos y la operación FMA propiamente dicha. Además, incluye soporte para número denormalizados tanto en las entradas como en las salidas. Su diagrama de bloques se muestra en la figura A.6.2.

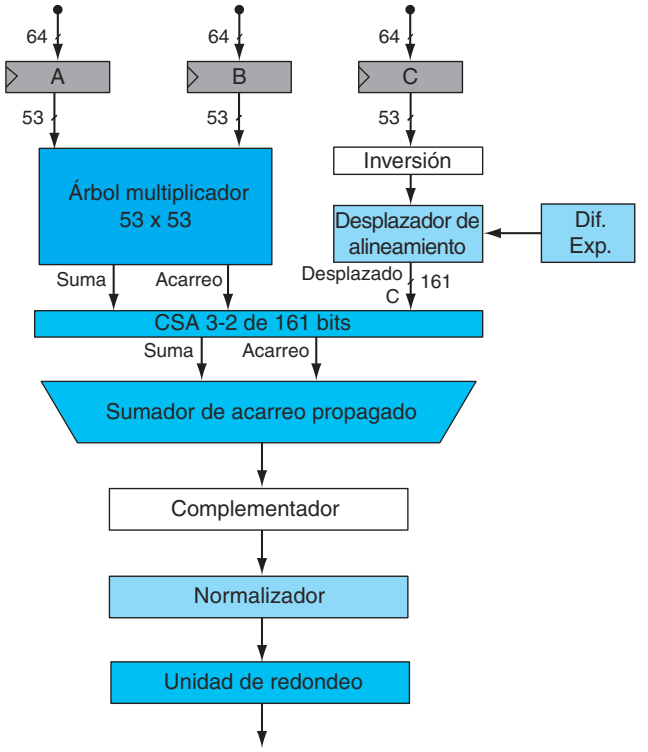


FIGURA A.6.2 Unidad de multiplicación-suma mezclada (FMA) de precisión doble. Hardware para implementar la operación A x B + C en punto flotante de precisión doble.

Como se muestra en la figura, se multiplican los significandos de A y B y se obtiene un producto de 106 bits en formato de acarreo almacenado (*carry save*). En paralelo, el sumando C de 53 bits se invierte y se alinea condicionalmente respecto al producto. A continuación se suman las palabras de acarreo y suma del producto con el sumando alineado en un sumador de acarreo almacenado (*carry save adder*, CSA) de 161 bits. Las palabras de acarreo y suma del resultado del CSA se suman en un sumador de acarreo propagado para obtener un resultado no redundante en complemento a 2, pero todavía no redondeado. Este resultado se recomplementa condicionalmente para convertirlo a la representación signo magnitud. Finalmente, se normaliza y redondea al formato deseado.

A.7 Casos reales: NVIDIA GeForce 8800

La GPU NVIDIA GeForce 8800, introducida en noviembre de 2007, es un procesador unificado de vértices y píxeles con soporte para aplicaciones de computación escritas en C utilizando el modelo de programación paralela de CUDA. Es la primera implementación de la arquitectura Tesla, unificada para gráficos y computación, que se ha descrito en la sección A.4 y en Lindholm, Nickolls, Oberman y Montrym [2008]. La familia de GPUs con arquitectura Tesla ofrece soluciones a las diferentes necesidades de los computadores portátiles y de sobremesa, estaciones de trabajo y servidores.

Conjunto de procesadores de *streaming* (SPA)

La GPU GeForce 8800, que se muestra en la figura A.7.1, consta de 128 procesadores de *streaming* (SP) organizados en 16 multiprocesadores de *streaming* (SM). Cada dos SM comparten una unidad de texturas y conforman un clúster textura/procesador (*texture/processor cluster,* TPC). Finalmente, un conjunto de 8 TPCs forman el Conjunto de Procesadores de *Streaming* (SPA, *Streaming Processor Array*), que se encarga de la ejecución de todos los programas de sombreado gráfico y de computación.

La interfaz con el anfitrión comunica la GPU con la CPU anfitriona a través de un bus PCI-Express, comprueba la coherencia de los comandos y hace los cambios de contexto. La unidad de ensamblaje de entrada recoge las primitivas geométricas (puntos, líneas y triángulos). Los bloques de distribución de trabajos envían los conjuntos de vértices, píxeles e hilos de computación a los TPCs del SPA. Los TPCs ejecutan los programas de sombreado vértices y geometría y los programas de computación. Los datos geométricos de salida se envían al bloque Ventana de Visualización/Recorte/Configuración/Raster/ZCull para la rasterización en fragmentos de píxeles que después se redistribuyen a los SPA para la ejecución de los programas de sombreado de píxeles. Los píxeles sombreados se envían, a través de la red de interconexión, a las unidades ROP. La red también se encarga de dirigir las solicitudes de lectura de la memoria de texturas desde el SPA a las DRAM y lee datos de la DRAM que se almacenan en la cache de nivel 2 del SPA.

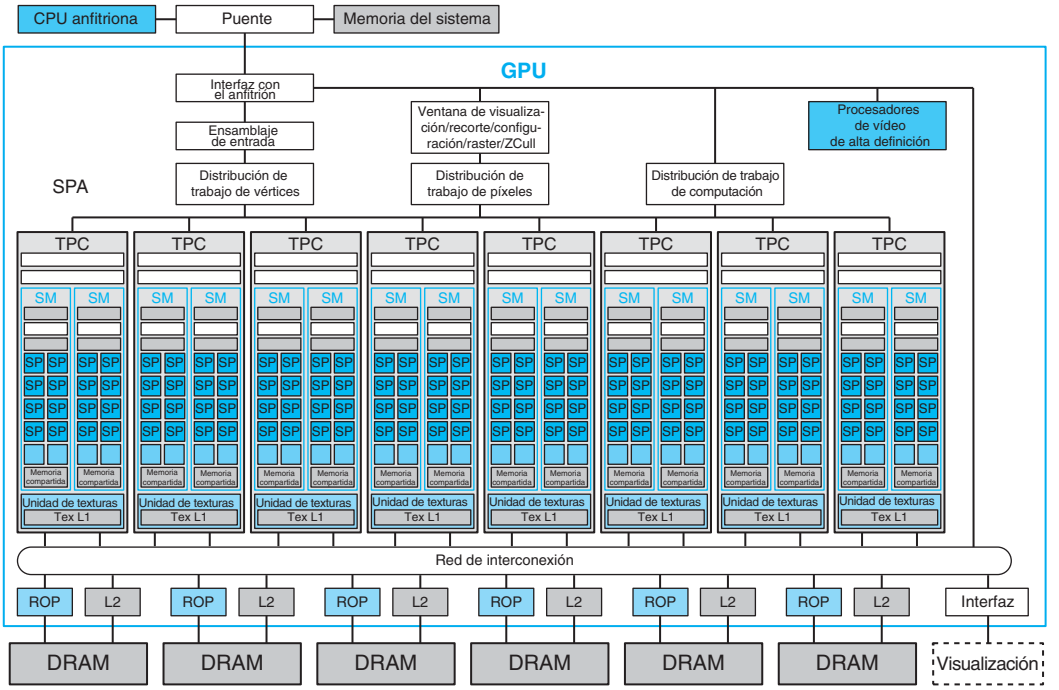


FIGURA A.7.1 Arquitectura de la GPU unificada de gráficos y computación Tesla de NVIDIA. Esta GeForce 8800 tiene 128 procesadores de *streaming* (SP) en 16 multiprocesadores de *streaming* (SM), organizados en ocho clústeres de textura/procesador (TPC). Los procesadores están conectados con seis particiones DRAM de 64 bits a través de una red de interconexión. El número de núcleos SP, SM, particiones DRAM y otras unidades pueden varias entre diferentes implementaciones de la arquitectura Tesla en diferentes GPUs.

Clúster de textura/procesador (TPC)

Tal como se muestra en la figura A.7.2, cada TPC consta de un controlador de geometría, un controlador SM (SMC), dos multiprocesadores de *streaming* (SM) y una unidad de texturas.

El controlador de geometría proyecta el pipeline lógico de vértices gráficos en los SM físicos, dirigiendo los flujos de los atributos de los vértices y primitivas y de topología en el TPC.

El SMC controla varios SMs, actuando de árbitro en los accesos a la unidad compartida de texturas, la vía de carga/almacenamiento y la vía de E/S. Atiende a tres cargas de trabajo simultáneamente: vértices, geometría y píxeles.

La unidad de texturas procesa una instrucción de textura para un quad de vértices, geometrías o píxeles o quad de hilos de computación por ciclo. Los operandos de la instrucción de textura son las coordenadas de textura y típicamente proporciona como salidas los cuatro componentes punto flotante del color (RGBA) ponderados. Esta unidas está profundamente segmentada. Aunque tiene una cache para aprovechar la localidad del filtrado, procesa los aciertos mezclados con los fallos sin paradas.

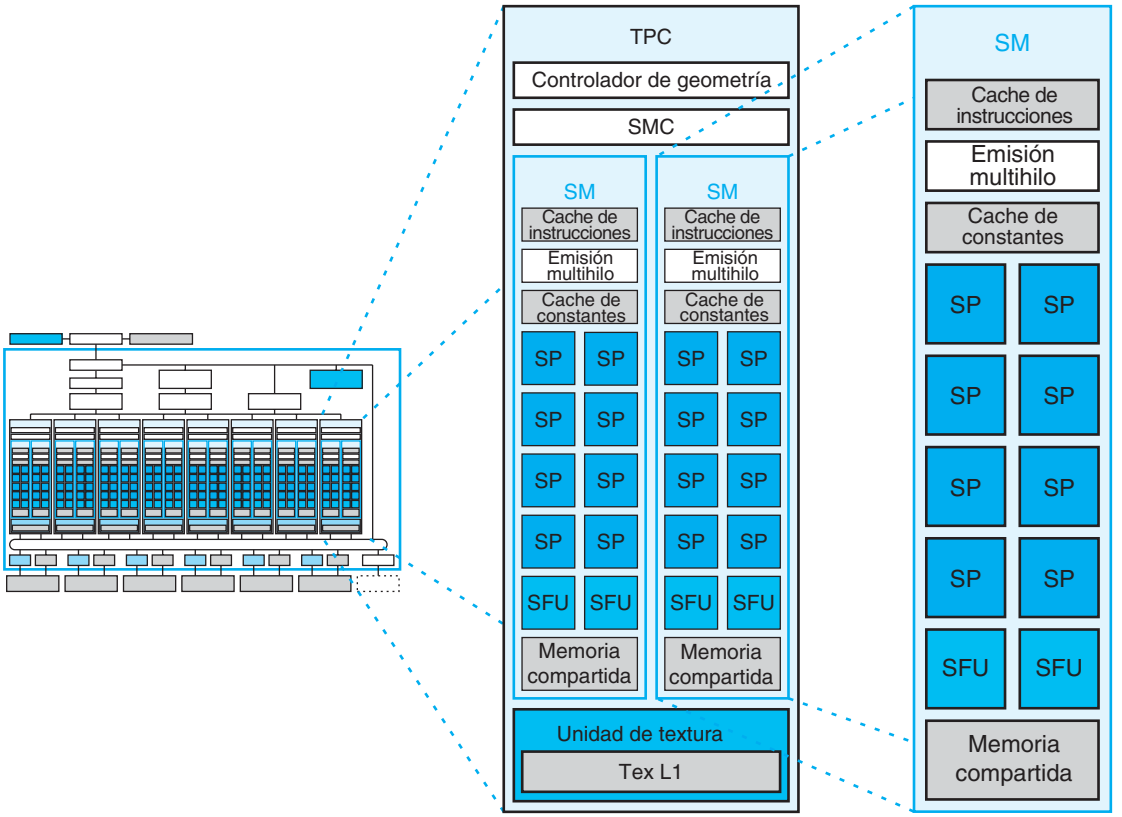


FIGURA A.7.2 Clúster de textura/procesador (TPC) y un multiprocesador de *streaming* (SM). Cada SM tiene ocho procesadores de *streaming* (SP), dos SFUs y una memoria compartida.

Multiprocesadores de *streaming* (SM)

El SM es un multiprocesador unificado de gráficos y computación que ejecuta programas de sombreado de vértices, geometría y fragmentos de píxeles y programas paralelos de computación. Consta de ocho procesadores SP de hilo, dos SFUs, una unidad de búsqueda y emisión multihilo de instrucciones (MT Issue), una cache de instrucciones, una cache de constantes de sólo lectura y una memoria compartida de lectura/escritura de 16 KB. Ejecuta instrucciones escalares para hilos individuales.

La frecuencia del reloj de los núcleos SP y de las SFUs en la GeForce 8800 Ultra es 1.5 GHz, para obtener 36 GFLOPS pico por SM. Para optimizar consumo y área, algunas unidades del SM, que no forman parte de la vía de datos, tienen un reloj con frecuencia mitad.

El SM tiene soporte para ejecución multihilo como un modo de ejecutar eficientemente cientos de hilos paralelos mientras están ejecutando varios programas diferentes. Gestiona y ejecuta hasta un máximo de 768 hilos concurrentes en hardware con un sobrecoste de planificación cero. Cada hilo tiene su propio estado de ejecución de hilo y puede ejecutar una vía del código independiente.

Una trama consta de un máximo de 32 hilos del mismo tipo: vértices, geometría, píxeles o computación. El diseño SIMT, que se ha descrito en la sección A.4, comparte la unidad de búsqueda y emisión multihilo de instrucciones entre 32 hilos, pero las prestaciones máximas solo se alcanzan con una trama completa de hilos activos.

El SM planifica y ejecuta varios tipos de tramas concurrentemente. En cada ciclo de emisión, el planificador selecciona una de las 24 tramas para ejecutar una instrucción SIMT de trama. La instrucción de trama emitida se ejecuta como cuatro conjuntos de 8 hilos en cuatro ciclos de procesador. Las unidades SP y SFU ejecutan instrucciones independientemente y el planificador mantiene ambas unidades totalmente ocupadas alternando en cada ciclo la emisión de instrucciones para cada unidad. En cada ciclo, un marcador determina si se pueden emitir instrucciones de una trama. El planificador prioriza las tramas disponibles y selecciona la de mayor prioridad. La priorización tiene en cuenta el tipo de trama, el tipo de instrucción y la "imparcialidad" de todas las tramas ejecutándose en el SM.

El SM ejecuta conjuntos de hilos cooperativos (CTA) como si fuesen tramas concurrentes que acceden a regiones de memoria compartida asignada dinámicamente al CTA.

Repertorio de instrucciones

Al contrario que las GPUs anteriores con arquitectura vectorial, los hilos ejecutan instrucciones escalares. Son más sencillas y facilitan el trabajo al compilador. Hay una excepción, las instrucciones de texturas continúan siendo vectoriales, la entrada es un vector de coordenadas y el resultado es un vector de color filtrado.

El repertorio de instrucciones basado en registros incluye las instrucciones aritméticas de punto flotante y enteras, las instrucciones trascendentales, lógicas, de control de flujo, de carga/almacenamiento de memoria y de texturas que se muestran en la tabla de instrucciones PTX de la figura A.4.3. Las instrucciones de carga/almacenamiento de memoria utilizan direccionamiento de byte y la dirección se obtiene a partir del contenido de un registro y un desplazamiento. En aplicaciones de computación, la carga/almacenamiento puede acceder a tres espacios de memoria de lectura-escritura: la memoria local de hilo con datos privados y temporales, la memoria de baja latencia compartida del CTA con datos compartidos por los hilos del CTA y la memoria global con datos compartidos por todos los hilos. Estas aplicaciones utilizan la instrucción de sincronización de barrera `bar.sync` para sincronizar los hilos del CTA, que se comunican a través de la memoria compartida o la memoria global. Las últimas GPUs con arquitectura Tesla implementan operaciones de memoria atómicas de PTX, que facilitan las reducciones paralelas y la gestión de estructuras de datos paralelas.

Procesador de *streaming* (SP)

Tal como se dijo en la sección A.4, el núcleo SP multihilo es el procesador de hilo primordial. Su banco de registros tiene 1024 registros escalares de 32 bits para un máximo de 96 hilos (más hilos que el SP de ejemplo de la sección A.4). Las operaciones de suma y multiplicación punto flotante son compatibles con el estándar IEEE 754 de precisión simple, incluyendo las representaciones de NaN (*not-a-number*) e

infinito, y tienen el redondeo al más próximo par como modo de redondeo por defecto. El núcleo SP implementa también todas las instrucciones de enteros de 32 y 64 bits y las instrucciones de comparación, conversión y lógicas de PTX relacionadas en la figura A.4.3. El procesador está completamente segmentado y su latencia se ha optimizado para equilibrar el retraso y el área.

Unidad de funciones especiales (SFU)

En la SFU se implementan las funciones trascendentales y la interpolación de atributos planares. Como se ha descrito en la sección A.6, se utiliza un algoritmo de interpolación cuadrática basada en una aproximación minimax mejorada para calcular las funciones recíproco, inverso de la raíz cuadrada, $\log_2 x$, 2^x, seno/coseno, obteniendo un resultado por ciclo. Además, en SFU se implementa la interpolación de atributos de píxel, como el color, profundidad, coordenadas de textura, a un ritmo de cuatro muestras por ciclo.

Rasterización

La primitivas geométricas se envían desde el SM al bloque Ventana de Visualización/Recorte/Configuración/Raster/ZCull en su orden de entrada original según un turno rotatorio. Las unidades Ventana de Visualización y Recorte recortan las primitivas de acuerdo con el campo de visión y con cualquier otro plano de recorte especificado por el usuario y transforman los vértices al espacio de visualización (píxel).

Las primitivas se envían a continuación a la unidad de Configuración, que genera las ecuaciones de aristas (*edge equations*) para la rasterización. La etapa de rasterización gruesa genera las teselas de píxeles (*pixel tiles*) que están dentro de la primitiva, al menos parcialmente. La unidad ZCull mantiene una superficie *z* jerárquica, eliminando las teselas de píxeles que están ocultas por otros píxeles que se han dibujado anteriormente. El ritmo de eliminación es de 256 píxeles por ciclo. Los píxeles que sobreviven al ZCull pasan a la etapa de rasterización fina que genera la información de cobertura detallada y los valores de profundidad.

El test y la actualización de la profundidad se realiza anticipadamente al sombreado de fragmentos, o posteriormente, dependiendo del estado actual. El SMC junta los píxeles supervivientes en tramas que serán procesadas en el SM que esté ejecutando el programa de sombreado de píxeles. Finalmente, el SMC envía los píxeles supervivientes y los datos asociados al ROP.

Procesador de operaciones de raster (ROP) y sistema de memoria

Cada ROP está emparejado con una partición de memoria específica. Los ROPs realizan los test de profundidad y cliché de los fragmentos de píxeles emitidos por el programa de sombreado de píxeles y, en paralelo, el mezclado y la actualización del color. Para reducir el ancho de banda de la DRAM utilizan compresión sin pérdidas del color (hasta 8:1) y de la profundidad (hasta 8:1). El ROP tiene un ritmo de procesamiento pico de cuatro píxeles por ciclo y soporta formatos HDR punto flotante de 16 y 32 bits. Cuando la escritura de color está deshabilitada, se dobla el ritmo de procesamiento de la profundidad.

Utilizan multimuestreado y supermuestreado 16× para el anti-aliasing. El algoritmo *coverage-sampling antialiasing* (CSAA) calcula y almacena coberturas boolenas de hasta 16 muestras y comprime la información redundante de color, profundidad y cliché en la memoria de huellas con un ancho de banda de cuatro u ocho muestras.

El bus de datos de DRAM tiene 384 pines, organizados en seis particiones independientes con 64 pines. La partición soporta los protocolos DDR2, de doble ritmo de datos, y GDDR3, orientado a gráficos, a 1 GHz, alcanzando un ancho de banda aproximado de 16 GB/s por partición, o 96 GB/s en total.

Los controladores de memoria reconocen un amplio rango de frecuencias de reloj de DRAM, protocolos, densidad de dispositivos y anchos del bus de datos. Las solicitudes de carga/almacenamiento de texturas pueden proceder de cualquier TPC y dirigirse a cualquier partición de memoria, por lo tanto, se necesita una red de interconexión para guiar las solicitudes y las respuestas.

Escalabilidad

La arquitectura unificada Tesla está diseñada para la escalabilidad. El equilibrio idóneo entre las prestaciones y el coste de los diferentes segmentos del mercado de GPUs se obtiene variando el número de SM, TPCs, ROPs, caches y particiones de memoria. Además, una conexión escalable (*Scalable Link Interconnect*, SLI) permite obtener escalabilidad adicional.

Prestaciones

El reloj de los SPs y SFUs en la GeForce 8800 Ultra es de 1.5 GHz, para unas prestaciones pico de 576 GFLOPS. Por otra parte, la GeForce GTX tiene un reloj de 1.35 GHz y unas prestaciones pico de 518 GFLOPS.

En las tres secciones siguientes se comparan las prestaciones de la GPU GeForce 8800 frente a una CPU multinúcleo con tres aplicaciones diferentes: álgebra lineal densa, transformada rápida de Fourier y ordenación. Los programas y bibliotecas de la GPU se han implementado con CUDA y C. Los programas de la CPU utilizan la biblioteca Intel MKL 10.0 multihilo de precisión simple para obtener una implementación eficiente con instrucciones SSE y varios núcleos.

Prestaciones con álgebra lineal densa

Las operaciones de álgebra lineal densa son fundamentales en muchas aplicaciones. Volkov y Demmel [2008] presentan resultados de prestaciones en GPU y CPU para varias operaciones sobre matrices densas en precisión simple: multiplicación matriz-matriz (la rutina SGEMM) y factorizaciones LU, QR y Cholesky. En la figura A.7.3 se comparan los GFLOPS obtenidos en la multiplicación SGEMM de matrices densas de una GPU GeForce 8800 GTX y una CPU de cuatro núcleos. Asimismo, la figura A.7.4 muestra la comparación de la ejecución de varios algoritmos de factorización.

Como la mayor carga computacional de la factorización de matrices corresponde a la multiplicación matriz-matriz de SGEMM y rutinas similares en BLAS3, sus prestaciones fijan un límite superior en el ritmo de factorización. Cuando el tamaño de las

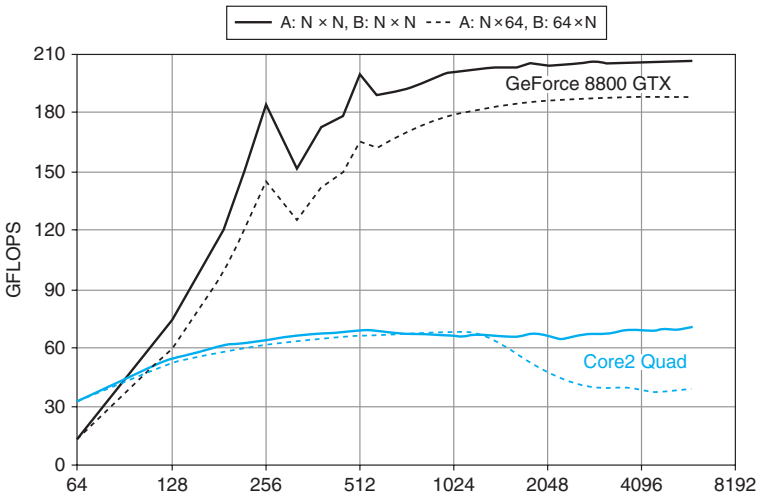


FIGURA A.7.3 Prestaciones de la multiplicación SGEMM de matrices densas. La gráfica muestra los GFLOPS obtenidos en la multiplicación de matrices cuadradas N×N (línea sólida) y matrices N×64 y 64×N (líneas punteadas). Extraída de la figura 6 de Volkov y Demmel [2008]. Las gráficas en color negro corresponden a una GeForce 8800 GTX a 1.35 GHz ejecutando el código SGEMM de Volkov (actualmente en CUBLAS 2.0 de NVIDIA) con matrices en la memoria de la GPU. Las gráficas en azul corresponden a la ejecución en un Intel Core2 Quad Q6600 de cuatro núcleos a 2.4 GHz, con Linux de 64 bits y la biblioteca Intel MKL con matrices en la memoria de la CPU.

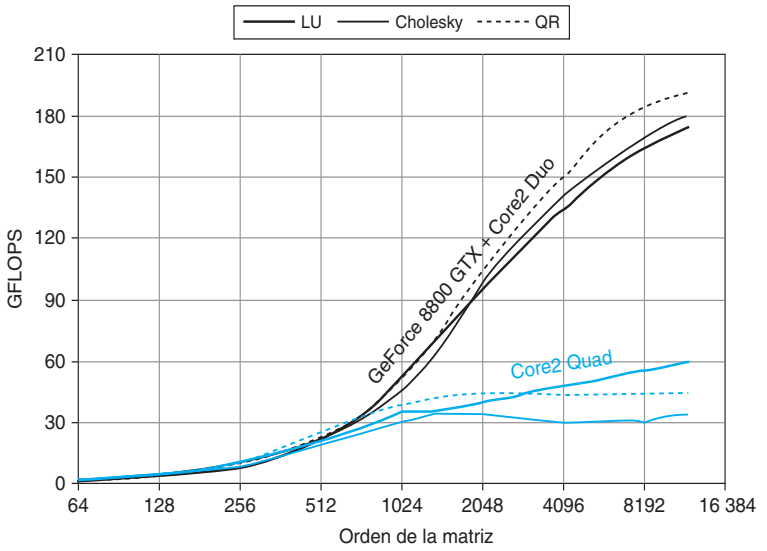


FIGURA A.7.4 Prestaciones de la factorización de matrices densas. Las gráficas muestran los GFLOPS alcanzados en factorización de matrices utilizando la GPU y utilizando sólo la CPU. Extraída de la figura 7 de Volkov y Demmel [2008]. Las gráficas en color negro corresponden a un sistema con una GeForce 8800 GTX de NIVIDIA a 1.35 GHz, CUDA 1.1, Windows XP y un procesador Intel Core2 Duo E6700 a 2.67 GHz con Windows XP, incluyendo el tiempo de transferencia de datos entre CPU y GPU. Las gráficas en azul corresponden a la ejecución en un Intel Core2 Quad Q6600 de cuatro núcleos a 2.4 GHz, con Linux de 64 bits y la biblioteca Intel MKL.

matrices es mayor de 200 o 400, la carga computacional de la factorización es lo suficientemente elevada como para que el paralelismo de la GPU supere el sobrecoste del sistema GPU-CPU y de la copia. La multiplicación de matrices SGEMM de Volkov alcanza los 206 GFLOPS, aproximadamente el 60% del ritmo de multiplicaciones-suma pico de la GeForce GTX, mientras que la factorización QR alcanza los 192 GFLOPS, aproximadamente 4.3 veces los GFLOPS de la CPU con cuatro núcleos.

Prestaciones de la FFT

La transformada rápida de Fourier se utiliza en muchas aplicaciones. Las transformadas de muchos puntos o multidimensionales se dividen en grupos de transformadas 1D más pequeñas.

La figura A.7.5 las prestaciones de la implementación de una FFT compleja de precisión simple en una GeForce 8800 GTX a 1.35 GHz (de finales de 2006) con la implementación en Intel Xeon E5462 de cuatro núcleos a 2.8 GHz (de finales de 2007). Las prestaciones de la CPU se midieron utilizando la FFT de la biblioteca de núcleos computacionales MKL 10.0 de Intel con cuatro hilos. Las prestaciones de la GPU se midieron utilizando la biblioteca CUFFT 2.1 de NIVIDIA con una descomposición en FFT 1D radix 16 con decimación en frecuencia. Tanto en la GPU como en la CPU se utilizaron $2^{24}/n$ grupos siendo n el tamaño de la FFT. Por lo tanto, la carga de trabajo de cada tamaño de transformada es 128 MB. Para determinar los GFLOPS, se han considerado $5n\log_2 n$ operaciones por transformada.

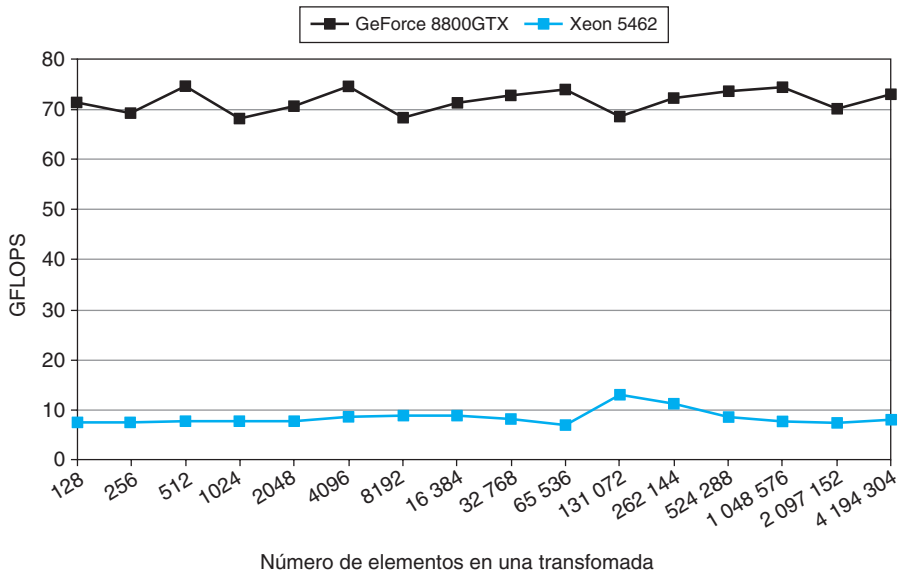


FIGURA A.7.5 Prestaciones de la transformada rápida de Fourier. Las gráficas comparan las prestaciones de FFTs complejas unidimensionales en un GeForce GTX a 1.35 GHz frente a un Intel Xeon E5462 con cuatro núcleos a 2.8 GHz, 6 MB de cache L2.4 GB de memoria, 1600 FSB, Linux Red Hat e Intel MKL 10.0.

Prestaciones de la ordenación

A diferencia de las aplicaciones anteriores, la ordenación requiere una mayor coordinación entre hilos paralelos, y en consecuencia es más difícil obtener escalabilidad. Sin embargo, varios algoritmos de ordenación se pueden paralelizar eficientemente para su ejecución en la GPU. El diseño de algoritmos de ordenación en CUDA se describe de forma detallada en Satish et al. [2008]. A continuación se resumen los resultados que han obtenido.

La figura A.7.6 compara las prestaciones de varios algoritmos de ordenación paralela en una GeForce 8800 Ultra y en sistema Intel Clovertown de ocho núcleos, ambos de comienzo de 2007. Los ocho núcleos de la CPU están distribuidos en dos ranuras, cada una con un módulo multichip con chips Core2 iguales, y cada chip tiene una cache L2 de 4 MB. Los algoritmos de ordenación se diseñaron para ordenar parejas clave-valor, siendo tanto la clave como el valor enteros de 32 bits. El principal algoritmo estudiado es *radix sort*, aunque, por comparación, se ha incluido también el procedimiento `parallel_sort` basado en *quicksort* e incluido en los bloques de construcción de hilos (*Threading Building Blocks*) de Intel. De los dos códigos de ordenación *radix sort* basados en CPU, uno se ha implementado utilizando únicamente el repertorio de instrucciones escalares y el otro utilizando rutinas en lenguaje ensamblador cuidadosamente diseñadas para aprovechar las instrucciones vectoriales SSE2 SIMD.

La gráfica muestra el ritmo de ordenación, definido como el número de elementos ordenados dividido entre el tiempo de ordenación, para varios tamaños. Se aprecia claramente que el mayor ritmo de ordenación se obtiene con el algoritmo *radix sort* en la GPU

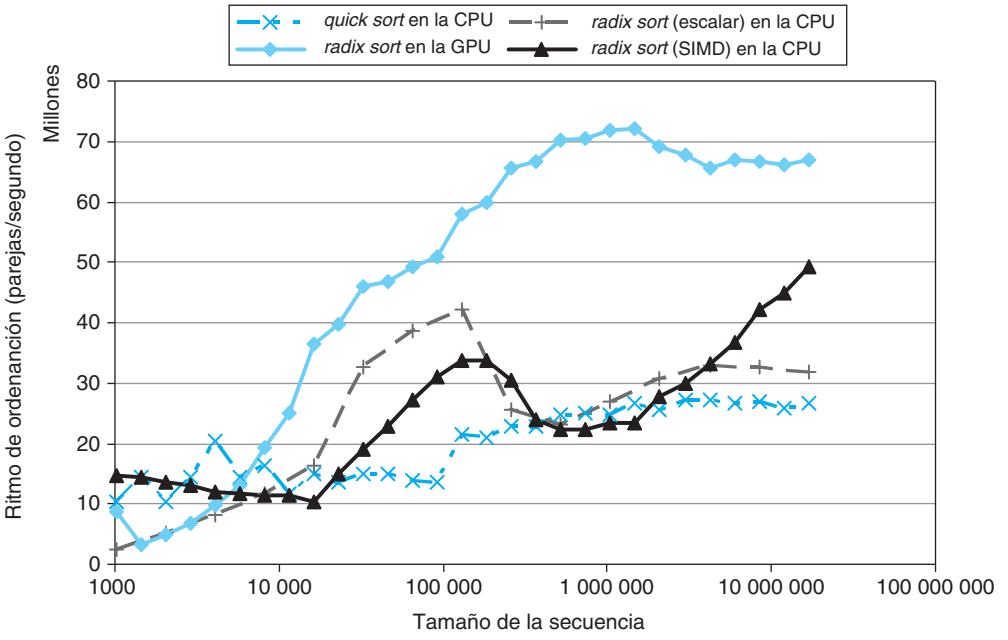


FIGURA A.7.6 Prestaciones de la ordenación paralela. Esta gráfica compara los ritmos de ordenación de implementaciones paralelas del *radix sort* en una GeForce 8800 Ultra a 1.5 GHz y en un Intel Core2 Xeon E5345 con ocho núcleos a 2.33 GHz.

para secuencias a partir de 8000 elementos. En este rango, es aproximadamente 2.6 veces más rápido que la rutina basada en el algoritmo *quicksort* y 2 veces más rápido que la rutina radix sort, ambos en la CPU. Las prestaciones de *radix sort* en la CPU sufren grandes variaciones, probablemente debido a la pobre localidad de la permutaciones globales.

A.8 Casos reales: Implementación de aplicaciones en la GPU

La llegada de las CPU multinúcleo y de las GPU con muchos núcleos (*manycore*) significa que los procesadores dominantes en la actualidad son sistemas paralelos. Además, el paralelismo de estos procesadores sigue aumentando de acuerdo con la ley de Moore. Por lo tanto, el desafío al que nos enfrentamos es desarrollar aplicaciones de computación visual y computación de altas prestaciones que escalen su paralelismo de forma transparente para aprovechar el cada vez mayor número de procesadores disponibles, de forma similar a como escalan las aplicaciones de gráficos 3D en GPUs con un número variable de núcleos.

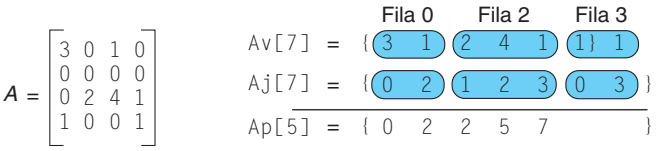
En esta sección se presentan ejemplos de proyección con CUDA de aplicaciones de computación paralelas escalables en la GPU.

Matrices dispersas

Una gran variedad de algoritmos paralelos pueden escribirse en CUDA de forma bastante sencilla, incluso cuando las estructuras de datos implicadas no son mallas regulares sencillas. Un buen ejemplo es la multiplicación matriz dispersa-vector (SpMV), una operación importante en muchas aplicaciones numéricas que se puede paralelizar de forma bastante directa utilizando las abstracciones de CUDA. En efecto, los núcleos computacionales que discutiremos más adelante, en combinación con las rutinas vectoriales de CUBLAS, hacen que la codificación de resolutores iterativos como el método del gradiente conjugado sea bastante sencilla.

Una matriz dispersa $n \times n$ es una matriz en la que el número de elementos distintos de cero, m, es una pequeña fracción del total de elementos. Las representaciones de este tipo de matrices buscan almacenar sólo los $m = O(n)$ elementos distintos de cero, lo cual representa una ahorro sustancial en espacio de almacenamiento y tiempo de procesamiento.

Una de las representaciones más utilizadas para las matrices dispersas no estructuradas generales es la representación fila dispersa comprimida (*Compressed Sparse Row*, CSR). Los m elementos distintos de cero de la matriz `A` se almacenan en un vector `Av` siguiendo el orden de las filas. Un segundo vector `Aj` registra el índice de columna que le corresponde a cada elemento en `Av`. Finalmente, un vector `Ap` de $n + 1$ elementos almacena el número de elementos de cada fila en los vectores anteriores; el número de elementos de la fila i en `Aj` y `Av` se extiende desde el índice `Ap[i]` en `Av` y `Aj` hasta el índice `Ap[i + 1]` no incluido. Esto implica que `Ap[0]` siempre es cero y `Ap[n]` siempre es m, el número de elementos distintos de cero en la matriz. La figura A.8.1 muestra un ejemplo de la representación CSR para una matriz sencilla.



a. Matriz *A* de ejemplo b. Representación CSR de la matriz

FIGURA A.8.1 Representación CSR de una matriz.

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, //índices de la columna
                                     para fila
                   float *Av,       // elementos distintos de
                                     cero de fila
                   float *x)
{
    float sum = 0;

    for(unsigned int column = 0; column< rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
```

FIGURA A.8.2 Código C serie para una fila de la multiplicación matriz dispersa-vector.

Entonces, dada un matriz *A* en representación CSR y un vector *x*, cada fila del producto $y = Ax$ se calcula con la función `multiply_row()` de la figura A.8.2. El cálculo del producto completo es simplemente un lazo que recorre todas las filas y llama, para cada fila, a la función `multiply_row()`, tal como se hace en el código C serie de la figura A.8.3.

El algoritmo puede traducirse a un núcleo CUDA paralelo de forma bastante sencilla. Simplemente distribuimos el lazo en `csrmul_serial()` en varios hilos paralelos, de forma que cada hilo calcule una fila del vector resultado *y*. El código se muestra en la figura A.8.4. Observe que es muy similar al lazo serie de `csrmul_serial()`, solamente hay dos diferencias. Primero, el índice `row` de cada hilo se obtiene a partir de los índices de bloque e hilo asignados a cada hilo, eliminando el lazo for. Segundo, hemos incluido una sentencia condicional para que sólo se evalúe una fila del producto si el índice de fila está dentro de los límites de la matriz (esto es necesario porque el número de filas *n* no tiene por qué ser un múltiplo del tamaño del bloque utilizado para lanzar el núcleo computacional).

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
     for (unsigned int row = 0; row < num_rows; ++row)
     {
         unsigned int row_begin = Ap[row];
         unsigned int row_end   = Ap[row+1];

         y[row] = multiply_row(row_end-row_begin, Aj+row_begin,

                               Av+row_begin, x);
     }
}
```

FIGURA A.8.3 Código serie para la multiplicación matriz dispersa-vector.

```
__global__
void csrmul_kernel (unsigned int *Ap, unsigned int *Aj,
                    float *Av, unsigned int num_rows,
                    float *x, float *y)
{
     unsigned int row = blockIdx.x*blockDim.x + theradIx.x;

     if( row<num_rows )
     {
         unsigned int row_begin = Ap[row];
         unsigned int row_end   = Ap[row+1];

         y[row] = multiply_row(row_end-row_begin, Aj+row_begin,

                               Av+row_begin, x);
     }
}
```

FIGURA A.8.4 Versión CUDA de la multiplicación matriz dispersa-vector.

Suponiendo que las estructuras de datos matriciales ya han sido copiadas en el sistema de memoria de la GPU, el lanzamiento de este código sería algo similar a

```
unsigned int blocksize = 128; // o cualquier tamaño menor o igual a 512
unsigned int nblocks = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

El patrón que hemos visto en este ejemplo es bastante habitual: el algoritmo serie original es un lazo con iteraciones independientes. Este tipo de lazos son fácilmente paralelizables simplemente asignando una o más iteraciones a cada hilo paralelo. La formalización de este tipo de paralelismo es particularmente sencilla con el modelo de programación CUDA.

Esta estrategia general de descomponer una computación en bloques independientes y, más específicamente, partir las iteraciones independientes de un lazo, no es exclusivo de CUDA. Es una estrategia habitual en varios sistemas de programación paralela, incluyendo OpenMP y Threading Building Blocks de Intel.

Caches y memoria compartida

Los algoritmos SpMV que acabamos de resumir son bastante simples. Se pueden implementar varias optimizaciones, tanto en el código de la CPU como en el código de la GPU, para mejorar las prestaciones; entre estas optimizaciones podemos incluir el desenrollamiento de lazos, la reordenación de la matriz y el agrupamiento de los registros en bloques. También se pueden reimplementar los núcleos paralelos en términos de las operaciones *scan* de paralelismo de datos presentadas por Sengupta y al. [2007].

Una de las características arquitecturales explotada por CUDA es la memoria compartida entre los hilos de un bloque, una pequeña memoria de baja latencia integrada en el chip del procesador. Es posible conseguir mejoras significativas en las prestaciones si se aprovecha adecuadamente esta memoria. Una forma de hacerlo es utilizar esta memoria compartida como una cache gestionada por software que almacena los datos reutilizados frecuentemente. En la figura A.8.5 se muestra código SpMV que aprovecha esta memoria.

En el contexto de la multiplicación de una matriz dispersa, observamos que varias filas de *A* usan un mismo elemento `x[i]` del vector. En muchos casos, y particularmente cuando la matriz se ha reordenado, las filas que necesitan el elemento `x[i]` son filas próximas a la fila *i*. Por lo tanto, podemos implementar un esquema simple que se base en la utilización de una cache y obtener algunas mejoras en las prestaciones. Así, los bloques de hilos que procesan las filas *i* a *j* cargarán los elementos entre `x[i]` y `x[j]`, ambos incluidos, en su memoria compartida. Entonces, se desenrolla el lazo `multiply_row()` y se toman los elementos de `x` de la cache siempre que sea posible. El código resultante es el mostrado en la figura A.8.5. La memoria compartida también puede utilizarse para implementar otras optimizaciones, por ejemplo, obtener `Ap[row+1]` de un hilo adyacente en lugar buscarlo otra vez en memoria.

Como la arquitectura Tesla proporciona una memoria compartida integrada en el chip gestionada de forma explícita, en lugar de una cache hardware activa de forma implícita, es bastante sencillo incorporar esta clase de optimizaciones. Aunque esto puede imponer alguna carga adicional al programador, ésta es relativamente pequeña, y los beneficios que se pueden obtener son muy significativos. En el ejemplo anterior, esa utilización de la memoria compartida, que ha sido fácil de implementar, proporciona una mejora en las prestaciones del 20%, aproximadamente, con matrices representativas obtenidas de mallas de superficies 3D. Además, la disponibilidad de una memoria gestionada explícitamente en lugar de una cache implícita tiene la ventaja adicional de que las estrategias de gestión de la cache y prebúsqueda pueden hacerse a medida de las necesidades de la aplicación.

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
     // la filas x[] de ese bloque se llevan a la cache
     __shared__ float cache[blocksize];

      unsigned int block_begin = blockIdx.x * blockDim.x;
      unsigned int block_end   = block_begin + blockDim.x;
      unsigned int row         = block_begin + threadId.x;

     // Buscar y almacenar en cache la ventana de x[]
     if (row < num_rows) cache[threadIdx.x] = x[row];
     __syncthreads();

     if( row<num_rows )
     {
         unsigned int row_begin = Ap[row];
         unsigned int row_end   = Ap[row+1];
         float sum = 0, x_j;

         for(unsigned int col=row_begin; col<row_end; ++col)
         {
             unsigned int j = Ap[col];

            // Lectura de x_j de la cache cuando sea posible
            if ( j >= block_begin && j< block_end )
                  x_j = cache[j-block_begin];
            else
                  x_j = x[i];

            sum += Av[col] * x_j;
          }

          y[row] = sum;
      }
}
```

FIGURA A.8.5 Versión de memoria compartida de la multiplicación matriz dispersa-vector.

Estos núcleos computacionales son bastante sencillos y su propósito es ilustrar algunas técnicas básicas del diseño de programas CUDA y no pretende ser una guía para obtener unas prestaciones máximas. Hay muchas formas de optimizar los programas paralelos, algunas de ellas han sido exploradas por Williams y al. [2007] en un puñado de arquitecturas multinúcleo diferentes. Sin embargo, es ilustrativo comparar las prestaciones de estos núcleos computacionales sencillos. En un procesador Intel Core2 Xeon E5335 a 2 GHz, el núcleo computacional `crsmul_serial()` procesa 202 millones de elementos distintos de cero por segundo, para un conjunto de matrices laplacianas obtenidas de mallas de superficies trianguladas 3D. En la paralelización de este núcleo computacional utilizando la estructura `parallel_for` del Threading Building Blocks de Intel se obtiene una aceleración de 2.0, 2.1 y 2.3 con dos, cuatro y ocho núcleos respectivamente. En la GeForce 8800 Ultra, los núcleos computacionales `csrmul_kernel()` y `csrmul_cached()` alcanzan ritmos de procesamiento de 771 y 920 millones de elementos distintos de cero por segundo, lo que supone una aceleración de 3.8 y 4.6 respecto a la implementación en una CPU con un núcleo.

Scan y reducción

El *scan* paralelo, también conocido como suma *prefix* paralela, es uno de los bloques de construcción de algoritmos más importante en el paralelismo de datos [Blelloch, 1990]. Dada una secuencia de n elementos:

$$[a_0, a_1, \ldots, a_{n-1}]$$

y un operador binario asociativo $\oplus$, la función `scan` realiza la siguiente operación:

$$\texttt{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

Por ejemplo, si el $\oplus$ es el operador suma, entonces el scan del siguiente conjunto de datos

$$a = [3\ 1\ 7\ 0\ 4\ 1\ 6\ 3]$$

es la siguiente secuencia de sumas parciales:

$$\texttt{scan}(a, +) = [3\ 4\ 11\ 11\ 15\ 16\ 22\ 25]$$

El operador scan así definido es un scan *inclusivo*, en el sentido de que el elemento i de la secuencia de salida incluye el elemento a_i de la secuencia de entrada. Si el scan incorporase sólo los elementos anteriores, sería un scan *exclusivo,* también conocido como operación *prefix-sum*.

La implementación serie de esta operación es extremadamente simple. Consiste en un lazo que itera una vez sobre la secuencia entera, como se muestra en la figura A.8.6.

A primera vista puede parecer que esta operación es inherentemente serie. Sin embargo, puede implementarse en paralelo de forma eficiente. El punto clave es la suma asociativa: podemos cambiar con total libertad el orden en el que se suman los elementos. Por ejemplo, podemos sumar pares de elementos consecutivos en paralelo, después sumar esas sumas parciales y así sucesivamente.

```
template>class T>
__host__ T plus_scan(T *x, unsigned int n)
{
      for(unsigned int i=1; i<n; ++i)
          x[i] = x[i-1] + x[i]
}
```

FIGURA A.8.6 Plantilla de plus-scan serie.

```
template<class T>
__device__ T Plus_scan(T *x)
{
      unsigned int i = threadIdx.x;
      unsigned int n = blockDim.x;

      for (unsigned int offset=1; offset< n; offset *= 2)
      {

          T t;

          if(i >= offset) t = x[i-offset];
          __syncthreads();

          if(i >= offset)  x[i] = t + x[i];
          __syncthreads();
      }
      return x[i];
}
```

FIGURA A.8.7 Plantilla CUDA de plus-scan paralelo.

La implementación CUDA de un algoritmo simple extraído de Hillis y Steele [1989] se muestra en la figura A.8.7. Supone que el conjunto de entrada `x[]` tiene un elemento por hilo del bloque de hilos. Tiene un lazo de $\log_2 n$ iteraciones que recoge las sumas parciales.

Para entender lo que hace este lazo, consideremos el ejemplo para $n = 8$ hilos y elementos que se muestra figura A.8.8. Cada nivel del diagrama representa una iteración del lazo y las líneas especifican la posición en la que se busca el dato. Para cada elemento del conjunto de salida (es decir, la última final del diagrama) se construye un árbol de sumas con elementos del conjunto de entrada. A modo de ejemplo, las líneas resaltadas en azul muestran los elementos del conjunto de entrada que forman parte del árbol del último elementos de la salida. Las hojas del árbol son todos los elementos del conjunto de entrada. Rastreando los elementos del conjunto de entrada que forman parte del árbol de cada elemento del conjunto de salida, nos daremos cuenta que incorpora todos las entradas hasta e incluyendo la que tiene su mismo índice.

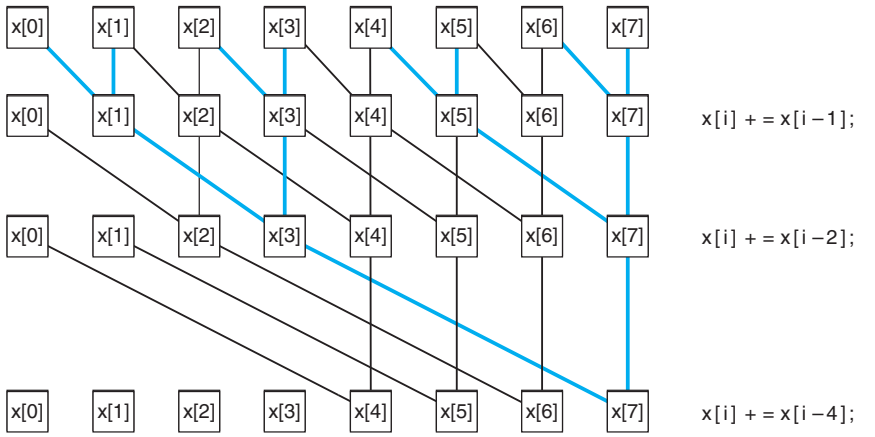


FIGURA A.8.8 Referencias a los datos en un scan paralelo basado en un árbol.

Aunque simple, este algoritmo no es tan eficiente como nos gustaría. Examinando la implementación serie, vemos que se hacen *O(n)* sumas. En comparación, la implementación paralela tiene *O(n* log *n)* sumas. Por este motivo no es una implementación eficiente, ya que se hace más trabajo en la versión paralela que en la versión serie para obtener el mismo resultado. Afortunadamente, hay otras técnicas más eficientes para implementar la función `scan`. En Senguprta y al. [2007] se pueden encontrar otras implementaciones más eficientes, descritas con todo detalle, y la extensión de esta implementación por bloques a conjuntos multibloque.

En algunos casos podríamos estar interesados únicamente en obtener la suma de todos los elementos del conjunto de entrada, en lugar de la secuencia completa de sumas prefix. Este problema se conoce con el nombre de *reducción paralela*. Podríamos utilizar el scan para hacer la reducción, pero generalmente ésta puede ser implementada más eficientemente que el scan.

El código para el cálculo de la reducción con sumas se muestra en la figura A.8.9. Cada hilo carga un elemento de la secuencia de entrada (es decir, inicialmente suma una subsecuencia de longitud 1) y al final de la reducción, el hilo 0 tiene la suma de todos los elementos inicialmente cargados por los hilos del bloque. El lazo en este núcleo computacional implícitamente construye un árbol de sumas de los elementos de entrada, de una forma muy parecida a como lo hace el algoritmo de scan anterior.

Al final del lazo, el hilo 0 tiene la suma de todos los valores cargados por este bloque. Si se quiere que el valor final en la posición apuntada por total sea la suma `total` de todos los elementos del conjunto de entrada, tendremos que combinar las sumas parciales obtenidas por todos los bloques de la malla. Se podría hacer que cada bloque escribiese su suma parcial en un segundo conjunto y lanzar el núcleo computacional de reducción otra vez, repitiendo el proceso hasta que la secuencia haya sido reducida a un único valor. Una alternativa más atractiva es usar la primitiva `atomicAdd()` de Tesla, una primitiva atómica de lectura-modificación-escritura soportada por el subsistema de memoria. Con esto se evita la utilización de conjuntos temporales adicionales y la ejecución repetida de núcleos computacionales.

```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
      unsigned int tid = threadIdx.x;
      unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;

      // Cada bloque carga sus elementos en memoria compartida, rellenando
      // con 0 si N no es un múltiplo del tamaño de bloque
      __shared__ int x[blocksize];
      x[tid] = (i<N) ? input[i] : 0;
      __syncthreads();

      // Cada hilo tiene 1 valor de entrada en x[]
      //
      // Construcción del árbol de sumas
      for (int s=blockDim.x/2; s>0; s=s/2)
      {
          if(tid < s)  x[tid] += x[tid+s];
          __syncthreads();
      }

      // El hilo 0 tiene la suma de todos los valores de entrada
      // de este bloque. Tiene que sumar este valor con la suma total
      if (tid == 0)  atomicAdd(total, x[tid]);
}
```

FIGURA A.8.9 Implementación CUDA de plus-reduction.

La reducción paralela es una primitiva esencial en la programación paralela y pone de manifiesto la importancia de la memoria compartida entre bloques y de las barreras de bajo coste para la cooperación entre hilos. Este grado de barajamiento de datos entre hilos tendría un coste prohibitivo si se utilizase la memoria externa.

Radix Sort (ordenación radix)

Una aplicación importante de las primitivas de scan es la implementación de rutinas de ordenación. El código de la figura A.8.10 implementa una ordenación radix de enteros con un bloque de hilos. El conjunto de entrada `values` tiene un entero de 32 bits para cada hilo del bloque; por motivos de eficiencia, este conjunto debe almacenarse en la memoria compartida del bloque, pero aunque no se haga así la ordenación se hace correctamente.

Ésta es una implementación bastante sencilla de la ordenación radix. Se supone que se dispone de una función `partition_by_bit()` que divide el conjunto de entrada de forma que los valores con un 0 en el bit indicado estarán antes que los valores con este bit a 1. Para que la salida sea correcta esta partición tiene que ser estable.

```
__device__ void radix_sort(unsigned int *values)
{
      for (int bit = 0; bit <32; ++bit)
      {
          partition_by_bit(values, bit);
          __syncthreads();
      }
}
```

FIGURA A.8.10 Código CUDA para la ordenación radix.

La partición se puede implementar con scan. El hilo i tiene el valor x_i y debe calcular el índice de salida en el que escribir este valor. Para ello, tiene que calcular (1) el número de hilos $j < i$ para los que el bit indicado es 1 y (2) el número total de hilos para los que el bit es 0. El código CUDA para `partition_by_bit()` se muestra en la figura A.8.11.

```
__device__ void partition_by_bit(unsigned int *values,
                                 unsigned int bit)
{
      unsigned int i    = threadIdx.x;
      unsigned int size = blockDim.x;
      unsigned int x_i  = values[i];
      unsigned int p_i  = (x_i >> bit) & 1;

      values[i] = p_i;
      __syncthreads();

      // Calcula el número T de bits hasta p_i incluido
      // Registra el número total F de bits.
      unsigned int T_before = plus_scan(values);
      unsigned int T_total  = values[size-1];
      unsigned int F_total  = size - T_total;
      __syncthreads();

      // Escribe cada x_i en su lugar correcto
      if (p_i)
          values[T_before-1 + F_total] = x_i;
      else
          values[i - T_before] = x_i;
}
```

FIGURA A.8.11 Código CUDA para dividir los datos bit-a-bit, como parte de la ordenación radix.

Para la ordenación de un conjunto de elementos de longitud mayor que un bloque se puede utilizar una estrategia similar. El paso fundamental sigue siendo el scan, aunque al distribuir la computación entre varios núcleos computacionales tendremos que utilizar un doble búfer para el almacenamiento de los valores, en lugar de hacer la distribución de forma local. Satish, Harris y Garland [2008] proporcionan una descripción detallada de la ordenación eficiente de grandes conjuntos de datos.

Aplicaciones N-cuerpos en una GPU[2]

Nyland, Harris y Prins [2007] describen un simple pero útil núcleo computacional con excelentes prestaciones en la GPU, el algoritmo *all-pairs N-body*. Éste es un componente de muchas aplicaciones científicas que consume mucho tiempo de ejecución. Las simulaciones de N-cuerpos calculan la evolución de un sistema de cuerpos en el cual cada cuerpo está interaccionando continuamente con los otros cuerpos del sistema. Por ejemplo, una simulación en astrofísica en la que cada cuerpo representa una estrella y los cuerpos se atraen mutuamente por la fuerza de la gravedad. Otros ejemplos son el doblado de proteínas, donde la simulación de N-cuerpos se utiliza para calcular las fuerzas electrostáticas y de Van der Waals, la simulación de turbulencias en fluidos y la iluminación global en computación gráfica.

El algoritmo *all-pairs N-body* calcula la fuerza total sobre un cuerpo del sistema a partir de las fuerzas entre pares de cuerpos; una vez obtenidas éstas, sólo queda hacer la suma para cada cuerpo. Muchos científicos consideran que este método es el más exacto, ya que solamente se pierde precisión en las operaciones punto flotante. Sin embargo, su desventaja es la complejidad algorítmica, $O(n^2)$, demasiado elevada para sistema con más de 10^6 cuerpos. Para superar este elevado coste se han propuesto diversas modificaciones que reducen la complejidad algorítmica a $O(n \log n)$ y $O(n)$; algunos ejemplos son el algoritmo de Barnes-Hut, el método del multipolo rápido y el método PME (*Particle-Mesh-Ewald*). En todos estos métodos rápidos se utiliza también el método all-pairs como un núcleo computacional para el cálculo exacto de fuerzas de rango corto; por lo tanto, este método sigue siendo de interés.

Matemática de los N-cuerpos

Calcular la fuerza cuerpo-cuerpo utilizando física elemental en una aplicación de simulación gravitacional. El vector fuerza 3D entre dos cuerpos de índices i y j es:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{||\mathbf{r}_{ij}||^2} \times \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}$$

La magnitud de la fuerza se calcula con el término de la izquierda y la dirección con el de la derecha (vector unitario apuntando de un cuerpo a otro).

Dada una lista de cuerpos que interaccionan (un sistema completo o un subconjunto), el cálculo es sencillo: para todos las parejas de interacciones, se calcula la fuerza y se suma para cada cuerpo. Una vez que se han calculado todas las fuer-

[2] Extraído de Nyland, Harris y Prins [2007], "Fast N-body Simulation with CUDA", capítulo 3 de *GPU Gems 3*.

zas, se utilizan para la actualizar la posición y la velocidad de los cuerpos a partir de las posiciones y velocidades anteriores. El cálculo de las fuerzas tiene complejidad $O(n^2)$ y la actualización tiene complejidad $O(n)$.

El código serie para el cálculo de fuerzas tiene dos lazos anidados con iteraciones para los diferentes pares de cuerpos. El lazo externo selecciona el cuerpo para el que se está calculando la fuerza total y el lazo interno itera sobre todos los cuerpos. El lazo interno llama a una función que calcula las fuerzas entre los pares de cuerpos y acumula la fuerza resultante en la fuerza total con una suma iterativa.

Para calcular las fuerzas en paralelo asignamos un hilo a cada cuerpo, porque el cálculo de la fuerza en cada cuerpo es independiente del cálculo en los otros cuerpos. Una vez calculadas todas las fuerzas, se actualizan la posición y la velocidad de los cuerpos.

Los códigos para la versiones serie y paralelo se muestran en la figuras A.8.12 y A.8.13, respectivamente. La versión serie tiene dos lazos anidados. Al igual que en otros muchos ejemplos, la conversión a CUDA tranforma el lazo externo en el núcleo computacional del hilo, para que cada hilo compute la fuerza total en un cuerpo. El núcleo computacional de CUDA calcula una identificación (ID) global para cada hilo, reemplazando la variable de control del lazo externo. Ambos núcleos computacionales terminan almacenando la aceleración total en un vector global que se utiliza para calcular la nueva posición y la nueva velocidad en otra iteración.

```
Void accel_on_all_bodies()
{
    int i, j;
    float3 acc(0.0f, 0,0f, 0.0f);

    for (i = 0; i < N; i++) {
        for (j=0; j < N; j++) {
            acc = body_body_interaction(acc, body[i], body[j]);
        }
        accel[i] = acc;
    }
}
```

FIGURA A.8.12 Código serie para el cálculo de todos los pares de fuerzas en un sistema de N cuerpos.

```
__global__ void  accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j;
    float3 acc(0.0f, 0,0f, 0.0f);

    for (j=0; j<N; j++){
        acc = body_body_interactions(acc, body[i], body[j]);
    }
    accel[i] = acc;
}
```

FIGURA A.8.13 Código CUDA para el cálculo de la fuerza total en un cuerpo.

El lazo externo se reemplaza por una malla de núcleos computacionales CUDA que lanza N hilos, uno por cada cuerpo.

Optimización para la ejecución en la GPU

El código CUDA mostrado es funcionalmente correcto, pero no es eficiente porque ignora algunas características claves de la arquitectura. Con las tres siguientes optimizaciones básicas se mejoran las prestaciones. Primero, se puede utilizar la memoria compartida para evitar lecturas de memoria idénticas en varios hilos. Segundo, la utilización de varios hilos por cuerpo mejora las prestaciones para valores de N pequeños. Tercero, el desenrollamiento de lazos reduce el sobrecoste de los lazos.

Utilización de la memoria compartida

La memoria compartida puede almacenar un subconjunto de posiciones de cuerpos, tal como haría una cache, evitando solicitudes redundantes a la memoria global desde varios hilos. Optimizamos el código mostrado anteriormente para que cada uno de los *p* hilos del bloque cargue una posición en la memoria compartida (hasta un total de *p* posiciones). Una vez que todos los hilos han cargado un valor en la memoria compartida, de lo que nos aseguramos con `_syncthread()`, cada hilo hace *p* iteraciones (utilizando los datos de la memoria compartida). Repetimos esta operación *N/p* veces para completar el cálculo de la fuerza en cada cuerpo, lo que permite reducir el número de solicitudes de memoria en un factor *p* (típicamente en el rango 32–128).

La función `accel_on_one_body()` precisa pocos cambios para implementar esta optimización. El código modificado se muestra en la figura A.8.14.

```
__shared__ float4 shPosition[256];
...
__global__ volid accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j, k;
    int p = blockDim.x;
    float3 acc(0.0f, 0,0f, 0.0f);
    float4 myBody = body[i];

    for (j = 0; j <N; j+= p){ // el lazo externo salta de p en p
        shPosition[threadIdx.x] = body[j+threadIdx.x];
        __syncthreads();
        for (k = 0; k < p; k++){ // el lazo interno accede a p posiciones
            acc = body_body_interaction(acc, myBody, shPosition[k]);
        }
        __syncthreads();
    }
    accel[i] = acc;
}
```

FIGURA A.8.14 Código CUDA para el cálculo de la fuerza total en un cuerpo, utilizando la memoria compartida para mejorar las prestaciones.

El lazo que anteriormente iteraba sobre todos los cuerpos, ahora salta de p en p, siendo p el tamaño del bloque. En cada iteración del lazo externo se cargan p posiciones consecutivas en la memoria compartida (una posición por hilo). Los hilos se sincronizan y, a continuación, cada hilo calcula p fuerzas. Se requiere una segunda sincronización para asegurar que los nuevos valores no se cargan en la memoria compartida antes de que todos los hilos hayan terminado el cálculo de fuerzas con los datos actuales.

La utilización de la memoria compartida reduce las necesidades de ancho de banda a menos del 10% del ancho de banda total que puede proporcionar la GPU (utiliza menos de 5 GB/s). Esta optimización mantiene la aplicación ocupada en los cálculos, en lugar de esperando por lo accesos a memoria como ocurriría si no se utiliza la memoria compartida. En la figura A.8.15 se muestran las prestaciones para varios valores de N.

Utilización de varios hilos por cuerpo

La figura A.8.15 muestra la degradación de las prestaciones para N pequeño ($N < 4096$) en la GeForce 8800 GTX. Muchas aplicaciones científicas basadas la simulación de N-cuerpos tienen un N pequeño (y largos tiempos de simulación), por lo que centraremos nuestros esfuerzos de optimización en estos casos. Nuestra explicación de las bajas prestaciones fue simplemente que, con N pequeño, no había el trabajo suficiente como para mantener la GPU ocupada. La solución es asignar más hilos por cuerpo. Así, cambiamos las dimensiones del bloque de hilos de (p, 1, 1) a (p, q, 1), es decir, el trabajo sobre un cuerpo se reparte a partes iguales entre q hilos. Si los hilos adicionales pertenecen al mismo bloque de hilos,

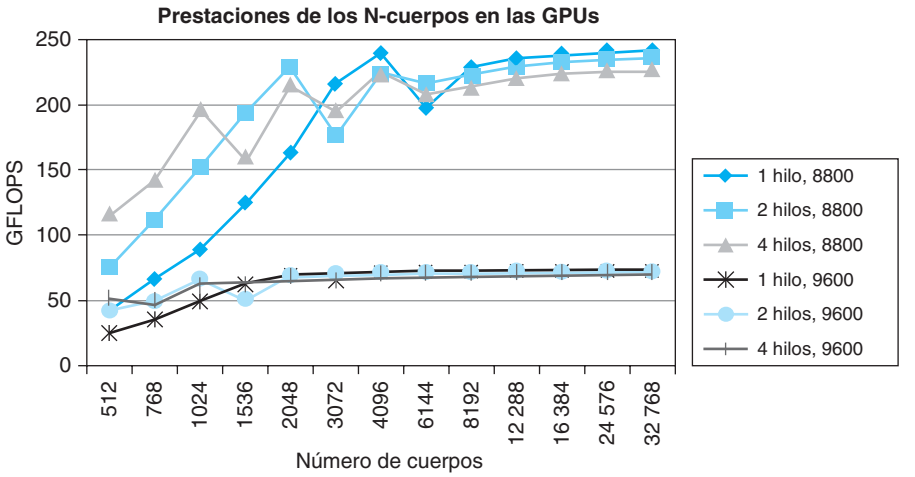


FIGURA A.8.15 Prestaciones de la aplicación de N-cuerpos en una GeForce 8800 GTX y en una GeForce 9600. La 8800 tiene 128 procesadores de *streaming* a 1.35 GHz, mientras que la 9600 tiene 64 a 0.80 GHz (aproximadamente un 30% más lento que en la 8800). Las prestaciones pico son 242 GFLOPS. Para una GPU con más procesadores el problema tiene que ser mayor para llegar a las máximas prestaciones (el pico en la 9600 está en 2048 cuerpos, mientras que en la 8800 no se alcanza este pico hasta los 16 384 cuerpos). Para valores de N pequeños, se pueden mejorar las prestaciones de manera significativa utilizando más de un hilo por cuerpo, pero posiblemente pueden producirse penalizaciones al aumentar N.

podemos almacenar los resultados parciales en la memoria compartida. De este modo, cuando se ha terminado el cálculo de las fuerzas, se recopilan los q resultados parciales y se suman para obtener el resultado final. Con dos o tres hilos por cuerpo se consiguen grandes mejoras en las prestaciones para N pequeño.

Por ejemplo, las prestaciones en la 8800 GTX para $N = 1024$ aumentan un 110% (un hilo llega a los 90 GFLOPS y cuatro a los 190 GFLOPS). Por contra, las prestaciones se degradan ligeramente para N grande, por lo tanto, esta optimización sólo debe utilizarse para $N < 4096$. En figura A.8.15 se muestra como aumentan las prestaciones en una GPU con 128 procesadores y en una GPU con 64 procesadores con un reloj más lento.

Comparación de las prestaciones

Las prestaciones del código de N-cuerpos se muestran en las figuras A.8.15 y A.8.16. Las prestaciones de GPUs de gama alta y media, junto con las mejoras obtenidas con la utilización de varios hilos por cuerpo, se muestran en la figura A.8.15. Las prestaciones en la GPU más rápida varían entre 90 y 250 GFLOPS.

La figura A.8.16 muestra los resultados obtenidos en la ejecución de un código casi idéntico (C++ frente a CUDA) en una CPU Core2 de Intel. Las prestaciones que se obtienen son un 1% de las obtenidas en la GPU, están entre 0.2 y 2 GFLOPS, y permanecen casi constantes en todo el rango de tamaño del problema.

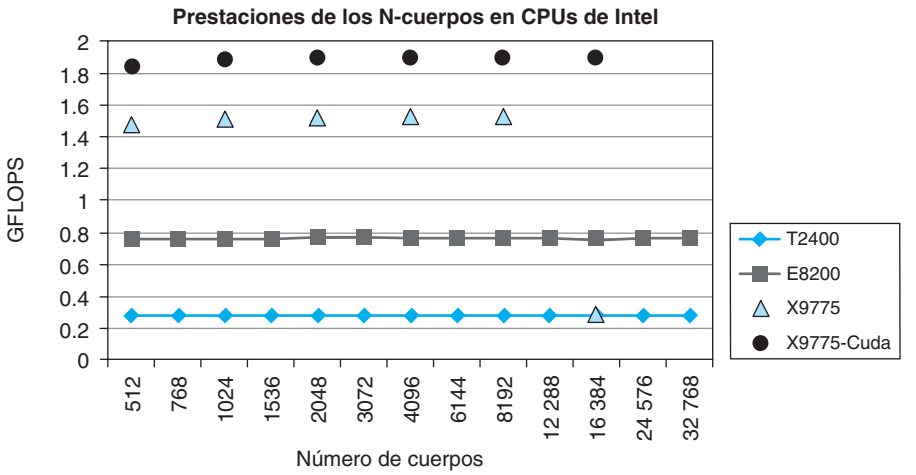


FIGURA A.8.16 Prestaciones de la aplicación de N-cuerpos en una CPU. La gráfica muestra las prestaciones del código de N-cuerpos en precisión simple en una CPU Core2 de Intel, etiquetada con su número de modelo. Observe la espectacular reducción en los GFLOPS (eje etiquetado como GFLOPS), demostrando que la GPU es mucho más rápida que la CPU. Las prestaciones de la CPU son independientes del tamaño del problema, excepto un comportamiento anómalo en la CPU X9775 para N = 16 384. Se muestran también los resultados alcanzados con la versión CUDA del código (utilizando el compilador CUDA-for-CPU), que mejoran en un 24% los resultados de C++. Como un lenguaje de programación, CUDA expone el paralelismo y la localidad para que sean explotados por el compilador. Las CPUs de Intel utilizadas son una Extreme X9775 a 3.2 GHz ("Penryn"), una E8200 a 2.6 GHz ("Wolfdale"), una CPU para computadores de sobremesa anterior al Penryn, y una T2400 a 1.83 GHz Yonah"), una CPU para computadores portátiles. La versión Penryn de la arquitectura Core2, con su divisor de 4 bits, es particularmente interesante para el problema de los N-cuerpos porque permite que las operaciones de división y raíz cuadrada se ejecuten cuatro veces más rápido que en CPUs anteriores.

La gráfica también muestra los resultados obtenidos compilando el código CUDA en una CPU; los resultados mejoran un 24%. Como un lenguaje de programación, CUDA expone el paralelismo y permite que el compilador aproveche mejor la unidad vectorial SSE en un único núcleo. La versión CUDA se adapta de forma natural a las CPUs multinúcleo (con mallas de bloques) y se obtiene un escalamiento casi perfecto en un sistema con ocho núcleos y N = 4096 (tasas de 2.0, 3.97 y 7.94 en dos, cuatro y ocho núcleos, respectivamente).

Resultados

Con un esfuerzo relativamente pequeño, hemos desarrollado un núcleo computacional que mejora las prestaciones en la GPU en un factor 157 respecto a las prestaciones en una CPU multinúcleo. El código de N-cuerpos ejecutándose en CPUs recientes de Intel (Penryn X9775 a 3.2 GH y un núcleo) necesitó más de tres segundos por cuadro que el mismo código que se ejecuta a una velocidad de 44 Hz por cuadro en la GPU GeForce 8800. En CPUs anteriores a la Penryn, se necesitaron entre 6 y 16 segundos, y en procesadores más antiguos Core2 y Pentium IV, el tiempo es de aproximadamente 25 segundos. Este incremento aparente en las prestaciones se debe dividir entre dos, porque la CPU necesita sólo la mitad de las operaciones para calcular el mismo resultado (utilizando las optimizaciones que suponen que las fuerzas en un par de cuerpos son iguales en magnitud y con direcciones opuestas).

¿Cómo es posible obtener esta aceleración? Para responder a esta pregunta tenemos que fijarnos en los detalles de la arquitectura. El cálculo de la fuerzas de pares de cuerpos tiene 20 operaciones punto flotante, básicamente sumas y multiplicaciones (algunas de las cuales se pueden combinar en instrucciones de multiplicación-suma), pero también hay normalizaciones de vectores, que necesitan divisiones y raíces cuadradas. En cálculo de las divisiones y raíces cuadradas en precisión simple en las CPUs de Intel tiene una latencia elevada,[3] aunque ésta se ha mejorado en la familia Penryn con la incoporación de un divisor de 4 bits.[4] Además, las limitaciones en la capacidad de los registros de las arquitectura x86 obliga a tener muchas instrucciones MOV (presumiblemente a/desde la cache L1). Por el contrario, la GeForce 8800 ejecuta las instrucción de inverso de la raíz cuadrada en cuatro ciclos (véase sección A.6), dispone de un mayor banco de registros (por hilo) y una memoria compartida en la que pueden estar los operandos de las instrucciones. Finalmente, el compilador de CUDA emite sólo 15 instrucciones en cada iteración del lazo, comparado con las más de 40 de los compiladores de las CPUs x86. Mayor paralelismo, ejecución más rápida de instrucciones complejas, más registros y un compilador eficiente; todos estos factores se combinan para explicar la gran mejora en las prestaciones del código de N-cuerpos en GPU respecto a la CPU.

[3] No consideramos las instrucciones x86 SSE de inverso de la raíz cuadrada (RSQRT*) e inverso (RCP*) porque su exactitud es muy baja.

[4] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Noviembre de 2007. Número: 248966-016. También disponible en www3.intel.com/design/processor/manuals/248966.pdf.

El algoritmo *all-pairs N-body* alcanza más de 240 GFLOPS en la GeForce 8800, comparado con los menos de 2GFLOPs de algunos procesadores secuenciales recientes. La compilación y ejecución del código CUDA en la CPU demuestra que el problema escala perfectamente en CPUs multinúcleo, pero que todavía es significativamente más lento que en una GPU.

Hemos acoplado la simulación de N-cuerpos en la GPU con una visualización gráfica del movimiento y hemos podido visualizar interactivamente 16K cuerpos interaccionando a 44 cuadros por segundo. Esto permite la visualización de sucesos de astrofísica y biofísica a tasas que permiten la interactividad. Además, hemos parametrizado muchas características, reducción de ruido, factor de amortiguamiento, técnicas de integración, visualizando inmediatamente sus efectos en la dinámica del sistema. Esto proporciona a los científicos excelentes sistemas de visualización que mejoran su comprensión de sistemas que serían invisibles para ellos (demasiado grandes o demasiado pequeños, demasiado rápidos o demasiado lentos) y de este modo pueden crear mejores modelos para el fenómeno físico.

La figura A.8.17 muestra una serie de visualizaciones de la simulación de un fenómeno astrofísico con 16K cuerpos, en el que cada cuerpos es una galaxia. La configuración inicial es un escudo esférico de cuerpos que rotan sobre el eje *z*. El agrupamiento que se produce, junto con la mezcla de galaxia con el tiempo, son fenómenos de interés en astrofísica. Para el lector interesado, el código CUDA para esta aplicación está disponible en el sistema de desarrollo (SDK) de CUDA, www.nvidia.com/CUDA.

Las GPUs evolucionaron y cambiaron tan rápidamente que han surgido muchas falacias y se han cometido muchos errores. En esta sección comentamos algunos.

Falacia: Las GPUs no son más que multiprocesadores vectoriales SIMD. Es fácil extraer la conclusión de que las GPUs son multiprocesadores vectoriales SIMD. El modelo de programación de las GPUs es de estilo SPMD, en el que un programador escribe un único programa que se ejecuta en varios hilos con diferentes datos. Sin embargo, la ejecución de estos hilos no es puramente SIMD o vectorial, es SIMT (*single-instruction, multiple-thread*), tal como se ha descrito en la sección A.4. Cada hilo de la GPU tiene sus propios registros escalares, memoria privada, estado de ejecución del hilo, identificación de hilo, ejecución y salida del salto independiente, contador de programa y puede direccionar la memoria independientemente. Aunque un grupo de hilos (por ejemplo, una trama de 32 hilos) se ejecuta más eficientemente cuando los contadores de programa (PC) son iguales para todos los hilos, también se ejecutan correctamente cuando son diferentes. Por lo tanto, los multiprocesadores no son estrictamente SIMD. El modelo de ejecución de hilos es MIMD con sincronización de barrera y optimizaciones SIMT. Por otra parte, la ejecución es más eficiente si los accesos de carga/almacenamiento en memoria de los hilos individuales se unen en accesos de bloques de hilos. Sin embargo, otra vez, esto no es estrictamente necesario. En una arquitectura vectorial SIMD pura, los accesos a registros/memoria desde diferentes hilos tendrían que alinearse según un patrón de acceso vectorial. En la GPU no existe esta restricción pero la ejecución es más eficiente cuando las tramas de hilos acceden a bloques de datos locales.

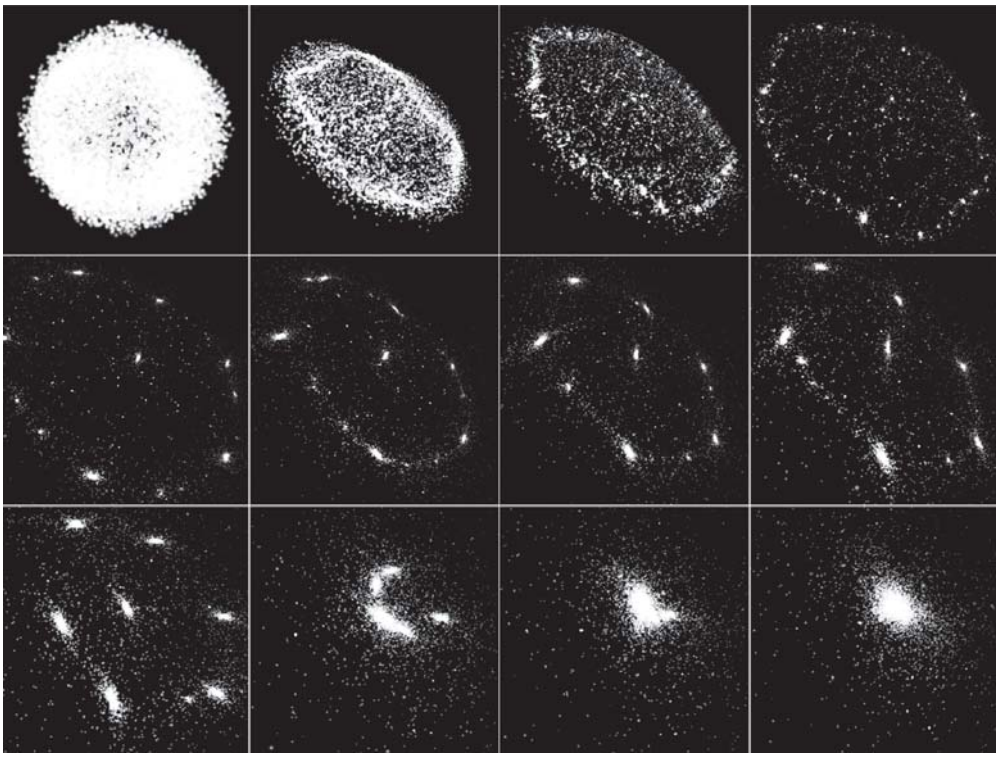


FIGURA A.8.17 Imágenes obtenidas durante la evolución de un sistema de N-cuerpos con 16 384 cuerpos.

A.9 Falacias y errores habituales

Otra diferencia respecto a un modelo puramente SIMD radica en que una GPU SIMT ejecuta varias trama de hilos concurrentemente. En aplicaciones gráficas, pueden estar corriendo al mismo tiempo varios grupos de programas de vértices, programas de píxeles y programas de geometría. En las aplicaciones de computación también se pueden ejecutar concurrentemente varios programas en tramas diferentes.

Falacia: Las prestaciones de la GPU no pueden crecer más rápidamente que la ley de Moore. La ley de Moore es simplemente un tasa de crecimiento, no es un límite insalvable para ninguna otra tasa. La ley de Moore describe las expectativas de que, con el paso de tiempo, y a medida que avanza la tecnología de semiconductores y los transistores se fabrican de menor tamaño, el coste de fabricación por transistor disminuirá exponencialmente. Dicho de otra forma,

dado un coste de fabricación constante, el número de transistores crecerá exponencialmente. Gordon Moore [1965] predijo que con esta progresión se duplicaría el número de transistores cada año manteniendo el mismo coste de fabricación; más tarde esta predicción se revisó a que el número de transistores se duplicaría cada dos años. Aunque Moore hizo la predicción inicial en 1965, cuando había sólo 50 componentes en un circuito integrado, se ha revelado como una predicción consistente. La reducción del tamaño del transistor tiene otros beneficios, por ejemplo, reducción del consumo de potencia por transistor y frecuencias de reloj más elevadas con el mismo consumo.

Los arquitectos de chips utilizan los transistores adicionales para construir procesadores, memorias y otros componentes. Durante un tiempo, los diseñadores de CPUs utilizaron los transistores extra para aumentar las prestaciones de los procesadores al ritmo marcado por la ley de Moore y, en consecuencia, mucha gente piensa que la ley de Moore predice una duplicación de las prestaciones de los procesadores cada 18 o 24 meses, cuando en realidad no es así.

Los diseñadores de microprocesadores emplean los transistores adicionales en nuevos núcleos, mejorando la arquitectura y el diseño, y en mejorar la segmentación para tener relojes más rápidos. El resto, se emplea en aumentar la capacidad de la cache y en un acceso más rápido a la memoria. Por el contrario, los diseñadores de GPUs no los utilizan para aumentar el tamaño de la cache; en su lugar, se utilizan para mejorar y añadir más núcleos.

Las GPUs son más rápidas debido a cuatro motivos. Primero, los diseñadores de GPU obtienen la recompensa de la ley de Moore empleando más transistores para introducir mayor paralelismo en los procesadores y, por lo tanto, disponer de procesadores más rápidos. Segundo, los diseñadores de GPUs pueden mejorar la arquitectura aumentando la eficiencia del procesamiento. Tercero, la ley de Moore asume un coste constante, por lo tanto, la tasa de la ley de Moore puede superarse gastando más en chips más grandes y con más transistores. Cuarto, se ha aumentado el ancho de banda del sistema de memoria de la GPU a un ritmo casi comparable al de procesamiento, incorporando memorias más rápidas y más anchas, compresión de datos y mejores caches. La combinación de estos cuatro aspectos ha permitido históricamente duplicar las prestaciones de la GPU cada 12 o 18 meses. Esta tasa, que excede la tasa de la ley de Moore, se ha verificado en aplicaciones gráficas a lo largo de los últimos 10 años y no muestra signos de disminuir. El desafío más limitante parece ser el sistema de memoria, pero están apareciendo innovaciones competitivas a un ritmo también muy elevado.

Falacia: Las GPUs sólo sirven para el rendering 3D, no son útiles en computación general. Las GPUs se diseñan para renderizar gráficos 3D, 2D y vídeo. Para atender las demandas de los desarrolladores de software gráfico, expresadas en las interfaces y en los requerimientos de prestaciones de las APIs gráficas, las GPUs se han convertido en procesadores punto flotante programables masivamente paralelos. En el dominio gráfico, esto procesadores se programan con APIs gráficas y con misteriosos lenguajes de programación gráfica (GLSL, Cg y HLSL, en OpenGL y Direct3D). Sin embargo, no hay nada que impida que los diseñadores

de GPUs expongan los núcleos paralelos a los programadores, sin las APIs gráficas o los misteriosos lenguajes de programación gráfica.

De hecho, la familia de GPUs con arquitectura Tesla expone los procesadores a través de un entorno software llamado CUDA, que permite que los programadores desarrollen programas de aplicación general en C y pronto en C++. Las GPUs son procesadores completos de Turing, por lo tanto pueden ejecutar cualquier programa que se ejecute en una CPU, aunque quizás no tan bien como en la CPU. O quizá mejor.

Falacia: Las GPUs no pueden ejecutar de un forma rápida programas punto flotante de precisión doble. En el pasado, las GPUs no podían ejecutar programas punto flotante de precisión doble, excepto por emulación software. Las GPUs han progresado desde una representación aritmética indexada (búsqueda en tablas para los colores) a enteros de 8 bits para cada componente del color, aritmética de punto fijo, punto flotante de precisión simple y recientemente se añadió la precisión doble. Las GPUs modernas pueden hacer prácticamente todas las operaciones en aritmética punto flotante IEEE de precisión simple y están empezando a utilizar, adicionalmente, precisión doble.

Con un pequeño coste adicional, una GPU soporta aritmética punto flotante de precisión doble y de precisión simple, aunque actualmente la precisión doble es unas cinco veces más lenta que la precisión simple. Por un coste adicional incremental, las prestaciones de la operaciones en precisión doble pueden ir aumentando paso a paso, a medida que las aplicaciones lo demanden.

Falacia: Las GPUs no hacen correctamente las operaciones en punto flotante. Las GPUs, al menos la familia de procesadores con arquitectura Tesla, hacen las operaciones punto flotante de precisión simple tal como se indica en el estándar del punto flotante IEEE 754. Así, en términos de exactitud, las GPUs son exactamente iguales a cualquier otro procesador que cumpla con el estándar IEEE 754.

Hoy en día, las GPUs no implementan algunas de las características del estándar, como por ejemplo el procesamiento de los número denormales y las excepciones precisas. Sin embargo, la GPU Tesla T10P, reciente introducida, proporciona todos los modos de redondeo del estándar, multiplicación-suma mezclada y procesamiento de números denormalizados para precisión doble.

Error: Utilizar más hilos para ocultar las latencias de memoria más largas. Los núcleos de la CPU están diseñados para ejecutar un único hilo al máximo de velocidad. Para ello, cuando le llega el turno de ejecución a una instrucción, ésta y sus datos deben estar ya disponibles; en caso contrario la instrucción no se puede ejecutar y el procesador se para. La memoria externa está alejada del procesador y la búsqueda de datos en memoria necesita muchos ciclos. Consecuentemente, las CPUs utilizan grandes caches locales para mantener las ejecuciones sin paradas. La latencia de la memoria es elevada y una forma de evitar paradas debido al acceso a los datos en memoria es esforzarse por tener estos datos en la cache. Sin embargo, en algún momento el conjunto de trabajo del programa puede ser demasiado grande para cualquier cache. Para tolerar estas altas latencias, algunas

CPUs hacen de uso de la ejecución multihilo, pero el número de hilos por núcleo está limitado a unos pocos hilos.

La estrategia de la GPUs es diferente. Los núcleos de la GPU están diseñados para ejecutar muchos hilos concurrentemente, pero en cada instante sólo se ejecuta una instrucción, que puede tomarse de cualquier hilo. Dicho de otra forma, en la GPU cada hilo se ejecuta lentamente, pero en conjunto los hilos se ejecutan eficientemente. Así, los hilos pueden tolerar las latencias de los accesos a memorias porque cuando un hilo se para, la GPU puede ejecutar otros hilos.

El lado negativo es que se necesitan muchos hilos para ocultar la latencia de memoria. Adicionalmente, si los accesos están dispersos o no correlacionados, el sistema de memoria se irá haciendo progresivamente más lento al responder las solicitudes individuales. Eventualmente, podría no ser posible ocultar la latencia de memoria incluso con varios hilos. Por lo tanto, el error es que para que la estrategia "utilizar más hilos" sea efectiva, hay que tener hilos suficientes y los hilos tienen que comportarse correctamente en cuanto a localidad de los accesos a memoria.

Falacia: Es difícil acelerar los algoritmos con complejidad O(*n*). No importa la velocidad a la que la GPU procesa los datos, la transferencia de datos a la memoria y desde ella pueden limitar las prestaciones de los algoritmos con complejidad O(n) (poco trabajo por dato). El ritmo de transferencia máximo en un bus PCIe, cuando se utilizan DMAs, es aproximadamente 48 GB/segundo. Por el contrario, la velocidad típica de acceso al sistema de memoria de la CPU es 8–12 GB/segundo. Algunos ejemplos, como la suma de vectores, estarán limitados por la transferencia de las entradas a la GPU y el volcado de los resultados.

Hay tres formas para superar el coste de la transferencia de datos. Primero, intentar dejar los datos en la GPU todo el tiempo que sea posible, en lugar de moverlos adelante y atrás en diferentes pasos de un algoritmo complicado. Para ello, CUDA deja los datos en la GPU, de forma deliberada, entre lanzamientos.

Segundo, la GPU tiene soporte para la ejecución concurrente de operaciones copiar-desde, copiar-en y de cálculo, de forma que se puede mantener un flujo constante de datos desde y hacia la memoria mientras se están haciendo cálculos. Este modelo es útil para cualquier flujo de datos que puede procesarse a medida que que va llegando. Algunos ejemplos son el procesamiento de vídeo, encaminamiento en redes, compresión/descompresión de datos e incluso algunas computaciones simples como la manipulación matemática de vectores con muchos elementos.

Las tercera estrategia es utilizar la CPU y la GPU de forma conjunta, como una plataforma de computación heterogénea, para mejorar la prestaciones, asignando un subconjunto del trabajo a realizar a cada una. El modelo de programación de CUDA permite la asignación de trabajo a una o varias GPUs junto con un uso continuado de la CPU sin usar hilos (a través de funciones de la GPU asíncronas), por lo tanto, es relativamente sencillo mantener a las GPUs y la CPU trabajando concurrentemente para resolver problemas aún más rápido.

A.10 Conclusiones finales

Las GPUs son procesadores masivamente paralelos y cada vez tienen mayor aceptación, no sólo para gráficos 3D, sino también en muchas otras aplicaciones. Este campo de utilización tan amplio es posible gracias a la transformación de los dispositivos gráficos en procesadores programables. El modelo de programación de aplicaciones gráficas en la GPU es usualmente una API, por ejemplo DirectX™ y OpenGL™. Para computación de propósito general, el modelo de programación CUDA utiliza un estilo SPMD (*Single-Program, Multiple-Data*), ejecutando el programa con muchos hilos paralelos.

El paralelismo de la GPU continuará escalando con la ley de Moore, principalmente incrementando el número de procesadores. Sólo los modelos de programación paralela, que escalen realmente a cientos o miles de núcleos, serán capaces de aprovechar las GPUs y CPUs con muchos núcleos. Por otra parte, las arquitecturas masivamente paralelas con muchos núcleos sólo podrán acelerar las aplicaciones con muchas tareas paralelas altamente independientes.

Los modelos de programación de las GPUs son cada vez más flexibles, tanto para gráficos como para computación paralela. Por ejemplo, CUDA está evolucionando rápidamente en la dirección de incorporar la funcionalidad completa de C/C++. Las APIs gráficas y los modelos de programación probablemente adaptarán las capacidades paralelas y los modelos de CUDA. El modelo de ejecución multihilo SPMD es un modelo escalable, conveniente, conciso y fácil de aprender para la expresión del paralelismo a gran escala.

Al hilo de estos cambios en los modelos de programación, las GPUs se están haciendo más flexibles y más programables. Así, los programas de propósito general pueden acceder a las unidades de función fija de la GPU y los programas CUDA ya utilizan las funciones intrínsecas de texturas para hacer búsquedas de texturas con las instrucciones y la unidad de texturas de la GPU.

La arquitectura de la GPU seguirá adaptándose a los patrones de los programadores de gráficos y otras aplicaciones. Seguirá expandiéndose, implementando núcleos adicionales y poniendo a disposición de los programas mayor ancho de banda de hilos y memoria, para tener mayor capacidad computacional. Además, los modelos de programación tienen que evolucionar para incorporar la programación de sistemas heterogéneos con muchos núcleos, incluyendo GPUs y CPUs.

Agradecimientos

Este apéndice es el resultado del trabajo de varios autores de NVIDIA. Agradecemos sinceramente las importantes contribuciones de Michael Garland, John Montrym, Dough Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman y Vasily Volkov.

Perspectiva histórica y lecturas recomendadas

Esta sección, disponible en el CD, revisa la historia de las unidades de procesamiento gráfico de tiempo real programables (GPUs) desde comienzos de los 80 hasta la actualidad, el descenso del precio en dos órdenes de magnitud y el aumento de las prestaciones en dos órdenes de magnitud. Sigue la evolución de la GPU desde los pipelines de función fija hasta los procesadores gráficos programables, con referencias a la computación en la GPU, procesadores unificados de gráficos y computación, computación visual y GPUs escalables.

B

APÉNDICE

Ensambladores, enlazadores y el simulador SPIM

El temor a graves daños no puede, por sí solo, justificar la supresión de la libertad de expresión y de reunión.

Louis Brandeis
Whitney v. California, 1927

James R. Larus
Microsoft Research
Microsoft

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

B.1 Introducción

La codificación de instrucciones como números binarios es natural y eficiente para los computadores. Los seres humanos, sin embargo, tienen gran dificultad para comprender y manipular estos números. Las personas leen y escriben símbolos (palabras) con más facilidad que secuencias largas de dígitos. El capítulo 2 mostraba que no es necesario elegir entre números o palabras, porque las instrucciones del computador se pueden representar de muchas maneras. Los seres humanos pueden leer y escribir símbolos, y los computadores pueden ejecutar los números binarios equivalentes. Este apéndice describe el proceso por el cual un programa legible para humanos se traduce a un formato que un computador puede ejecutar, ofrece algunos consejos para escribir programas en lenguaje ensamblador y explica cómo ejecutar estos programas en SPIM, un simulador que ejecuta programas MIPS. Las versiones del simulador SPIM para UNIX, Windows y Mac OS X están disponibles en el CD.

El *lenguaje ensamblador* es la representación simbólica de la codificación binaria de un computador: el **lenguaje máquina**. El lenguaje ensamblador es más fácil de leer que el lenguaje máquina ya que usa símbolos en lugar de bits. Los nombres de los símbolos en lenguaje ensamblador producen normalmente patrones de bits, como los códigos de operación (opcodes) e identificadores de registro, de esta manera las personas pueden leerlos y recordarlos. Además, el lenguaje

Lenguaje máquina: representación binaria utilizada para la comunicación dentro de un computador.

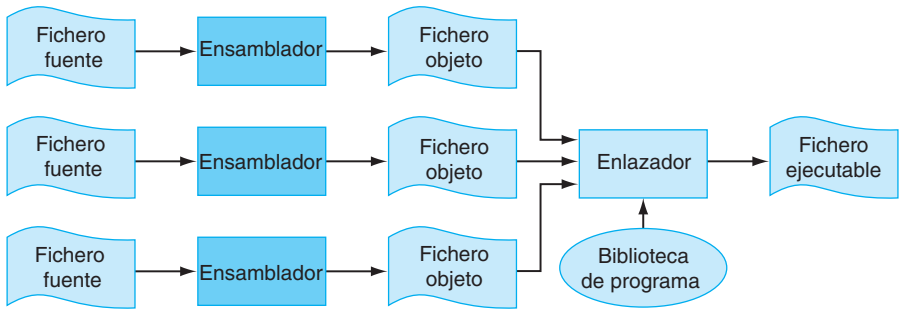


FIGURA B.1.1 El proceso que produce un fichero ejecutable. Un ensamblador traduce un fichero en lenguaje ensamblador a un fichero objeto, que se vincula (enlaza/monta) con otros ficheros y bibliotecas en un fichero ejecutable.

ensamblador permite a los programadores usar *etiquetas* para identificar y dar nombre a una palabra de memoria concreta que almacena instrucciones o datos.

Una herramienta llamada **ensamblador** traduce lenguaje ensamblador en instrucciones binarias. Los lenguajes ensambladores proporcionan una representación más amigable al programador que los 0s y 1s de un computador, lo que simplifica la lectura y escritura de los programas. Los nombres simbólicos de las operaciones y posiciones de memoria son una faceta de esta representación. Otra faceta son los recursos de programación que incrementan la claridad de un programa. Por ejemplo, las **macros**, que se presentan en la sección B.2, permiten a los programadores ampliar el lenguaje ensamblador definiendo nuevas operaciones.

Ensamblador: programa que traduce una versión simbólica de las instrucciones a la versión binaria.

Macros: herramienta de agrupación de patrón y reemplazo que ofrece un mecanismo sencillo para dar nombre a una secuencia de instrucciones de uso frecuente.

Referencias no resueltas: referencias que requieren más información de una fuente externa con el fin de completarse.

Enlazador también llamado ***editor de enlaces***: programa del sistema que combina programas en lenguaje máquina ensamblados independientemente y resuelve todas las etiquetas no definidas en un fichero ejecutable.

Un ensamblador lee un único *fichero fuente* escrito en lenguaje ensamblador y genera un *fichero objeto* que contiene las instrucciones en lenguaje máquina y la información contable que ayuda a combinar varios ficheros objeto en un programa ejecutable. La figura B.1.1 ilustra cómo se construye un programa ejecutable. La mayoría de los programas constan de varios ficheros —también llamados *módulos*— que están escritos, compilados y ensamblados independientemente. Un programa también puede usar rutinas preescritas suministradas en una *biblioteca de programa*. Un módulo contiene *referencias* a subrutinas y datos definidos en otros módulos y en bibliotecas. El código de un módulo no se puede ejecutar cuando contiene **referencias sin resolver** a etiquetas en otros ficheros objeto o bibliotecas. Otra herramienta, llamada **enlazador** (***linker***), combina una colección de objeto y ficheros de la biblioteca en un *fichero ejecutable*, que un computador puede ejecutar.

Para ver la ventaja del lenguaje ensamblador, se puede considerar la siguiente secuencia de figuras, todas contienen una subrutina corta que calcula e imprime la suma de los cuadrados de los números enteros de 0 a 100. La figura B.1.2 muestra el código en lenguaje máquina que ejecuta un computador MIPS. Con un considerable esfuerzo, podría utilizar las tablas con el código de operación (opcode) y el formato de las instrucciones del capítulo 2 para traducir las instrucciones en un programa simbólico similar al que muestra la figura B.1.3. Esta forma de la rutina es mucho más fácil de leer porque las operaciones y los ope-

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000011110000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

FIGURA B.1.2 Código de una rutina para calcular e imprimir la suma de los cuadrados de los números enteros entre 0 y 100 en lenguaje máquina MIPS.

randos están escritos con símbolos, en lugar de con patrones de bits. Sin embargo, este lenguaje ensamblador es aún difícil de seguir porque las posiciones de memoria se referencian por su dirección, en lugar de por una etiqueta simbólica.

La figura B.1.4 muestra el código de la rutina en lenguaje ensamblador en el que se etiquetan direcciones de memoria con nombres mnemotécnicos. La mayoría de los programadores prefieren leer y escribir en esta forma. Los nombres que comienzan con un punto, como por ejemplo `.data` y `.globl`, son **directivas del ensamblador** que informan al ensamblador cómo traducir un programa, pero no producen instrucciones en lenguaje máquina. Los nombres seguidos de dos puntos, como `str:` o `main:`, son etiquetas que dan nombre a la siguiente posición de memoria. Este programa es tan legible como la mayoría de los programas escritos en lenguaje ensamblador (excepto por una evidente falta de comentarios), pero, aún así, es difícil de seguir, porque son necesarias muchas operaciones sencillas para realizar tareas simples y porque el lenguaje ensamblador carece de modelos de control de flujo lo que proporciona muy pocas pistas sobre el funcionamiento del programa.

Directivas del ensamblador: operación que le indica al ensamblador cómo traducir un programa, pero no producen instrucciones máquina; siempre se inicia con un punto.

Por el contrario, la rutina en lenguaje C en la figura B.1.5 es a la vez corta y clara, ya que las variables tienen nombres mnemotécnicos y el lazo es explícito en lugar de estar construido con saltos. De hecho, la rutina en lenguaje C es la única que escribimos nosotros. Los otros códigos del programa fueron producidos por un compilador de C y un ensamblador.

En general, el lenguaje ensamblador juega dos papeles (véase figura B.1.6). El primero es el lenguaje de salida de los compiladores. Un compilador traduce un programa escrito en un lenguaje de alto nivel (como Pascal o C) a un pro-

grama equivalente escrito en lenguaje ensamblador o máquina. El lenguaje de alto nivel se llama **lenguaje fuente**, y el lenguaje de salida del compilador es el *lenguaje destino*.

Lenguaje fuente: lenguaje de alto nivel en el que se escribe originalmente un programa.

```
addiu       $29, $29, -32
sw          $31, 20($29)
sw          $4,  32($29)
sw          $5,  36($29)
sw          $0,  24($29)
sw          $0,  28($29)
lw          $14, 28($29)
lw          $24, 24($29)
multu       $14, $14
addiu       $8,  $14, 1
slti        $1,  $8, 101
sw          $8,  28($29)
mflo        $15
addu        $25, $24, $15
bne         $1,  $0, -9
sw          $25, 24($29)
lui         $4,  4096
lw          $5,  24($29)
jal         1048812
addiu       $4,  $4, 1072
lw          $31, 20($29)
addiu       $29, $29, 32
jr          $31
move        $2,  $0
```

FIGURA B.1.3 La misma rutina de la figura B.1.2 escrita en lenguaje ensamblador. Sin embargo, el código de la rutina no etiqueta registros o posiciones de memoria ni incluye comentarios.

El otro papel del lenguaje ensamblador es como un lenguaje para escribir programas. Este papel era el dominante. Actualmente, sin embargo, a causa de la mayor capacidad de memoria principal y la disponibilidad de mejores compiladores, la mayoría de los programadores escriben en un lenguaje de alto nivel y pocas veces, o nunca, consultan las instrucciones que ejecuta el computador. Sin embargo, el lenguaje ensamblador continúa siendo importante para escribir programas en los que la velocidad o el tamaño son críticos o para aprovechar las características hardware que no tienen análogos en lenguajes de alto nivel.

A pesar de que este apéndice se centra en el lenguaje ensamblador MIPS, la programación en ensamblador en la mayoría de los sistemas es muy similar. Las instrucciones y modos de direccionamiento adicionales que poseen máquinas CISC, como el VAX, pueden reducir el tamaño de los programas escritos en ensamblador, pero no cambian el proceso de ensamblado de un programa o proporcionan al lenguaje ensamblador de las ventajas de los lenguajes de alto nivel como son la comprobación de tipos o el control de flujo estructurado.

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "La  suma desde 0 .. 100 es %d\n"
```

FIGURA B.1.4 La misma rutina escrita en lenguaje ensamblador con etiquetas, pero sin comentarios. Los comandos que empiezan con un punto son las directivas de ensamblador (véanse las páginas B-47-B-49). `.text` indica que las líneas siguientes contienen instrucciones. `.data` indica que contienen los datos. `.align n` indica que los campos en las líneas siguientes deben estar alineados en una dirección múltiplo de 2^n bytes. Por lo tanto, `.align 2` significa que el siguiente elemento debe estar en una dirección múltiplo de 22, que es una dirección límite de una palabra. `.globl main` declara que `main` es un símbolo global que debe ser visible para el código almacenado en otros ficheros. Por último, `.asciiz` almacena una cadena de caracteres, acabado con un carácter de terminación o nulo en la memoria.

Cuándo utilizar lenguaje ensamblador

La razón principal para programar en lenguaje ensamblador, y no en un lenguaje disponible de alto nivel, es que la velocidad o el tamaño de un programa sean de importancia crítica. Por ejemplo, considere un computador que controla una pieza mecánica, como los frenos de un coche. Un computador que se incorpora en otro dispositivo, como puede ser un coche, se denomina *computador empotrado*. Este tipo de computadores necesita responder de forma rápida y predecible a los acontecimientos en el mundo exterior. Debido a que un compilador introduce incertidumbre sobre el coste en tiempo de las operaciones, los programadores pueden encontrar dificultades para asegurar que un programa en lenguaje de alto nivel responde dentro de un intervalo de tiempo finito prefijado, por ejemplo, 1 milisegundos después de que un sensor detecte que un neumático está derrapando. En cambio, un programador de lenguaje ensamblador tiene un estricto control de la ejecución de instrucciones. Además, en aplicaciones empotradas, reducir el

```
   #include <stdio.h>
int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("La suma desde 0 .. 100 es %d\n", sum);
}
```

FIGURA B.1.5 La rutina escrita en el lenguaje de programación C.

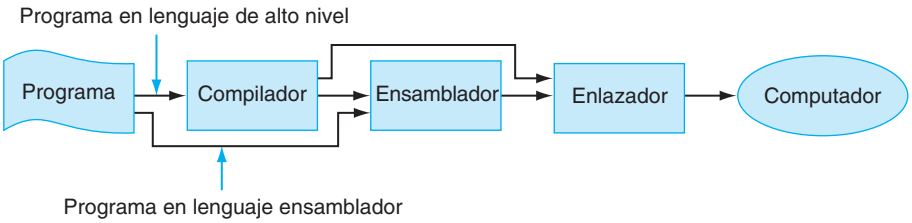


FIGURA B.1.6 El programa en lenguaje ensamblador puede ser escrito por un programador o es la salida de un compilador.

tamaño de un programa, para que quepa en un menor número de chips de memoria, reduce el coste del computador empotrado.

Un enfoque híbrido, en el que la mayor parte de un programa se escribe en un lenguaje de alto nivel y las secciones críticas en cuanto a tiempo se escriben en lenguaje ensamblador, se basa en los puntos fuertes de ambos lenguajes. Los programas normalmente pasan la mayor parte de su tiempo de ejecución en una pequeña fracción del código fuente del programa. Esta observación es precisamente el principio de localidad que subyace en la caches (véase sección 5.1 en el capítulo 5).

Los programas que analizan perfiles miden donde gasta su tiempo los programas y permiten encontrar sus partes críticas. En muchos casos, esta parte del programa puede acelerarse con mejores estructuras de datos o algoritmos. A veces, sin embargo, las mejoras más significativas en las prestaciones sólo se consiguen recodificando en lenguaje ensamblador la parte crítica de un programa.

Esta mejora no es necesariamente una indicación de que el compilador del lenguaje de alto nivel ha fracasado. Los compiladores normalmente son mejores que los programadores en la producción de código máquina de alta calidad de manera uniforme en un programa completo. Los programadores, sin embargo, comprenden el comportamiento y algoritmo de un programa con un nivel más profundo que un compilador y pueden invertir más esfuerzo e ingenio en mejorar pequeñas secciones del programa. En particular, los programadores a menudo consideran

simultáneamente varios procedimientos mientras escriben su código. Los compiladores normalmente traducen los procedimientos de forma aislada y deben seguir estrictamente los convenios que rigen el uso de los registros en el ámbito del procedimiento. Mediante la conservación en los registros de los valores utilizados frecuentemente, incluso a través de distintos procedimientos, los programadores pueden conseguir que un programa se ejecute más rápido.

Otra importante ventaja de la programación en lenguaje ensamblador es la capacidad de aprovechar las instrucciones especializadas, por ejemplo instrucciones de copiar cadenas o comparación de patrones coincidentes. En la mayoría de los casos los compiladores no pueden determinar que un bucle de un programa pueda ser reemplazado por una única instrucción. Sin embargo, el programador que escribió el bucle puede reemplazarlo fácilmente por una única instrucción.

Actualmente, la ventaja de un programador sobre un compilador se ha vuelto difícil de mantener ya que las técnicas de compilación mejoran y aumenta la complejidad de los procesadores segmentados (capítulo 4).

Por último, una razón para utilizar el lenguaje ensamblador es que ningún lenguaje de alto nivel está disponible para un computador particular. Muchos computadores antiguos o especializados no tienen un compilador de lenguaje de alto nivel, de modo que la única alternativa de un programador es el lenguaje ensamblador.

Inconvenientes del lenguaje ensamblador

El lenguaje ensamblador tiene muchas desventajas que sirven de fuerte argumento en contra de su uso generalizado. Quizás su principal desventaja es que los programas escritos en lenguaje ensamblador son inherentemente específicos de una máquina y deben ser totalmente reescrito para funcionar en un computador con otra arquitectura. La rápida evolución de los computadores tratamos en el capítulo 1 significa que las arquitecturas van quedando obsoletas. Un programa en lenguaje ensamblador permanece estrechamente vinculado a su arquitectura original, incluso después de que el computador es eclipsado por máquinas nuevas, más rápidas y más rentables.

Otra desventaja es que los programas en lenguaje ensamblador son más largos que el programa equivalente escrito en un lenguaje de alto nivel. Por ejemplo, el programa en C en la figura B.1.5 tiene 11 líneas, mientras que el programa ensamblador de la figura B.1.4 es de 31 líneas. En programas más complejos, la relación entre el tamaño del programa en lenguaje ensamblador y el tamaño del programa en lenguaje de alto nivel (*factor de expansión*) puede ser mucho mayor que el factor tres de en este ejemplo. Lamentablemente, estudios empíricos demuestran que los programadores escriben más o menos el mismo número de líneas de código por día tanto en lenguaje ensamblador como en lenguaje de alto nivel. Esto implica que los programadores son unas x veces más productivos si trabajan en un lenguaje de alto nivel, donde x es el factor de expansión del lenguaje ensamblador.

Para complicar aún más el problema, los programas más largos son más difíciles de leer y entender y contienen más errores. El lenguaje ensamblador agrava el problema debido a su completa falta de estructura. Construcciones de programación comunes, tal como sentencias "si-entonces" (*if-then*) o los lazos, se deben

realizar mediante sentencias de salto condicional o de saltos incondicionales. Los programas resultantes son difíciles de leer porque el lector debe recomponer cada construcción de alto nivel a partir de sus partes y además cada instancia de una sentencia puede ser ligeramente diferente. Por ejemplo, observe la figura B.1.4 y conteste a estas preguntas: ¿Qué tipo de bucle se utiliza? ¿Cuáles son sus límites inferior y superior?

Extensión: Los compiladores pueden producir lenguaje máquina directamente, en lugar de depender de un ensamblador. Estos compiladores suelen ejecutar mucho más rápido que los que utilizan un ensamblador como parte de la compilación. Sin embargo, un compilador que genera lenguaje máquina debe llevar a cabo muchas tareas que normalmente realiza un ensamblador, tales como la resolución de direcciones y la codificación de las instrucciones como números binarios. La elección está entre la velocidad de compilación y la simplicidad del compilador.

Extensión: A pesar de estas consideraciones, algunas aplicaciones para sistemas empotrados están escritas en un lenguaje de alto nivel. Muchas de estas aplicaciones son programas grandes y complejos que deben ser extremadamente fiables. Los programas escritos en lenguaje ensamblador son más largos y más difíciles de leer y escribir que los programas escritos en lenguajes de alto nivel. Esto aumenta enormemente el coste de la escritura de un programa en lenguaje ensamblador y hace que sea extremadamente difícil verificar el funcionamiento correcto de este tipo de programa. De hecho, estas consideraciones llevaron el Departamento de Defensa, que paga por numerosos y complejos sistemas empotrados, a desarrollar ADA, un lenguaje de alto nivel indicado para escribir sistemas empotrados.

B.2 Ensambladores

Un ensamblador traduce un fichero de sentencias del lenguaje ensamblador a un fichero que contiene instrucciones del lenguaje máquina en binario y datos binarios. El proceso de traducción tiene dos pasos principales. El primer paso consiste en asociar las posiciones de memoria con etiquetas, así la relación entre el nombre simbólico y la dirección se conoce cuando se traducen las instrucciones. El segundo paso consiste en traducir cada sentencia del lenguaje ensamblador combinando los equivalentes numéricos de los códigos de operación (*opcodes*), los registros especificados, y las etiquetas permitidas en una instrucción. Como se muestra en la figura B.1.1, el ensamblador genera un fichero de salida, llamado *fichero objeto*, que contiene las instrucciones máquina, datos e información de contadores que necesita el enlazador.

Etiqueta externa (también llamada **etiqueta global**): etiqueta que hace referencia a un objeto que se puede referenciar desde otros ficheros distintos de aquel en el que se define.

Un fichero objeto normalmente no se puede ejecutar debido a que referencia a procedimientos o datos en otros ficheros. Una **etiqueta** es **externa** (también llamada **global**) si el objeto etiquetado puede ser referenciado desde otros ficheros

distinto de aquel en el que se define. Una **etiqueta** es **local** si el objeto se puede utilizar únicamente dentro del fichero en el que se define. En la mayoría de los lenguajes ensambladores, las etiquetas se consideran por defecto locales y deben ser declaradas explícitamente como etiquetas globales. Subrutinas y variables globales requieren etiquetas externas, ya que están referenciados desde varios de los ficheros de un programa. Las etiquetas locales ocultan nombres que no deben ser visibles para los otros módulos, por ejemplo, funciones estáticas del lenguaje C, que sólo puede ser llamadas por otras funciones en el mismo fichero. Además, los nombres generados por el compilador —por ejemplo, un nombre para la instrucción en el inicio de un bucle local— son locales por lo que el compilador no necesita generar nombres exclusivos para cada fichero.

Etiqueta local: etiqueta que hace referencia a un objeto que se puede utilizar únicamente en el fichero en que se define

Etiquetas locales y globales

EJEMPLO

Considere el programa que se muestra en la figura B.1.4, en la página B-7. La subrutina tiene una etiqueta externa (global) `main`. También contiene dos etiquetas locales —`loop` y `str`— que sólo son visibles en este fichero de lenguaje ensamblador. Por último, la rutina también contiene una referencia sin resolver a una etiqueta externa `printf`, que es la rutina de la biblioteca que imprime los valores. ¿Qué etiquetas de la figura B.1.4 pueden ser referenciada desde otro fichero?

RESPUESTA

Sólo las etiquetas globales son visibles desde fuera de un fichero, de modo que la única etiqueta que podría ser referenciada desde otro fichero es `main`.

Debido a que el ensamblador procesa cada uno de los ficheros de un programa de forma individual y aislada, sólo conoce las direcciones de las etiquetas locales. El ensamblador depende de otra herramienta, el enlazador, para combinar un conjunto de ficheros objeto y bibliotecas en un fichero ejecutable resolviendo las referencia a etiquetas externas. El ensamblador ayuda al enlazador proporcionando listas de etiquetas y de referencias sin resolver.

Sin embargo, incluso las etiquetas locales presentan un desafío interesante para un ensamblador. A diferencia de los nombres que se dan en la mayoría de los lenguajes de alto nivel, en lenguaje ensamblador las etiquetas pueden ser usadas antes de ser definidas. En el ejemplo de la figura B.1.4, la etiqueta `str` es utilizada por la instrucción `la` antes de que esté definida. La posibilidad de una **referencia hacia delante** (*forward reference*), como la mencionada, fuerza al ensamblador a traducir un programa en dos fases: en primer lugar buscar todas las etiquetas y luego generar las instrucciones. En el ejemplo, cuando el ensamblador ve la instrucción `la`, no sabe donde se encuentra la palabra etiquetada `str` o ni tan siquiera si `str` es la etiqueta de una instrucción o de un dato.

Referencia hacia delante: etiqueta que es utilizada antes de que sea definida.

En el primer paso de un ensamblador se lee cada línea de un fichero ensamblador y lo divide en las piezas que lo componen. Estas piezas, denominadas *lexemas*, son palabras individuales, números, caracteres de puntuación. Por ejemplo, la línea

```
ble $t0, 100, loop
```

contiene seis lexemas: el código de operación (*opcode)* `ble`, el identificador de registro `$t0`, una coma, el número `100`, una coma y el símbolo `loop`.

Tabla de símbolos: tabla que empareja los nombres de las etiquetas con las direcciones de las palabras de memoria que ocupan las instrucciones.

Si una línea empieza por una etiqueta, el ensamblador graba en su **tabla de símbolos** el nombre de la etiqueta y la dirección de la palabra de memoria que ocupa la instrucción. El ensamblador calcula el número de palabras de memoria que ocupa la instrucción en la línea actual. Llevando la cuenta del tamaño de las instrucciones, el ensamblador puede determinar dónde se almacenará la siguiente instrucción. Para calcular el tamaño de una instrucción de longitud variable, como las del VAX, el ensamblador tiene que examinarla con detalle. Instrucciones de longitud fija, como las instrucciones de los MIPS, en cambio, sólo requieren un examen superficial. El ensamblador realiza un cálculo similar para determinar el espacio que requieren las declaraciones de los datos. Cuando el ensamblador llega al final del fichero ensamblador, la tabla de símbolos contiene la posición de cada etiqueta definida en el fichero.

El ensamblador utiliza la información contenida en la tabla de símbolos para realizar el segundo paso sobre el fichero, que es el que en realidad produce el código máquina. El ensamblador analiza de nuevo cada una de las líneas en el fichero. Si la línea contiene una instrucción, el ensamblador combina las representaciones binarias de su código de operación (*opcode*) y de los operandos (identificadores de registros o dirección de memoria) para crear una instrucción válida. El proceso es similar al utilizado en la sección 2.5 en el capítulo 2. Las instrucciones y datos que hacen referencia a un símbolo externo definido en otro fichero no pueden ser completamente ensamblados (están sin resolver) puesto que la dirección del símbolo no se encuentra en la tabla de símbolos. El ensamblador no se queja por referencias sin resolver, puesto que la etiqueta correspondiente es probable que se defina en otro fichero.

IDEA clave

El lenguaje ensamblador es un lenguaje de programación. Su principal diferencia con los lenguajes de alto nivel como BASIC, Java y C es que el lenguaje ensamblador sólo proporciona unos pocos tipos de datos y un control de flujo sencillo. Los programas escritos en lenguaje ensamblador no especifican el tipo de valor almacenado en una variable. En cambio, el programador debe aplicar las operaciones adecuadas al valor (por ejemplo, suma de enteros o punto flotante). Además, en lenguaje ensamblador, los programas deben implementar todo el flujo de control con saltos *go to*. Ambos factores hacen que la programación en lenguaje ensamblador para cualquier máquina —MIPS u 80x86— sea más difícil y propensa a errores que escribiendo los programas en un lenguaje de alto nivel.

Extensión: Si la velocidad de un ensamblador es importante, este proceso de dos pasos se puede hacer en uno solo sobre el fichero ensamblador con una técnica conocida como ***backpatching***. En su paso por el fichero, el ensamblador construye una representación binaria (posiblemente incompleta) de cada instrucción. Si la instrucción hace referencia a una etiqueta que aún no ha sido definida, el ensamblador almacena la etiqueta y la instrucción en una tabla. Cuando se define una etiqueta, el ensamblador consulta esta tabla para encontrar todas las instrucciones que contienen referencia hacia delante de la etiqueta. El ensamblador vuelve atrás y corrige su representación binaria para incorporar la dirección correcta de la etiqueta. *Backpatching* acelera el proceso de ensamblado debido a que el ensamblador lee su entrada sólo una vez. Sin embargo, requiere que el ensamblador almacene en memoria toda la representación binaria de un programa, de modo que se pueda realizar el *backpatched* de las instrucciones. Este requisito puede limitar el tamaño de los programas que pueden ser ensamblados. El proceso es complicado para las máquinas con varios tipos de saltos condicionales que abarcan distintos rangos de instrucciones. Cuando el ensamblador ve primero una etiqueta sin resolver en una instrucción de salto condicional, debe usar el salto más largo posible o corre el riesgo de tener que volver atrás y reajustar muchas instrucciones para hacer espacio para un salto más largo.

Backpatching: método para traducir de lenguaje ensamblador a instrucciones máquina en el que el ensamblador construye una representación binaria de cada instrucción (posiblemente incompleta), en un paso sobre un programa y, a continuación, vuelve a llenar las etiquetas previamente no definidas.

Formato del fichero objeto

Los ensambladores producen ficheros objeto. Un fichero objeto en UNIX contiene seis secciones distintas (véase figura B.2.1):

- La cabecera del *fichero objeto* describe el tamaño y la posición del resto de las secciones del fichero.
- El **segmento de texto** contiene el código en lenguaje máquina de las rutinas del fichero fuente. Estas rutinas pueden ser no ejecutables, por contener referencias sin resolver.
- El **segmento de datos** contiene la representación binaria de los datos del fichero fuente. Los datos también pueden ser incompletos debido a las referencias no resueltas a las etiquetas en otros ficheros.
- La **información de reubicación** identifica las instrucciones y los datos que dependen de **direcciones absolutas**. Estas referencias deben cambiar si partes del programa se mueven en la memoria.
- La *tabla de símbolos* asocia direcciones con etiquetas externas del fichero fuente y también lista las referencias sin resolver.
- La *información para depuración* contiene una descripción concisa de la forma en que el programa fue compilado, de modo que el depurador puede encontrar qué direcciones de instrucciones corresponden a las líneas en un fichero fuente e imprimir las estructuras de datos en forma comprensible.

El ensamblador genera un fichero objeto que contiene la representación binaria del programa y los datos y la información adicional que ayudará a enlazar las partes de un programa. Esta información de reubicación es necesaria porque el ensamblador no conoce las posiciones de memoria que ocupará cada procedi-

Segmento de texto: segmento de un fichero objeto UNIX que contiene el código en lenguaje máquina para rutinas en el fichero fuente.

Segmento de datos: segmento de un fichero UNIX objeto o ejecutable que contiene una representación binaria de los datos iniciados utilizados por el programa.

Información de reubicación: segmento de un fichero objeto de UNIX que identifica las instrucciones y palabras de datos que dependen de direcciones absolutas.

Dirección absoluta: dirección real de una rutina o de una variable en memoria.

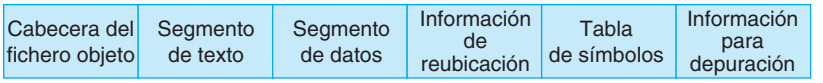


FIGURA B.2.1 Fichero Objeto. Un ensamblador UNIX produce un fichero objeto con seis secciones distintas.

miento o parte de los datos antes de que se enlace con el resto del programa. Los procedimientos y datos de un fichero se almacenan en posiciones contiguas de la memoria, pero el ensamblador no sabe donde será asignada esta memoria. El ensamblador también pasa al enlazador algunas entradas a la tabla de símbolos. En particular, el ensamblador debe almacenar los símbolos externos que se definen en cada fichero y las referencias sin resolver que se producen en dicho fichero.

Extensión: Por simplicidad, los ensambladores asumen que cada uno de los ficheros comienza en la misma dirección (por ejemplo, la dirección 0) con la expectativa de que el enlazador reubicará el código y los datos cuando se les asignen las posiciones en memoria. El ensamblador produce *información de reubicación*, que contiene una entrada que describe cada instrucción o palabra de datos del fichero que hace referencia a una dirección absoluta. En MIPS, sólo las instrucciones de llamada a subrutina, carga y de almacenamiento referencian direcciones absolutas. Las instrucciones que utilizan direccionamiento relativo al PC, tales como los saltos condicionales, no necesitan ser reubicadas.

Utilidades adicionales

Los ensambladores ofrecen una variedad de características que contribuyen a que los programas en lenguaje ensamblador sean más cortos y fáciles de escribir, pero no modifican de manera fundamental el lenguaje ensamblador. Por ejemplo, las directivas de declaración de datos (*data layout directivies)* permiten a un programador describir los datos de una forma más concisa y natural que su representación binaria.

En la figura B.1.4, la directiva

```
.asciiz "La suma desde 0 .. 100 es %d\n"
```

almacena la cadena de caracteres en la memoria. Contrasta esta solución con la alternativa de escribir cada carácter como su valor ASCII (la figura 2.15 en el capítulo 2 describe la codificación para caracteres ASCII):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

La directiva `.asciiz` es más fácil de leer porque representa los caracteres como letras, y no como números binarios. El ensamblador puede traducir los caracteres a su representación binaria mucho más rápido y con más exactitud que un ser

humano. Las directivas de declaración de datos especifican los datos en una forma legible para los humanos y que el ensamblador traduce a binario. Otras directivas de declaración se describen en la sección B.10.

Directiva *string* (cadena)

Definir la secuencia de bytes producidos por esta directiva:

EJEMPLO

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

RESPUESTA

```
.byte 84,  104, 101, 32,  113, 117, 105, 99
.byte 107, 32,  98,  114, 111, 119, 110, 32
.byte 102, 111, 120, 32,  106, 117, 109, 112
.byte 115, 32,  111, 118, 101, 114, 32,  116
.byte 104, 101, 32,  108, 97,  122, 121, 32
.byte 100, 111, 103, 0
```

Las macros son una herramienta de combinación de patrones y de sustitución que proporcionan un mecanismo sencillo para dar nombre a una secuencia de instrucciones utilizada frecuentemente. En lugar de escribir repetidamente las mismas secuencias de instrucciones cada vez que se usan, un programador invoca la macro y el ensamblador sustituye la llamada a la macro con la correspondiente secuencia de instrucciones. Las macros, al igual que las subrutinas, permiten al programador crear y dar nombre a una nueva abstracción para una operación habitual. A diferencia de las subrutinas, sin embargo, las macros no provocan una llamada a subrutina y un retorno cuando el programa se ejecuta, puesto que una llamada a la macro se sustituirá por el cuerpo de la macro cuando el programa se ensambla. Después de esta sustitución, el resultado ensamblado es indistinguible del programa equivalente escrito sin macros.

Macros

A modo de ejemplo, supongamos que un programador necesita imprimir muchos números. La rutina de biblioteca `printf` acepta una cadena de formato y uno o más valores para imprimir como argumentos. Un programador podría imprimir el entero en el registro `$7` con las siguientes instrucciones:

EJEMPLO

```
            .data
int_str:    .asciiz "%d"
            .text
```

```
la   $a0, int_str # Carga la dirección de la cadena
                  # en el primer argumento
mov  $a1, $7      # Carga el valor en el
                  # segundo argumento
jal printf        # Llama a la rutina printf.
```

La directiva `.data` indica al ensamblador que almacene la cadena en el segmento de datos del programa, y la directiva `.text` le dice al ensamblador que almacene las instrucciones en su segmento de texto.

Sin embargo, imprimir muchos números de esta manera es tedioso y produce un programa largo y difícil de entender. Una alternativa es introducir una macro, `print_int`, para imprimir un entero:

```
          .data
int_str:  .asciiz “%d”
          .text
          .macro print_int($arg)
          la $a0, int_str # Carga la dirección de la
                            cadena en el
                          # primer argumento
          mov $a1, $arg   # Carga el parámetro de la macro
                          # ($arg) en el segundo argumento
          jal printf      # Llama a la rutina printf.
          .end_macro
print_int($7)
```

Parámetro formal: variable que es el argumento de un procedimiento o macro; sustituido por este argumento una vez se expande la macro.

La macro tiene un **parámetro formal**, `$arg`, que identifica el argumento de la macro. Cuando la macro se expande, el argumento de la llamada es sustituido por el parámetro formal en todo el cuerpo de la macro. Luego, el ensamblador sustituye la llamada con el nuevo cuerpo ampliado de la macro. En la primera llamada a `print_int`, el argumento es `$7`, por lo que la macro se expande al código

```
la  $a0, int_str
mov $a1, $7
jal printf
```

En una segunda llamada a `print_int`, digamos, `print_int ($t0)`, el argumento es `$t0`, por lo que la macro se expande a

```
la  $a0, int_str
mov $a1, $t0
jal printf
```

¿A qué se expandiría la llamada `print_int($a0)`?

```
la  $a0, int_str
mov $a1, $a0
jal printf
```

RESPUESTA

Este ejemplo ilustra una desventaja de las macros. Un programador que utiliza esta macro debe ser consciente de que `print_int` usa el registro `$a0` y por tanto no puede imprimir correctamente el valor en este registro.

Interfaz hardware software

Algunos ensambladores también implementan *pseudoinstrucciones*, que son instrucciones que ofrece el ensamblador, pero que no se han implementado en hardware. El capítulo 2 contiene muchos ejemplos de como el ensamblador MIPS sintetiza pseudoinstrucciones y modos de direccionamiento a partir del espartano repertorio de instrucciones hardware MIPS. Por ejemplo, la sección de 2.7 en el capítulo 2 describe cómo el ensamblador sintetiza la instrucción `blt` a partir de otras dos instrucciones: `slt` y `bne`. Ampliando el conjunto de instrucciones, el ensamblador MIPS facilita la programación en lenguaje ensamblador sin complicar el hardware. Muchas pseudoinstrucciones se podrían simular con macros, pero el ensamblador MIPS puede generar un código mejor para estas instrucciones, ya que puede utilizar un registro dedicado (`$at`) y es capaz de optimizar el código generado.

Extensión: Los ensambladores *ensamblan condicionalmente* trozos de código, lo que permite al programador incluir o excluir grupos de instrucciones cuando se ensambla un programa. Esta característica es especialmente útil cuando algunas versiones de un programa difieren en una parte pequeña. En lugar de mantener estos programas en ficheros separados —lo que complica enormemente la corrección de errores en el código común— los programadores suelen fusionar las versiones en un solo fichero. El código particular de una versión se ensambla condicionalmente, de forma que pueda ser excluido cuando se ensamblan otras versiones del programa.

Si las macros y el ensamblado condicional son útiles, ¿por qué los ensambladores de sistemas UNIX rara vez, o nunca, las proporcionan? Una de las razones es que la mayoría de los programadores en estos sistemas escriben programas en lenguaje de alto nivel como C. La mayor parte del código en ensamblador lo producen los compiladores, para los que resulta más fácil repetir el código en lugar de definir macros. Otra razón es que otras herramientas de UNIX —como `cpp`, el preprocesador C, o `m4`, un procesador general de macros— pueden proporcionar macros y ensamblado condicional para programas en lenguaje ensamblador.

B.3 Enlazadores

Compilación separada: dividir un programa en muchos ficheros, cada uno de los cuales puede ser compilado sin conocimiento de lo que está en los otros ficheros.

La **compilación separada** permite dividir un programa en partes que son almacenados en ficheros diferentes. Cada fichero contiene una colección relacionada lógicamente de subrutinas y estructuras de datos que forman un *módulo* de un programa mayor. Un fichero puede ser compilado y ensamblado de forma independiente de otros ficheros, de manera que los cambios en un módulo no requieren recompilar el programa entero. Como ya se ha visto anteriormente, la compilación separada requiere el paso adicional de enlazar para combinar los ficheros objetos de módulos separados y resolver las referencias pendientes.

La herramienta que combina estos ficheros es el enlazador (véase figura B.3.1). Realiza tres tareas:

- Busca en las bibliotecas del programa para encontrar las rutinas de las bibliotecas que utiliza el programa.
- Determina las posiciones de memoria que va a ocupar el código de cada módulo y reubica sus instrucciones mediante un ajuste de las referencias absolutas.
- Resuelve referencias entre ficheros.

La primera tarea de un enlazador es asegurar que un programa no contenga etiquetas sin definir. El enlazador combina los símbolos externos y las referencias sin resolver de los ficheros de un programa. Un símbolo externo de un fichero resuelve una referencia de otro fichero si ambos hacen referencia a una etiqueta con el mismo nombre. Las referencias sin resolver significan que se han utilizado símbolos que no se definen en ninguna parte del programa.

Las referencias sin resolver en esta etapa del proceso de enlazar no significan necesariamente que el programador cometiera un error. El programa podría tener referencias a rutinas de biblioteca, cuyo código no está en los ficheros objeto pasados al enlazador. Después de emparejar los símbolos del programa, el enlazador busca en bibliotecas de los programas del sistema para encontrar las subrutinas predefinidas y las estructuras de datos referenciados por el programa. Las bibliotecas básicas contienen rutinas que leen y escriben datos, reservan y liberan memoria, y realizan operaciones numéricas. Otras bibliotecas contienen rutinas para acceder a bases de datos o manipular las ventanas del terminal. Un programa que hace referencia a un símbolo sin resolver que no se encuentra en ninguna biblioteca es erróneo y no puede enlazarse. Cuando el programa utiliza una rutina de la biblioteca, el enlazador extrae el código de la rutina de la biblioteca y lo incorpora en el segmento de texto del programa. Esta nueva rutina, a su vez, puede depender de otras rutinas de la biblioteca, por lo que el enlazador sigue buscando otras rutinas de la biblioteca hasta que ninguna referencia externa esté sin resolver hasta que una rutina no se pueda encontrar.

Si todas las referencias externas se resuelven, el enlazador, a continuación, determina las posiciones de memoria que ocupará cada módulo. Dado que los

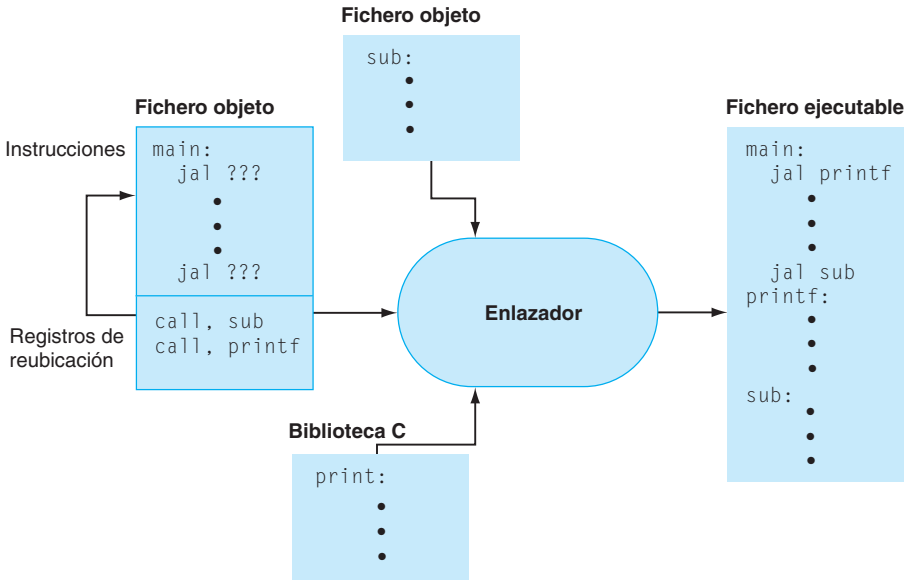


FIGURA B.3.1 El enlazador busca en un conjunto de ficheros objeto y bibliotecas de programas para encontrar las rutinas no locales utilizadas en un programa, las combina en un solo fichero ejecutable, y resuelve las referencias entre rutinas de diferentes ficheros.

ficheros se ensamblan por separado, el ensamblador no podría saber qué posición van a ocupar las instrucciones o los datos de cada módulo que se pondrá en relación con otros módulos. Cuando el enlazador coloca un módulo en memoria, todas las referencias absolutas se deben reasignar para reflejar su verdadera localización. Puesto que el enlazador tiene información de reasignación que identifica todos las referencias reasignables, puede encontrarlas eficientemente y arreglar (*backpatch*) convenientemente estas referencias.

El enlazador genera un fichero ejecutable que puede ejecutarse en un computador. Normalmente, este fichero tiene el mismo formato que un fichero objeto, con la diferencia de que no contiene referencias no resueltas o información de reubicación.

B.4 Cargador

Un programa enlazado sin ningún error se puede ejecutar. Antes de ser ejecutado, el programa reside en un fichero de almacenamiento secundario, como un disco. En los sistemas UNIX, el núcleo del sistema operativo trae un programa a la

memoria y comienza a ejecutarlo. Para iniciar un programa, el sistema operativo realiza los siguientes pasos:

1. Lee la cabecera del fichero ejecutable para determinar el tamaño de los segmentos de texto y datos.
2. Crea un nuevo espacio de direcciones para el programa. Este espacio de direcciones es lo suficientemente grande como para mantener los segmentos de texto y de datos, junto con un segmento de pila (véase sección B.5).
3. Copia instrucciones y datos desde el fichero ejecutable, en el nuevo espacio de direcciones.
4. Copias los argumentos pasados al programa en la pila.
5. Da un valor inicial a los registros de la máquina. En general, la mayoría de los registros se borran, pero al puntero de pila hay que asignarle la dirección de la primera posición libre de la pila (véase sección B.5).
6. Salta a la rutina de puesta en marcha que copia los argumentos del programa de la pila a los registros y llama a la rutina `main` del programa. Si la rutina `main` retorna, la rutina de puesta en marcha termina el programa con una llamada al sistema de salida.

B.5 Utilización de la memoria

Las secciones siguientes explican detalladamente la descripción de la arquitectura MIPS presentadas anteriormente en el libro. Los capítulos anteriores se centraron principalmente en el hardware y su relación con el software de bajo nivel. Estas secciones se centran principalmente en cómo los programadores de lenguaje ensamblador utilizan el hardware MIPS. Estas secciones describen un conjunto de convenios seguidos por la mayoría de los sistemas MIPS. En su mayor parte, el hardware no impone estos convenios. En lugar de ello, representan un acuerdo entre los programadores para seguir las mismas normas con el fin de que programas escritos por personas diferentes puedan funcionar juntos y hacer un uso eficaz del hardware de MIPS.

Los sistemas basados en procesadores MIPS suelen dividir la memoria en tres partes (véase figura B.5.1). La primera parte, junto al extremo inferior del espacio de direcciones (a partir de la dirección 400000_{hex}), es el *segmento de texto*, que contiene las instrucciones del programa.

La segunda parte, sobre el segmento de texto, es el *segmento de datos*, que se divide en dos partes. La parte de **datos estáticos** (a partir de la dirección 10000000_{hex}) contiene objetos cuyo tamaño es conocido por el compilador y cuya vida útil —el intervalo durante el cual un programa puede acceder a ellos— es el tiempo que tarda en ejecutarse el programa completo. Por ejemplo, en C, las variables globales son asignadas estáticamente, ya que se pueden referenciar en cualquier momento durante una ejecución del programa. El enlazador asigna objetos estáticos a las posiciones en el segmento de datos y resuelve las referencias a estos objetos.

Datos estáticos: parte de la memoria que contiene los datos cuyo tamaño es conocido por el compilador y cuya vida útil es la duración de la ejecución entera del programa.

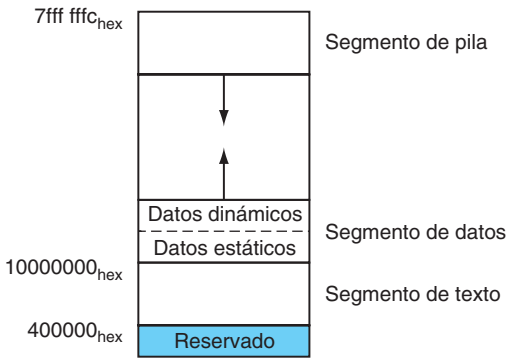


FIGURA B.5.1 Esquema de la memoria.

Interfaz hardware software

Debido a que el segmento de datos comienza muy por encima del programa, en la dirección 10000000_{hex}, las instrucciones cargar y almacenar (*load* y *store*) no pueden referenciar directamente objetos de datos con su campo de desplazamiento de 16 bits (véase sección 2.5 en el capítulo 2). Por ejemplo, para cargar la palabra en el segmento de datos en la dirección 10010020_{hex} en el registro `$v0` requiere dos instrucciones:

```
lui $s0, 0×1001   # 0×1001 significa  1001 base 16
lw  $v0, 0×0020($s0) # 0×10010000 + 0×0020 = 0×10010020
```

(El *0x* delante de un número significa que es un valor hexadecimal. Por ejemplo, `0×8000` es 8000_{hex} ó 32768_{diez}).

Para evitar la repetición de la instrucción `lui` en cada *load* o *store*, los sistemas MIPS suelen dedicar un registro `($gp)` como un *puntero global* al segmento de datos estáticos. Este registro contiene la dirección 10008000_{hex}, de forma que las instrucciones cargar y almacenar puedan utilizar su campo de desplazamiento de 16 bits con signo para acceder a los primeros 64 KB del segmento de datos estático. Con este puntero global, podemos reescribir el ejemplo como una única instrucción:

```
lw $v0, 0×8020($gp)
```

Por supuesto, un registro de puntero global hace el direccionamiento a las posiciones 10000000_{hex}-10010000_{hex} más rápido que a otras posiciones dinámicas. El compilador MIPS normalmente almacena las *variables globales* en este área debido a que estas variables tienen una direcciones fijas y se ajustan mejor que otros datos globales, como las matrices.

Inmediatamente por encima de los datos estáticos están los *datos dinámicos*. Estos datos, como su nombre indica, son asignados por el programa durante su

ejecución. En programas en C, la rutina de biblioteca `malloc` busca y devuelve un bloque nuevo de memoria. Desde un compilador no se puede predecir la cantidad de memoria que asignará un programa, el sistema operativo amplía el área de datos dinámicos para satisfacer la demanda. Tal como indica en la figura la flecha hacia arriba, `malloc` amplía el área dinámica con la llamada al sistema `sbrk`, que hace que el sistema operativo añada más páginas al espacio de direcciones virtual del programa (véase sección 5.4 en el capítulo 5) inmediatamente por encima del segmento de datos dinámicos.

Segmento de pila: parte de la memoria utilizada por un programa para mantener las referencias de llamadas a procedimientos.

La tercera parte, el **segmento de pila (*stack segment*)** del programa, reside en la parte superior del espacio de direcciones virtual (empezando en la dirección $7fffffff_{hex}$). Como los datos dinámicos, el tamaño máximo de la pila de un programa no se conoce de antemano. A medida que el programa apila los valores en la pila, el sistema operativo expande el segmento de pila hacia abajo, hacia el segmento de datos.

Esta división de la memoria en tres partes no es la única posible. Sin embargo, tiene dos características importantes: los dos segmentos que se expanden dinámicamente están lo más alejados posible, y pueden crecer hasta llegar a utilizar el espacio de direcciones entero del programa.

B.6 Convenio de llamada a procedimiento

Cuando los procedimientos de un programa se compilan por separado, se necesitan convenios que gobiernen el uso de los registros. Para compilar un procedimiento particular, un compilador debe saber qué registros puede utilizar y qué registros están reservados para otros procedimientos. A las reglas para el uso de registros se les llama **uso de registros** o **convenios de llamada a procedimiento**. Como el nombre implica, estas reglas son, en su mayor parte, convenios que sigue el software en lugar de normas aplicadas por hardware. Sin embargo, la mayoría de los compiladores y los programadores se esfuerzan en seguir estos convenios, porque su violación causa errores insidiosos.

Convenio de uso de registro (también llamado **convenios de llamada a procedimientos**): protocolo de software que rige el uso de los registros de los procedimientos.

El convenio de llamada que se describe en esta sección es el utilizado por el compilador gcc. El compilador nativo MIPS utiliza un convenio más complejo que es ligeramente más rápido.

La CPU MIPS tiene 32 registros de propósito general que están numerados 0-31. El registro `$0` siempre contiene el valor cableado 0.

- Los registros `$at(1)`, `$k0` (26) y `$k1` (27) están reservados para el ensamblador y el sistema operativo y no deben ser utilizados por programas de usuario o compiladores.
- Los registros `$a0-$a3` (4-7) se utilizan para pasar los primeros cuatro argumentos a las rutinas (el resto de los argumentos se pasan en la pila). Los registros `$v0` y `$v1` (2, 3) se utilizan para devolver valores de funciones.

- Los registros `$t0-$t9` (8-15, 24, 25) son **Registros salvados del invocador o llamador (*caller-saved registers*)** que se utilizan para mantener temporalmente datos que no necesitan ser preservados durante las llamadas (véase sección 2.8 en el capítulo 2).
- Los registros `$s0-$s7` (16-23) son **registros salvados del invocado o llamado (*callee-saved registers*)** que mantienen valores de vida más larga que deben preservarse a través de llamadas.
- El registro `$gp` (28) es un puntero global que apunta a la mitad de un bloque de 64K de memoria en el segmento de datos estáticos.
- El registro `$sp` (29) es un puntero de pila, que apunta a la última posición en la pila. El registro `$fp` (30) es el puntero de bloque de activación (*frame pointer*). La instrucción `jal` escribe en el registro `$ra` (31), la dirección de retorno de una llamada a procedimiento. Estos dos registros se explican en la siguiente sección.

Registros salvados del invocador o llamador: registro salvado por la rutina que está siendo llamada.

Registros salvados del invocado o llamado: registro salvado por la rutina que está haciendo un procedimiento de llamada.

Las abreviaturas de dos letras y los nombres de estos registros —por ejemplo, `$sp` para el puntero de pila— reflejan los usos previstos de los registros en los convenios de llamada a procedimiento. Al describir este convenio, vamos a usar los nombres en lugar de los números de registro. La figura B.6.1 lista los registros y describe sus usos previstos.

Llamadas a procedimiento

Esta sección describe los pasos que ocurren cuando un procedimiento (el invocador o llamador (*the caller*)) invoca a otro procedimiento (el invocado o llamado (*callee*)). Los programadores que escriben en un lenguaje de alto nivel (como Pascal o C) nunca ven los detalles de cómo un procedimiento llama a otro porque el compilador se encarga de esta gestión de bajo nivel. Sin embargo, los programadores de lenguaje ensamblador deben implementar explícitamente cada llamada y retorno de procedimiento.

La mayor parte de la gestión asociada con las llamadas se centra en torno a un bloque de memoria llamado **bloque de llamada a procedimiento (*procedure call frame*)**. Esta memoria se utiliza con varios propósitos:

Bloque o marco de llamada a procedimiento: bloque de memoria que se utiliza para mantener los valores pasados a un procedimiento como argumentos, para salvar los registros que un procedimiento puede modificar pero que el procedimiento invocador no los quiere cambiados, y proporciona espacio para las variables locales a un procedimiento.

- Mantener los valores pasados a un procedimiento como argumentos.
- Salvar los registros que un procedimiento que puede modificar, pero que el procedimiento invocador no quiere que cambien.
- Disponer de espacio para las variables locales a un procedimiento.

En la mayoría de los lenguajes de programación, las llamadas a procedimiento y los retornos siguen un orden estricto "último en entrar, primero en salir" (*last-in, first-out,* LIFO), por lo que esta memoria puede ser asignada y liberada en una pila, es por ello que estos bloques de memoria a veces se llaman bloques de pila (*stack frames*).

La figura B.6.2 muestra un típico bloque de pila. El bloque se compone de la memoria entre el puntero de bloque de activación (`$fp`), que apunta a la primera palabra del bloque, y el puntero de pila (`$sp`), que apunta a la última palabra del

Nombre del registro	Número	Uso
$zero	0	Constante 0
$at	1	Reservado para ensamblador
$v0	2	Evaluación de expresión y resultado de función
$v1	3	Evaluación de expresión y resultado de función
$a0	4	Argumento 1
$a1	5	Argumento 2
$a2	6	Argumento 3
$a3	7	Argumento 4
$t0	8	Temporal (no mantenido durante la llamada)
$t1	9	Temporal (no mantenido durante la llamada)
$t2	10	Temporal (no mantenido durante la llamada)
$t3	11	Temporal (no mantenido durante la llamada)
$t4	12	Temporal (no mantenido durante la llamada)
$t5	13	Temporal (no mantenido durante la llamada)
$t6	14	Temporal (no mantenido durante la llamada)
$t7	15	Temporal (no mantenido durante la llamada)
$s0	16	Temporal salvado (mantenido durante la llamada)
$s1	17	Temporal salvado (mantenido durante la llamada)
$s2	18	Temporal salvado (mantenido durante la llamada)
$s3	19	Temporal salvado (mantenido durante la llamada)
$s4	20	Temporal salvado (mantenido durante la llamada)
$s5	21	Temporal salvado (mantenido durante la llamada)
$s6	22	Temporal salvado (mantenido durante la llamada)
$s7	23	Temporal salvado (mantenido durante la llamada)
$t8	24	Temporal (no mantenido durante la llamada)
$t9	25	Temporal (no mantenido durante la llamada)
$k0	26	Reservado para el núcleo del SO
$k1	27	Reservado para el núcleo del SO
$gp	28	Puntero al área global (*global pointer*)
$sp	29	Puntero de pila (*stack pointer*)
$fp	30	Puntero del bloque de activación (*frame pointer*)
$ra	31	Dirección de retorno (usado por la función de llamada)

FIGURA B.6.1 Registros del MIPS y su uso convencional.

bloque. La pila crece hacia abajo desde las direcciones de memoria superiores, por lo que el puntero de bloque de activación apunta por encima del puntero de pila. El procedimiento en ejecución utiliza el puntero de bloque de activación para acceder rápidamente a los valores en su bloque de pila. Por ejemplo, un argumento en el bloque de pila se puede cargar en el registro `$v0` con la instrucción.

```
lw $v0, 0($fp)
```

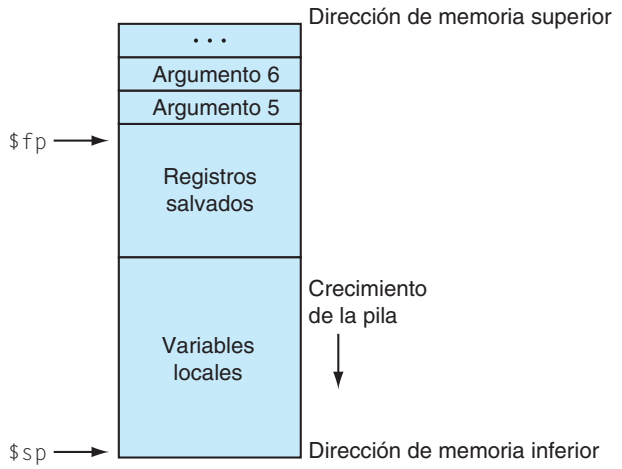


FIGURA B.6.2 Esquema de un bloque de pila. El puntero de bloque de activación (*frame pointer*) (`$fp`) apunta a la primera palabra del bloque del procedimiento que se está ejecutando. El puntero de pila (`$sp`) apunta a la última palabra del bloque. Los primeros cuatro argumentos se pasan en registros, por lo tanto el quinto argumento es el primero de los argumentos almacenado en la pila.

Un bloque de pila se puede construir de muchas formas diferentes; pero, el invocador y el invocado deben estar de acuerdo en la secuencia de pasos. Los pasos que se enumeran a continuación describen los convenios de llamadas utilizados en la mayoría de las máquinas MIPS. Este convenio interviene en tres puntos durante una llamada a procedimiento: inmediatamente antes de que el invocador llame al invocado, en el momento justo en el que el invocado empieza su ejecución, e inmediatamente antes de que el invocado retorne al invocador. En la primera parte, el invocador coloca los argumentos de la llamada a procedimiento en los lugares establecidos y llama al invocado para hacer lo siguiente:

1. Paso de argumentos: Por convenio, los cuatro primeros argumentos se pasan en los registros `$a0-$a3`. Cualquier argumento adicional se pone en la pila y aparecen al principio del bloque de pila de llamada a procedimiento.

2. Guarda los registros salvados por el invocador. El procedimiento invocado puede usar estos registros (`$a0-$a3` y `$t0-$t9`) sin salvar previamente su valor. Si el invocador espera utilizar uno de estos registros después de una llamada, debe salvar su valor antes de la llamada.

3. Ejecuta una instrucción `jal` (véase sección 2.8 del capítulo 2), que salta a la primera instrucción del invocado y guarda la dirección de retorno en el registro `$ra`.

Antes de que una rutina invocada comience a ejecutarse, es necesario llevar a cabo los siguientes pasos para preparar su bloque de pila:

1. Asignar memoria para el bloque restando el tamaño del bloque del puntero de pila.
2. Guardar los registros salvados por el invocado en el bloque de pila. El invocado debe guardar los valores en estos registros (`$s0-$s7`, `$fp`, `$ra`) antes de modificarlos, puesto que el invocador espera encontrar estos registros sin cambios después de la llamada. El registro `$fp` es salvado por cada procedimiento que asigna un nuevo bloque de pila. Sin embargo, el registro `$ra` sólo necesita ser salvado si el invocado hace una llamada. Los otros registros salvados por el invocado que se vayan a utilizar también deben salvarse.
3. Establecer el puntero de bloque de activación añadiendo el tamaño del bloque de pila menos 4 a `$sp` y almacenando la suma en el registros `$fp`.

Interfaz hardware software

El convenio de uso de registros MIPS proporciona registros salvados por el invocado —y por el invocador— porque ambos tipos de registros tiene ventajas en diferentes circunstancias. Los registros salvados por el invocado se utilizan para mantener los valores de vida larga, tales como las variables de un programa de usuario. Estos registros se guardan sólo durante una llamada a procedimiento si el invocado espera utilizar el registro. Por otra parte, los registros salvados por el invocador se utilizan para mantener valores de corta duración que no persisten a través de una llamada, como los valores inmediatos en un cálculo de direcciones. Durante una llamada, el invocado también puede usar estos registros para variables temporales de vida corta.

Por último, el invocado retorna al invocador ejecutando los siguientes pasos:

1. Si el invocado es una función que devuelve un valor, coloca el valor a devolver en el registro `$v0`.
2. Restaura todos los registros salvados por el invocado que se salvaron a la entrada del procedimiento.
3. Saca de la pila el bloque de pila sumando el tamaño del bloque a `$sp`.
4. Retorna saltando a la dirección indicada en el registro `$ra`.

Procedimientos recursivos: procedimientos que se llaman a sí mismos, ya sea directa o indirectamente a través de una cadena de llamadas.

Extensión: En un lenguaje de programación que no permita **procedimientos recursivos** —procedimientos que se llaman a sí mismos, ya sea directa o indirectamente a través de una cadena de llamadas— no es necesario asignar los bloques a una pila. En un lenguaje no recursivo, cada bloque de procedimiento puede ser asignado estáticamente ya que sólo puede estar activa al mismo tiempo una invocación a un procedimiento. Las versiones antiguas de Fortran prohibían la recursividad, porque los bloques asignados estáticamente producían código más rápido en algunas máquinas más antiguas. Sin embargo, en una

arquitectura de carga-almacenamiento como MIPS, los bloques de pila pueden ser igual de rápidos, porque un registro puntero de bloque de activación apunta directamente al bloque de pila activo, lo que permite una sola instrucción de carga o almacenamiento para acceder a los valores en el bloque. Además, la recursión es una valiosa técnica de programación.

Ejemplo de llamada a procedimiento

A modo de ejemplo, consideremos la rutina C:

```
main ()
{
   printf ("El factorial de 10 es %d\n", fact (10));
}

int fact (int n)
{
   if (n < 1)
      return (1);
   else
      return (n * fact (n - 1));
}
```

que calcula e imprime 10! (el factorial de 10, 10! = 10 × 9 ×... × 1). `fact` es una rutina recursiva que calcula *n*! multiplicando *n* veces (*n* – 1)!. El código ensamblador de esta rutina muestra cómo los programas manipulan los bloques de pila.

A partir de la entrada, la rutina `main` crea su bloque de pila y guarda los dos registros salvados por el invocado que se van a modificar: `$fp` y `$ra`. El bloque es mayor que lo requerido para estos dos registros debido a que el convenio de llamadas requiere que el tamaño mínimo de un bloque de pila sea 24 bytes. Este bloque mínimo puede mantener cuatro registros de argumento (`$a0-$a3`) y la dirección de retorno `$ra`, extendida al límite de tamaño de doble-palabra (24 bytes). Como la rutina `main` también necesita salvar `$fp`, su bloque de pila debe agrandarse dos palabras (recuerde: el puntero de pila se mantiene alineado a doble palabra).

```
    .text
    .globl main
main:
    subu  $sp,$sp,32  # El bloque de pila es de 32 bytes
    sw    $ra,20($sp) # Salva la dirección de retorno
    sw    $fp,16($sp) # Salva el puntero de bloque de
                        activación anterior
    addiu $fp,$sp,28  # Actualiza el puntero de bloque
                        de activación
```

La rutina `main` después llama a la rutina factorial y le pasa 10 como único argumento. Después `fact` retorna, `main` llama a la rutina de biblioteca `printf` y le pasa una cadena de formato y el resultado devuelto por `fact`:

```
        li    $a0,10       # Pone el argumento(10) en $a0
        jal   fact         # Llama a la función factorial

        la    $a0,$LC      # Pone la cadena de formato en $a0
        move  $a1,$v0      # Mueve el resultado de fact a $a1
        jal   printf       # Llama a la función de imprimir
```

Por último, después de imprimir el factorial, `main` retorna. Pero antes es preciso restaurar los registros que ha salvado y sacar de la pila su bloque de pila:

```
        lw    $ra,20($sp) # Restaura la dirección de retorno
        lw    $fp,16($sp) # Restaura el puntero de bloque
                            de activación
        addiu $sp,$sp,32  # Saca de la pila el bloque de pila
        jr    $ra         # Retorna al invocador

        .rdata
$LC:
        .ascii "El factorial de 10 es %d\n\000"
```

La rutina factorial es similar en estructura a `main`. En primer lugar, se crea un bloque de pila y guarda los registros salvados por el invocado que va a utilizar. Además de salvar `$ra` y `$fp`, `fact` también guarda su argumento (`$a0`), que utilizará para la llamada recursiva:

```
        .text
fact:
        subu  $sp,$sp,32  # El bloque de pila tiene 32 bytes
        sw    $ra,20($sp) # Salva la dirección de retorno
        sw    $fp,16($sp) # Salva el puntero de bloque de
                            activación
        addiu $fp,$sp,28  # Actualiza el puntero de bloque
                            de activación
        sw    $a0,0($fp)  # Salva el argumento (n)
```

El corazón de la rutina `fact` realiza el cálculo del programa C. Se comprueba si el argumento es mayor que 0. Si no es así, la rutina devuelve el valor 1. Si el argumento es mayor que 0, la rutina se llama a sí misma recursivamente para calcular `fact (n-1)` y multiplica el valor por *n*:

```
        lw    $v0,0($fp)  # Carga n
        bgtz  $v0,$L2     # Salta si n > 0
        li    $v0,1       # Devuelve 1
        jr    $L1         # Salta al código de retorno

$L2:
        lw    $v1,0($fp)  # Carga n
        subu  $v0,$v1,1   # Cálcula n - 1
        move  $a0,$v0     # Mueve el valor a $a0
```

```
        jal    fact         # Llama a la función factorial

        lw     $v1,0($fp)   # Carga n
        mul    $v0,$v0,$v1  # Calcula fact(n-1) * n
```

Finalmente, la rutina factorial restaura los registros salvados por el invocado y retorna el valor en el registro `$v0`

```
$L1:                        # El resultado está en $v0
        lw     $ra, 20($sp) # Restaura $ra
        lw     $fp, 16($sp) # Restaura $fp
        addiu  $sp, $sp, 32 # Saca de la pila el bloque de pila
        jr     $ra          # Retorna al invocador
```

La pila en un procedimiento recursivo

EJEMPLO

La figura B.6.3 muestra la pila de la llamada `fact(7)`. La rutina `main` es la que se ejecuta primero, por lo que su bloque es el más profundo en la pila. `main` llama a `fact(10)`, cuyo bloque de pila es el siguiente en la pila. Cada invocación de forma recursiva invoca `fact` para calcular el siguiente factorial. Los bloques de pila paralelos reproducen el orden LIFO de estas llamadas. ¿Qué le parece que hace la pila cuando la llamada a `fact(10)` retorna?

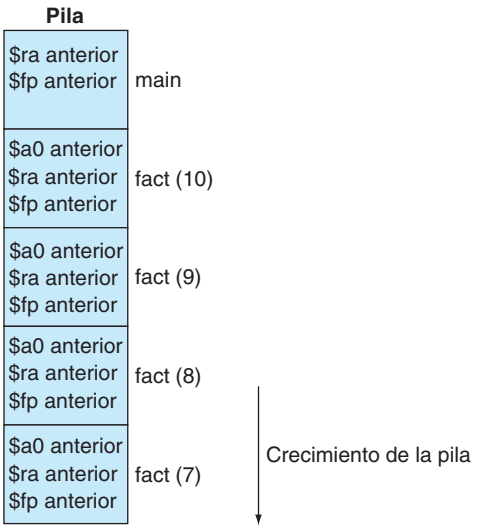
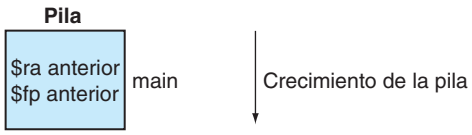


FIGURA B.6.3 Bloque de pila durante la llamada de `fact(7)`.

RESPUESTA



Extensión: La diferencia entre el compilador MIPS y el compilador gcc es que el primero no suele usar un puntero de bloque de activación, por lo que este registro está disponible como otro registro salvado por el invocado, `$S8`. Este cambio ahorra un par de instrucciones en la secuencia de llamada y retorno de procedimiento. Sin embargo, la generación de código se complica debido a que un procedimiento debe acceder a su bloque de pila con `$sp`, cuyo valor puede cambiar durante la ejecución de un procedimiento, si se apilan valores en la pila.

Otro ejemplo de llamada a un procedimiento

Como otro ejemplo, considere la siguiente rutina que calcula la función `tak`, que es un programa de prueba (*benchmark*) muy utilizado creado por Ikuo Takeuchi. Esta función no calcula nada útil, sino que es un programa altamente recursivo que ilustra los convenios de llamadas MIPS.

```
int tak (int x, int y, int z)
{
    if (y < x)
       return 1+ tak (tak (x - 1, y, z),
         tak (y - 1, z, x),
         tak (z - 1, x, y));
    else
       return z;
}
int main ()
{
    tak(18, 12, 6);
}
```

El código ensamblador para este programa se muestra a continuación. La función tak primero salva su dirección de retorno en su bloque de pila y sus argumentos en registros salvados por el invocado, ya que la rutina puede hacer que las llamadas que necesitan utilizar los registros `$a0-$a2` y `$ra`. La función utiliza registros salvados por el invocado ya que mantienen valores que persisten durante la vida útil de la función, que incluye varias llamadas que podrían modificar dichos registros.

```
        .text
        .globl   tak
```

```
tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)

    sw      $s0, 16($sp)      # x
    move    $s0, $a0
    sw      $s1, 20($sp)      # y
    move    $s1, $a1
    sw      $s2, 24($sp)      # z
    move    $s2, $a2
    sw      $s3, 28($sp)      # temporal
```

Entonces la rutina empieza la ejecución comprobando si `y` < `x`. Si no, salta a la etiqueta `L1`, que está más adelante

```
    bge     $s1, $s0, L1      # si (y < x)
```

Si `y` < `x`, entonces ejecuta el cuerpo de la rutina, que contiene cuatro llamadas recursivas. La primera llamada utiliza casi los mismos argumentos que su padre

```
    addiu   $a0, $s0, -1
    move    $a1, $s1
    move    $a2, $s2
    jal     tak               # tak (x - 1, y, z)
    move    $s3, $v0
```

Observe que el resultado de la primera llamada recursiva se salva en el registro `$s3`, de modo que puede ser utilizado posteriormente.

La función ahora prepara los argumentos para la segunda llamada recursiva.

```
    addiu   $a0, $s1, -1
    move    $a1, $s2
    move    $a2, $s0
    jal     tak               # tak (y - 1, z, x)
```

En las instrucciones siguientes, el resultado de esta llamada recursiva es salvado en el registro `$s0`. Pero primero se necesita leer, por última vez, el valor salvado del primer argumento de este registro.

```
    addiu   $a0, $s2, -1
    move    $a1, $s0
    move    $a2, $s1
    move    $s0, $v0
    jal     tak               # tak (z - 1, x, y)
```

Después de las tres llamadas recursivas internas, estamos listos para la llamada recursiva final. Tras la llamada, el resultado de la función está en `$v0` y el control salta al epílogo de la función.

```
move    $a0, $s3
move    $a1, $s0
move    $a2, $v0
jal     tak         # tak (tak(...), tak(...), tak(...))
addiu   $v0, $v0, 1
j       L2
```

Este código en la etiqueta `L1` es la salida de la sentencia *if-then-else*. Simplemente mueve el valor del argumento `z` al registro de retorno y continúa en la función de epílogo.

```
L1:
     move     $v0, $s2
```

El código siguiente es la función epílogo, que restaura los registros salvados y retorna el resultado de la función a su invocador.

```
L2:
     lw       $ra, 32($sp)
     lw       $s0, 16($sp)
     lw       $s1, 20($sp)
     lw       $s2, 24($sp)
     lw       $s3, 28($sp)
     addiu    $sp, $sp, 40
     jr       $ra
```

La rutina `main` llama a la función `tak` con sus argumentos iniciales, después toma el resultado calculado (7) y lo imprime utilizando la llamada de sistema de SPIM para imprimir enteros.

```
     .globl   main
main:
     subu     $sp, $sp, 24
     sw       $ra, 16($sp)

     li       $a0, 18
     li       $a1, 12
     li       $a2,  6
     jal      tak              # tak (18, 12, 6)
```

```
move      $a0,  $v0
li        $v0, 1            # llamada a sistema
                              (syscall) print_int
syscall

lw        $ra, 16 ($sp)
addiu     $sp, $sp, 24
jr        $ra
```

B.7 Excepciones e interrupciones

La sección 4.9 del capítulo 4 describe el mecanismo de excepciones del MIPS, que responde tanto a las excepciones causadas por errores durante la ejecución de una instrucción como a las interrupciones externas causadas por dispositivos de E / S. Esta sección describe el **manejo de** excepciones e **interrupciones** con más detalle.[5] En los procesadores MIPS, una parte de la CPU llamada *coprocesador 0* almacena la información que necesita el software para manejar excepciones e interrupciones. El simulador de MIPS SPIM no implementa todos los registros del coprocesador 0, ya que muchos no son útiles en un simulador o forman parte del sistema de memoria, que SPIM no modela. Sin embargo, SPIM proporciona los siguientes registros del coprocesador 0:

Manejador de interrupciones: trozo de código que se ejecuta como resultado de una excepción o una interrupción.

Nombre del registro	Número de registro	Uso
BadVAddr	8	Dirección de memoria en la que ha ocurrido el problema de referencia a memoria
Count	9	Contador o temporizador
Compare	11	Valor comparado con el contador que causa la interrupción cuando coinciden
Status	12	Máscara de interrupciones y bits de autorización
Cause	13	Tipo de excepción y bits de interrupción pendiente
EPC	14	Dirección de la instrucción que causó la excepción
Config	16	Configuración de máquina

5. En esta sección se examinan las excepciones en la arquitectura MIPS32, que es la que se implementa en la versión 7.0 de SPIM y posteriores. Las versiones anteriores de SPIM implementaban la arquitectura MIPS-I, que maneja de forma ligeramente diferente las excepciones. La conversión de programas de estas versiones para funcionar en MIPS32 no debería ser difícil, ya que los cambios se limitan a campos del registro de estado y control, así como la sustitución de la instrucción `rfe` de por la instrucción `eret`.

Estos siete registros son parte del conjunto de registros del coprocesador 0. Se accede a ellos mediante las instrucciones `mfc0` y `mtc0`. Después de una excepción, el registro EPC contiene la dirección de la instrucción que se estaba ejecutando cuando se produjo la excepción. Si la excepción fue causada por una interrupción externa, entonces la instrucción no habrá empezado la ejecución. Todas las demás excepciones son causadas por la ejecución de la instrucción por el EPC, excepto cuando la instrucción culpable del problema está en la ranura de retardo (*delay slot*) de una instrucción de salto condicional o salto incondicional. En ese caso, el EPC apunta a la instrucción de salto condicional o incondicional y se activa el bit BD del registro Cause. Cuando este bit está activo, el manejador de excepciones debe buscar en EPC + 4 la instrucción culpable del problema. Sin embargo, en ambos casos, un manejador de excepciones reanuda el programa adecuadamente volviendo a la instrucción en el EPC.

Si la instrucción que causó la excepción hizo un acceso a memoria, el registro BadVAddr contiene la dirección de la posición de memoria referenciada.

El registro *Count* es un temporizador que se incrementa a una frecuencia fija (por defecto, cada 10 milisegundos) mientras SPIM se está ejecutando. Cuando el valor en el registro *Count* es igual al valor en el registro *Compare*, se produce una interrupción hardware en el nivel de prioridad 5.

La figura B.7.1 se muestra el subconjunto de los campos del registro de Estado implementado por el simulador SPIM de MIPS. El campo máscara de interrupción (`interrupt mask`) contiene un bit para cada uno de los seis niveles de interrupción harware y los dos niveles de interrupción software. Un bit igual 1 en la máscara permite interrupciones a ese nivel en el procesador. Un bit a 0 en la máscara deshabilita las interrupciones a ese nivel. Cuando llega una interrupción, activa su bit de interrupción pendiente en el registro *Cause*, aun cuando el bit de máscara esté desactivado. Cuando una interrupción está pendiente, se interrumpirá el procesador cuando su bit de máscara sea posteriormente habilitado.

El bit de modo de usuario es 0 si el procesador está ejecutando en modo núcleo (*kernel*) y 1 en caso de que esté ejecutando en modo de usuario. En SPIM, este bit se fija a 1, puesto que el procesador SPIM no implementa el modo núcleo (*kernel*). El bit de nivel de excepción está normalmente a 0, pero se activa a 1 después de que ocurre una excepción. Cuando este bit está a 1, las interrupciones se deshabilitan y el EPC no se actualiza si se produce otra excepción. Este bit impide que un manejador de excepción sea perturbado por una interrupción o excepción, pero hay que restablecerlo cuando el manejador acabe. Si el bit "permiso de interrupción" (`interrupt enable`) es 1, se permiten las interrupciones; si es 0, no se permiten.

La figura B.7.2 muestra el subconjunto de campos del registro Causa que implementa SPIM. El bit de retardo de salto condicional (*branch delay)* es 1, si la última excepción se produjo en una instrucción ejecutada en la ranura de retardo (*slot delay*) de un salto condicional. Los bits de interrupción pendiente se activan a 1 cuando una interrupción ocurre en un determinado nivel hardware o software. El registro de código de excepción describe la causa de una excepción a través de los siguientes códigos:

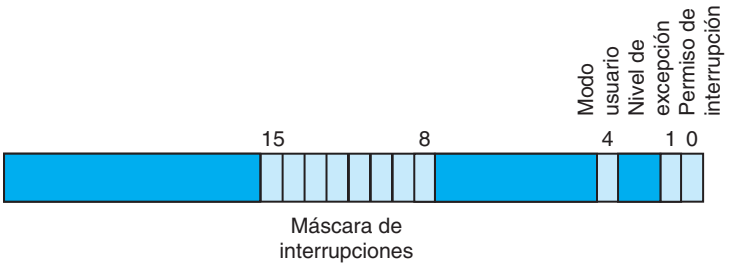


FIGURA B.7.1 El Registro de Estado (*Status register*).

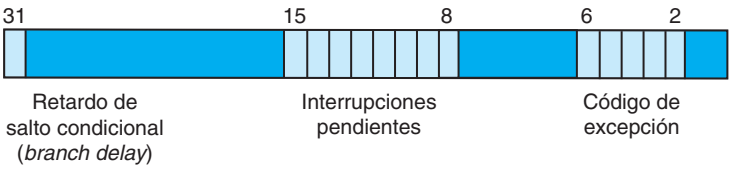


FIGURA B.7.2 El Registro Causa (*Cause register*).

Número	Nombre	Causa de la excepción
0	Int	Interrupción (hardware)
4	AdEL	Excepción de dirección errónea (cargar o buscar instrucción)
5	AdES	Excepción de dirección errónea (almacenar)
6	IBE	Error de bus en búsqueda de instrucción
7	DBE	Error de bus en carga o almacenamiento de datos
8	Sys	Excepción de llamada a sistema
9	Bp	Excepción de punto de parada (*breakpoint*)
10	RI	Excepción de instrucción reservada
11	CpU	Coprocesador no implentado
12	*Overhead*	Excepción de desbordamiento aritmético
13	Tr	Trap
15	FPE	Punto flotante

Las excepciones y las interrupciones causan que el procesador MIPS salte a un segmento de código, en la dirección 80000180_{hex} (en el espacio de direcciones del núcleo, no del usuario), denominado *manejador de excepciones* (*exception handler*). Este código examina la causa de la excepción y salta a un punto adecuado en el sistema operativo. El sistema operativo responde a una excepción, o bien terminando el proceso que la ha causado, o bien realizando alguna acción. Un proceso que causa un error, como la ejecución de una instrucción no existente, es abortado por el sistema operativo. Por otra parte, otras excepciones, tales como fallos de página son peticiones de un proceso al sistema operativo para llevar a cabo un servicio, tal

como traer una página de disco. El sistema operativo procesa estas peticiones y se reanuda el proceso. El último tipo de excepciones son las interrupciones de dispositivos externos. Éstos generalmente provocan que el sistema operativo mueva datos hacia o desde un dispositivo de E / S y reanude el proceso interrumpido.

El código en el ejemplo siguiente es un simple manejador de excepciones, que invoca una rutina para imprimir un mensaje para cada excepción (aunque no interrumpe). Este código es similar al manejador de excepciones (`exceptions.s`) utilizado por el simulador SPIM.

EJEMPLO

Manejador de excepciones

El manejador de excepciones primero salva el registro `$at`, que se utiliza en pseudoinstrucciones en el manejador de código, entonces salva `$a0` y `$a1`, que más tarde se utilizan para pasar argumentos. El manejador de excepciones no puede salvar los valores antiguos de estos registros en la pila, como haría una rutina ordinaria, porque la causa de la excepción podría haber sido una referencia a memoria que utilizó un valor erróneo (como 0) en el puntero de pila. En lugar de ello, el manejador de excepciones almacena estos registros en un registro manejador de excepciones (`$k1`, ya que no puede acceder a memoria sin utilizar `$at`) y dos posiciones de memoria (`save0` y `save1`). Si se pudiese interrumpir la rutina de excepción, no serían suficiente dos posiciones de memoria ya que desde la segunda excepción se sobreescribirían los valores guardados durante la primera excepción. Sin embargo, este simple manejador de excepciones acaba su ejecución antes de que permita las interrupciones, por lo que el problema no se plantea.

```
.ktext 0x80000180
mov   $k1, $at      # Salva el registro $at
sw    $a0, save0    # El manejador no es reentrante y no
sw    $a1, save1    # puede usar la pila para salvar $a0, $a1
                    # No necesita salvar $k0/$k1
```

El manejador de excepciones a continuación mueve los registros Cause y EPC a registros de la CPU. Los registros Cause y EPC no son parte del banco de registros de la CPU. En lugar de ello, son registros del coprocesador 0, que es la parte de la CPU que maneja las excepciones. La instrucción `mfc0 $k0, $13` mueve el registro 13 del coprocesador 0 (el registro Cause) al registro `$k0` de la CPU. Observe que el manejador de excepciones no necesita salvar los registros `$k0` y `$k1`, porque supone que los programas de usuario no usan estos registros. El manejador de excepciones utiliza el valor del registro Cause para comprobar si la excepción fue causada por una interrupción (véase la tabla anterior). Si es así, la excepción se ignora. Si la excepción no es una interrupción, el manejador llama a `print_excp` para imprimir un mensaje.

```
mfc0   $k0, $13      # Mueve Cause a $k0

srl    $a0, $k0,2    # Extrae el campo ExcCode
andi   $a0, $a0,0xf

bgtz   $a0, done     # Salta si ExcCode es Int (0)

mov    $a0, $k0      # Mueve Cause a $a0
mfc0   $a1, $14      # Mueve EPC a $a1
jal    print_excp    # Print exception error message
```

Antes del retorno, el manejador de excepciones borra el registro Cause; restablece el registro de Estado para permitir las interrupciones y la borra el bit EXL, que permite que excepciones posteriores cambien el registro EPC, y restaura los registros `$a0`, `$a1` y `$at`. A continuación, ejecuta la instrucción `eret` (retorno de excepción), lo que retorna a la instrucción apuntada por EPC. Este manejador de excepciones retorna a la instrucción que sigue a la que causó la excepción, a fin de no volver a ejecutar la instrucción que causó la excepción y que causaría misma excepción de nuevo.

```
done:       mfc0 $k0, $14     # Liquida EPC
            addiu $k0, $k0, 4 # No reejecuta la instrucción
                              # que causó la excepción
            mtc0 $k0, $14     # EPC

            mtc0 $0, $13      # Borra el registro Cause

            mfc0 $k0, $12     # Prepara el registro de Estado
            andi $k0, 0xfffd  # Borra el bit EXL
            ori  $k0, 0x1     # Capacita interrupciones
            mtc0 $k0, $12

            lw   $a0, save0   # Restaura registros
            lw   $a1, save1
            mov  $at, $k1

            eret              # Retorna a EPC

            .kdata
save0:      .word 0
save1:      .word 0
```

Extensión: En procesadores MIPS reales, el retorno de un manejador de excepciones es más complejo. El manejador de excepciones no siempre puede saltar a la instrucción siguiente al `EPC`. Por ejemplo, si la instrucción que causó la excepción estaba en una ranura de retardo de una instrucción salto condicional (véase capítulo 4), la siguiente instrucción a ejecutar no puede ser la siguiente instrucción en la memoria.

B.8 Entrada y salida

SPIM simula un dispositivo de E / S: una consola proyectada en memoria (*memory- mapped*) en la que un programa puede leer y escribir caracteres. Cuando un programa está ejecutándose, SPIM conecta su propio terminal (o una ventana de consola separada en la versión X-window `xspim` o la versión para Windows `PCSpim`) al procesador. Un programa MIPS que se está ejecutando en SPIM puede leer los caracteres que el usuario escribe en el teclado. Además, si el programa MIPS escribe caracteres en el terminal, aparecen en el terminal de SPIM o en la ventana de consola. Una excepción a esta regla es el control-C: este carácter no se pasa al programa, sino que hace que SPIM se detenga y vuelva al modo comando. Cuando el programa detiene la ejecución (por ejemplo, porque el ususario escribió control-C, o porque el programa alcanza un punto de parada (*breakpoint*)), el terminal se reconecta a SPIM para que pueda teclear comandos SPIM.

Para utilizar la E / S proyectada en memoria (véase más adelante), `spim` o `xspim` deben iniciarse con la opción `-mapped_io`. `PCSpim` puede permitir E / S proyectada en memoria con una opción de línea de comandos o el diálogo de "Configuración".

El dispositivo terminal consta de dos unidades independientes: un *receptor* y un *transmisor*. El receptor lee caracteres escritos con el teclado. El transmisor escribe caracteres en la consola. Las dos unidades son completamente independientes. Esto significa, por ejemplo, que los caracteres escritos en el teclado no se repiten de forma automática en la pantalla. En lugar de ello, un programa difunde un carácter leyéndolo del receptor y escribiéndolo en el transmisor.

Un programa controla el terminal con cuatro registros de dispositivo proyectados en memoria, tal y como se muestra en la figura B.8.1. "Memoria proyectada" significa que cada registro aparece como una posición de memoria especial. El *Registro de Control del Receptor* se encuentra en la posición $ffff0000_{hex}$. Realmente solo se utilizan dos de sus bits. El Bit 0 se llama "preparado": si es 1, significa que un carácter ha llegado desde el teclado, pero todavía no se ha leído del registro de Datos del Receptor. El bit "preparado" es de sólo lectura: las escrituras sobre él se ignoran. El bit "preparado" cambia de 0 a 1 cuando se escribe en el teclado un carácter, y cambia de 1 a 0 cuando el carácter se lee desde el registro de Datos del Receptor.

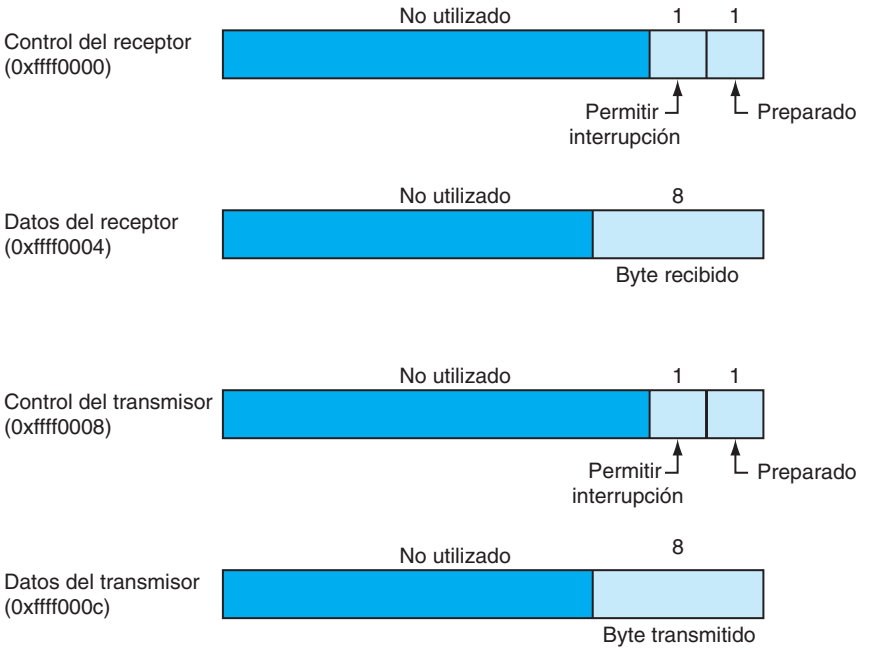


FIGURA B.8.1 El terminal se controla con cuatro registros de dispositivo, cada uno de los cuales se presenta como una posición de memoria en una dirección dada. En realidad, sólo se utilizan pocos bits de estos registros. Los demás están siempre a 0 al leer y se ignoran al escribir.

El bit 1 del registro de Control del Receptor es el "permitir interrupción (*interrupt enable*)" del teclado. Este bit puede ser leído y escrito por un programa. Inicialmente vale 0. Si se activa a 1 por un programa, el terminal pide una interrupción hardware de nivel 1 cuando se escribe un carácter con el teclado y el bit 0 que indica listo o "preparado" se pone a 1. Sin embargo, para que la interrupción afecte al procesador, las interrupciones también deben estar habilitadas en el registro de Estado (véase sección B.7). Los restantes bits del registro de Control del Receptor no se utilizan.

El segundo registro de dispositivo del terminal es el *Registro de Datos del Receptor* (en la dirección $ffff0004_{hex}$). Los 8 bits menos significativos de este registro contienen el último carácter escrito con el teclado. Los demás bits contienen 0s. Este registro es de sólo lectura y cambia sólo cuando se escribe un nuevo carácter con el teclado. La lectura del registro de Datos del Receptor reinicia a 0 el bit de preparado en el registro de Control del Receptor. El valor en este registro no está definido si el registro de Control del Receptor es 0.

El tercer registro de dispositivo del terminal es el *Registro de Control del Transmisor* (en la dirección $ffff0008_{hex}$). Sólo se utilizan los 2 bits de menos significativos de este registro. Se comportan de manera similar a los bits correspondientes del registro de Control del Receptor. El bit 0 se llama "preparado" y es de sólo lectura. Si este bit es 1, el transmisor está preparado para aceptar un nuevo carácter

de salida. Si es 0, el transmisor está todavía ocupado escribiendo el carácter anterior. El bit 1 es "permitir interrupción" y se puede leer y escribir. Si este bit está a 1, entonces el terminal pedirá una interrupción hardware de nivel 0 cuando el transmisor esté listo para un nuevo carácter y el bit de preparado se ponga a 1.

El último registro de dispositivo es el registro de Datos del Transmisor (en la dirección $ffff000c_{hex}$). Cuando un valor se escribe en esta posición, sus 8 bits de menos significativos (es decir, un carácter ASCII como en la figura 2.15 en el capítulo 2) se envían a la consola. Cuando se escribe el registro de Datos del Transmisor, el bit preparado en el registro de Control del Transmisor se reinicia a 0. Este bit permanece a 0 hasta que haya transcurrido el tiempo suficiente para transmitir el carácter al terminal, entonces el bit de preparado se pone en 1 de nuevo. El registro de Datos del Transmisor se debe escribir sólo cuando el bit de preparado del registro de Control del Transmisor es 1. Si el transmisor no está preparado, las escrituras en el registro de Datos del Transmisor se ignoran (la escritura parece tener éxito, pero el carácter no sale en la pantalla).

Los computadores reales necesitan tiempo para enviar caracteres a una consola o terminal. SPIM simula estos retardos. Por ejemplo, después de que el transmisor comienza a escribir un carácter, el bit de preparado del transmisor se pone a 0 durante un tiempo. SPIM mide el tiempo en instrucciones ejecutadas, no en tiempo real del reloj. Esto significa que el transmisor no vuelve a estar preparado de nuevo hasta que el procesador ejecuta un número fijo de instrucciones. Si se para la máquina y se observa el bit de preparado, este no cambiará. Sin embargo, si se deja que la máquina ejecute, el bit volverá a 1 al poco rato.

B.9 SPIM

SPIM es un simulador software que ejecuta programas en lenguaje ensamblador para procesadores que implementan la arquitectura MIPS32, específicamente la revisión 1 de esta arquitectura con un mapa de memoria fijo, sin caches, y únicamente los coprocesadores 0 y 1.[6] El nombre SPIM es justamente MIPS escrito al revés. SPIM puede leer y ejecutar inmediatamente archivos de lenguaje ensamblador. SPIM es un sistema autocontenido para ejecutar programas MIPS. Contiene un depurador y proporciona algunos servicios similares a los de un sistema operativo. SPIM es mucho más lento que un computador real (100 veces o más).

6. Las versiones anteriores de SPIM (antes de 7.0) implementan la arquitectura MIPS-I usada en los procesadores MIPS R2000 originales. Esta arquitectura es casi un subconjunto propio de la arquitectura MIPS32, cuya diferencia está en cómo se manipulan las excepciones. MIPS32 también introdujo aproximadamente 60 nuevas instrucciones, las cuales están soportadas en SPIM. Los programas que se ejecutaban en las versiones anteriores de SPIM y no usaban excepciones deberían ejecutarse sin modificación en las nuevas versiones de SPIM. Los programas que usaban excepciones requerirán cambios menores.

Sin embargo, ¡su bajo coste y gran disponibilidad no puede ser igualada por el hardware real!

Una pregunta obvia es: ¿por qué usar un simulador cuando la mayoría de las personas disponen de PCs que contienen procesadores que ejecutan significativamente más rápido que SPIM? Una razón es que los procesadores en los PCs son Intel 80x86s, cuya arquitectura es mucho menos regular y mucho más compleja de comprender y programar que los procesadores MIPS. La arquitectura MIPS puede ser la personificación de una máquina RISC clara y simple.

Además, los simuladores puede proporcionar un entorno mejor para la programación en ensamblador que la máquina real, porque pueden detectar más errores y proporcionar una interfaz mejor que la máquina real.

Finalmente, los simuladores son herramientas útiles para estudiar los computadores y los programas que se ejecutan en ellos. Debido a que se implementan en software en lugar de en silicio, los simuladores se pueden examinar y modificar fácilmente para añadir nuevas instrucciones, construir nuevos sistemas tales como multiprocesadores, o simplemente para recoger datos.

Simulación de una máquina virtual

La arquitectura MIPS básica es difícil de programar directamente a causa de los saltos retardados, las cargas retardadas y los modos de direccionamiento restringidos. Esta dificultad es tolerable porque estos computadores fueron diseñados para ser programados mediante lenguajes de alto nivel y presentan una interfaz diseñada para compiladores en lugar de para programadores en lenguaje ensamblador. Una buena parte de la complejidad de programación surge de las instrucciones retardadas. Un *salto retardado* requiere dos ciclos para ejecutarse (véanse Extensiones de la páginas 343 y 381 del capítulo 4). En el segundo ciclo se ejecuta la instrucción que sigue inmediatamente al salto. Esta instrucción puede ejecutar trabajo útil que normalmente se habría hecho antes del salto. Puede ser también un `nop` (no operación) que no hace nada. Asimismo, las *cargas retardadas* requieren 2 ciclos para traer un valor de memoria, de manera que la instrucción que sigue inmediatamente a la carga no puede usar el valor (véase sección 4.2 del capítulo 4).

MIPS sabiamente elige ocultar esta complejidad haciendo que su ensamblador implemente una **máquina virtual**. Este computador virtual no tiene saltos ni cargas retardadas y tiene un repertorio de instrucciones más rico que el hardware real. El ensamblador reorganiza las instrucciones para rellenar los huecos de retardo. El computador virtual también proporciona *pseudoinstrucciones*, las cuales aparecen como instrucciones reales en los programas de lenguaje ensamblador. El hardware, sin embargo, no sabe nada sobre las pseudoinstrucciones, de manera que el ensamblador las traduce en secuencias equivalentes de instrucciones máquina reales. Por ejemplo, el hardware MIPS solo proporciona instrucciones para saltar cuando un registro es igual o diferente a 0. Otros saltos condicionales, como el que se produce cuando un registro es mayor que otro, se sintetizan comparando los dos registros y saltando cuando el resultado de la comparación es verdadero (o no nulo).

Máquina virtual: computador virtual que parece tener saltos y cargas no retardados y un repertorio de instrucciones más rico que el hardware real.

Por defecto, SPIM simula la rica máquina virtual, puesto que ésta es la máquina que muchos programadores encontrarán útil. Sin embargo, SPIM puede también simular los saltos y cargas retardadas en el hardware real. Más abajo, describimos la máquina virtual y solo mencionamos de pasada las características que no pertenecen al hardware real. De esta forma, seguimos la convención de los programadores (y compiladores) de lenguaje ensamblador MIPS, quienes rutinariamente usan la máquina extendida como si fuese implementada en silicio.

Comenzando con SPIM

El resto de este apéndice introduce SPIM y el lenguaje Ensamblador MIPS R2000. Muchos detalles nunca preocuparán al lector; sin embargo, el volumen total de información puede a veces oscurecer el hecho de que SPIM es un programa simple y fácil de usar. Esta sección comienza con un rápido tutorial sobre el uso de SPIM, el cual le debería capacitar para cargar, depurar y ejecutar programas MIPS simples.

SPIM viene en versiones diferentes para diferentes tipos de computadores. La única constante es la versión más simple, llamada `spim`, la cual es un programa dirigido desde la línea de comandos que se ejecuta en una ventana de consola. Opera como la mayoría de los programas de este tipo: se teclea una línea de texto, se pulsa la tecla de retorno, y `spim` ejecuta el comando. A pesar de que carece de una interfaz elaborada, `spim` puede hacer todo lo que sus primos más refinados pueden hacer.

Existen dos primos refinados de `spim`. La versión que se ejecuta en el entorno de ventanas X de un sistema Linux o UNIX se llama `xspim`. `xspim` es un programa más fácil de aprender y usar que `spim`, porque sus comandos son siempre visibles en la pantalla y porque continuamente muestra los registros y memoria de la máquina. La otra versión elaborada se llama `PCspim` y se ejecuta en el entorno de Microsoft Windows. Las versiones para UNIX y Windows de **SPIM** están en el CD (hacer clic sobre tutoriales). Los tutoriales de `xspim`, `pcSpim`, `spim` y las **opciones de la línea de comandos de spim** están en el CD (hacer clic en software).

Si se va a ejecutar SPIM en un PC con Microsoft Windows, es conveniente mirar primero el tutorial de **PCSpim** de este CD. Si se va a ejecutar SPIM en un computador con UNIX o Linux, se debería leer el tutorial de **xspim** (hacer clic en Tutoriales).

Características sorprendentes

A pesar de que SPIM simula fielmente un computador MIPS, es un simulador y ciertas cosas no son idénticas a un computador real. Las diferencias más obvias son que el cronometraje de las instrucciones y del sistema de memoria no son iguales. SPIM no simula caches ni latencias de memoria, ni refleja con precisión los retardos de las instrucciones de multiplicación y división o de las operaciones en punto flotante. Además, las instrucciones en punto flotante no detectan muchas condiciones de error, las cuales causarían excepciones en la máquina real.

Otra sorpresa (que también ocurre en la máquina real) es que una pseudoinstrucción se expande a varias instrucciones máquina. Cuando se ejecuta paso a paso o se examina la memoria, las instrucciones que se ven son diferentes de las del programa fuente. La correspondencia entre los dos conjuntos de instrucciones es bastante simple porque SPIM no reorganiza las instrucciones para rellenar los huecos de retardo.

Ordenación de los bytes

Los procesadores pueden numerar los bytes de una palabra de manera que el byte con el número menor sea tanto el que está más a la izquierda como el que está más a la derecha. La convención que usa una máquina se llama *orden de los bytes*. Los procesadores MIPS pueden operar con el orden de los bytes *big-endian* (acabamiento en mayor) o *little-endian* (acabamiento en menor). Por ejemplo, en una máquina *big-endian*, la directiva `.byte 0, 1, 2, 3` daría como resultado una palabra que contiene

Byte #			
0	1	2	3

mientras que en una máquina *little-endian*, la palabra contendría

Byte #			
3	2	1	0

SPIM opera con ambos órdenes de byte. El orden de byte de SPIM es el mismo que el orden de byte de la máquina subyacente que ejecuta el simulador. Por ejemplo, en un Intel 80x86, SPIM es *little-endian*, mientras que en un Macintosh o Sun SPARC, SPIM es *big-endian*.

Llamadas al sistema

SPIM proporciona un pequeño conjunto de servicios similares a los de un sistema operativo a través de la instrucciones de llamada al sistema (`syscall`). Para solicitar un servicio, un programa carga el código de la llamada al sistema (véase figura B.9.1) en el registro `$v0` y los argumentos en los registros `$a0-$a3` (o `$f12` para valores en punto flotante). Las llamadas a sistema que retornan valores ponen sus resultados en el registro `$v0` (o `$f0` para valores en punto flotante). Por ejemplo, el siguiente código imprime "`la respuesta = 5`":

```
        .data
str:
        .asciiz "la respuesta = "
        .text
```

Servicio	Código de la llamada al sistema	Argumentos	Resultado
print_int	1	$a0 = entero	
print_float	2	$f12 = punto flotante	
print_double	3	$f12 = punto flotante doble precisión	
print_string	4	$a0 = string	
read_int	5		entero (in $v0)
read_float	6		punto flotante (en $f0)
read_double	7		punto flotante doble precisión (en $f0)
read_string	8	$a0 = búfer, $a1 = longitud	
sbrk	9	$a0 = cantidad	dirección (en $v0)
exit	10		
print_char	11	$a0 = carácter	
read_char	12		carácter (en $a0)
open	13	$a0 = nombre de fichero string), $a1 = indicadores, $a2 = modo	descriptor de fichero (en $a0)
read	14	$a0 = descriptor de fichero, $a1 = búfer, $a2 = longitud	núm. caracteres leídos (en $a0)
write	15	$a0 = descriptor de fichero, $a1 = búfer, $a2 = longitud	núm. caracteres escritos (en $a0)
close	16	$a0 = descriptor de fichero	
exit2	17	$a0 = resultado	

FIGURA B.9.1 Servicios del sistema.

```
li       $v0, 4     # código de llamada al sistema para
                      print_str
la       $a0, str   # dirección del string a imprimir
syscall             # imprime el string

li       $v0, 1     # código de llamada al sistema para
                      print_int
li       $a0, 5     # entero a imprimir
syscall             # imprimirlo
```

A la llamada al sistema `print_int` se le pasa un entero y lo imprime en la consola. `print_float` imprime un número en punto flotante; `print_double` imprime un número en doble precisión; y a `print_string` se le pasa un puntero a una cadena acabada en nulo, que se escribe en la consola.

Las llamadas al sistema `read_int`, `read_float` y `read_double` leen una línea entera de la entrada hasta el retorno de carro, éste incluido. Los caracteres que siguen al número se ignoran. `read_string` tiene la misma semántica que la rutina de

biblioteca UNIX `fgets`. Lee hasta $n - 1$ caracteres en un búfer y termina la cadena con un byte nulo. Si en la línea actual hay menos de $n - 1$ caracteres, `read_string` lee hasta el retorno de carro, éste incluido, y de nuevo termina la cadena con nulo. *Alerta:* Los programas que utilizan estas llamadas al sistema para leer del terminal no deberían utilizar entrada/salida proyectada en memoria (véase sección B.8)

`sbrk` devuelve un puntero a un bloque de memoria que contiene *n* bytes adicionales. `exit` para el programa que ejecuta SPIM. `exit2` termina el programa SPIM, y el argumento de `exit2` es el valor retornado cuando el propio simulador SPIM finaliza.

`print_char` y `read_char` escriben y leen un carácter simple. `open`, `read`, `write` y `close` son las llamadas estándar de la biblioteca de UNIX.

B.10 Lenguaje ensamblador MIPS R2000

Un procesador MIPS está formado una unidad de procesamiento de enteros (la CPU) y una colección de coprocesadores que realizan tareas auxiliares u operan sobre otros tipos de datos tales como números en punto flotante (véase figura B.10.1). SPIM simula dos coprocesadores. El coprocesador 0 trata las excepciones e interrupciones. El coprocesador 1 es la unidad de punto flotante. SPIM simula la mayoría de los aspectos de esta unidad.

Modos de direccionamiento

MIPS es una arquitectura de carga-almacenamiento, lo que significa que sólo las instrucciones de carga y almacenamiento acceden a memoria. Las instrucciones de cómputo operan solamente sobre valores en registros. La máquina mínima proporciona un solo modo de direccionamiento: `c(rx)`, que usa como dirección la suma del valor inmediato `c` más el registro `rx`. La máquina virtual proporciona los siguientes modos de direccionamiento para las instrucciones de carga y almacenamiento:

Formato	Cálculo de la dirección
(registro)	contenido del registro
imm	inmediato
imm (registro)	inmediato + contenido del registro
etiqueta	dirección de etiqueta
etiqueta ± imm	dirección de etiqueta + o – inmediato
etiqueta ± imm (registro)	dirección de etiqueta + o – (inmediato + contenido del registro)

La mayoría de las instrucciones de carga y almacenamiento operan solamente sobre datos alineados. Una cantidad está *alineada* si su dirección de memoria es múltiplo de su tamaño en bytes. Por lo tanto, un objeto de media palabra debe estar alma-

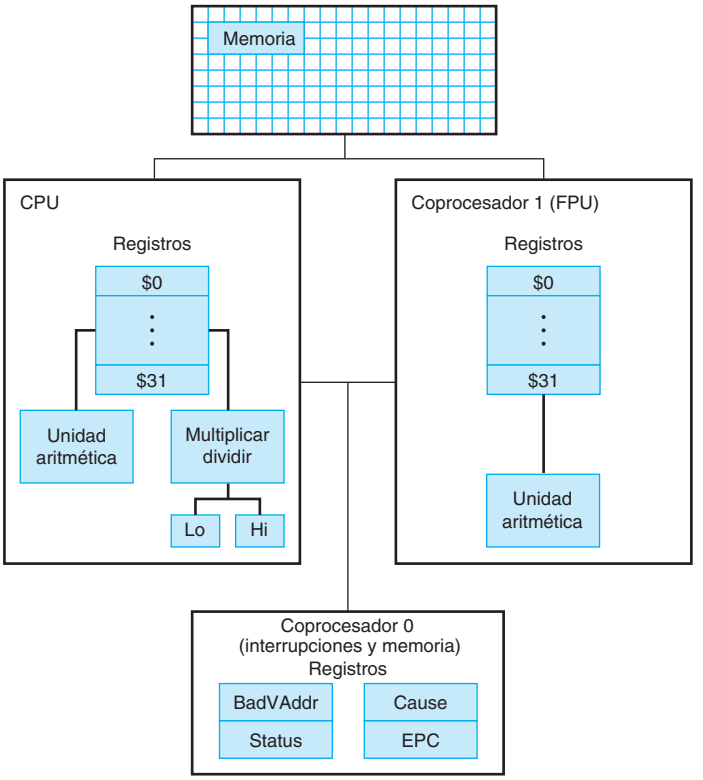


FIGURA B.10.1 CPU y FPU MIPS R2000.

cenado en direcciones pares y un objeto de palabra completa debe estar almacenado en direcciones que son múltiplos de 4. Sin embargo, MIPS proporciona algunas instrucciones para manipular datos desalineados (`lwl`, `lwr`, `swl` y `swr`).

Extensión: El ensamblador MIPS (y SPIM) sintetizan los modos de direccionamiento más complejos produciendo una o más instrucciones antes de la carga o almacenamiento para calcular la dirección compleja. Por ejemplo, suponga que la etiqueta `table` se refiere a la posición de memoria 0x10000004 y que un programa contiene la instrucción

```
ld $a0, table + 4($a1)
```

El ensamblador traduciría esta instrucción en las instrucciones

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

La primera instrucción carga los bits superiores de la dirección de la etiqueta en el registro `$at`, el cual es un registro que el ensamblador se reserva para su propio uso. La segunda instrucción suma el contenido del registro `$a1` a la dirección parcial de la etiqueta. Finalmente, la instrucción de carga usa el modo de direccionamiento del hardware para sumar la suma de los bits inferiores de la dirección de la etiqueta y el desplazamiento de la instrucción original con el valor en el registro `$at`.

Sintaxis del ensamblador

Los comentarios en los archivos en ensamblador empiezan con un signo de almohadilla (#). Cualquier cosa que esté a partir de este signo hasta el final de la línea se ignora.

Los identificadores son una secuencia de caracteres alfanuméricos, guiones bajos (_) y puntos (.) que no empiezan con un número. Los códigos de operación de las instrucciones son palabras reservadas que no se pueden usan como identificadores. Las etiquetas se declaran poniéndolas al principio de una línea seguidas de dos puntos, por ejemplo:

```
            .data
elemento:   .word 1
            .text
            .globl main       # Debe ser global
principal:  lw          $t0, elemento
```

Los números están en base 10 por defecto. Si están precedidos por *0x*, se interpretan como hexadecimal. Por ello, 256 y 0x100 denotan el mismo valor.

Las cadenas están encerradas entre comillas dobles ("). Los caracteres especiales en las cadenas siguen la convención de C:

- nueva línea `\n`
- tabulador `\t`
- comillas `\"`

SPIM soporta un subconjunto de las directivas de ensamblador de MIPS:

`.align n`	Alinea el próximo dato a un límite de 2^n. Por ejemplo, `.align2` alinea el próximo valor a un límite de palabra. La directiva `.align0` desactiva la alineación automática de las directivas `.half`, `.word`, `.float` y `.double` hasta la próximo directiva `.data` o `.kdata`.
`.ascii str`	Almacena la cadena *str* en memoria, pero no lo termina en nulo.
`.asciiz str`	Almacena la cadena *str* en memoria y lo termina en nulo.

`.byte b1,..., bn`	Almacena los *n* valores en bytes sucesivos de memoria.
`.data <addr>`	Los ítems subsiguientes se almacenan en el segmento de datos. Si el argumento opcional *addr* está presente, los ítems posteriores se almacenan comenzando en la dirección *addr*.
`.double d1, ..., dn`	Almacena los *n* números en punto flotante de doble precisión en posiciones de memoria sucesivas.
`.extern sym size`	Declara que el dato almacenado en *sym* es de tamaño de *size* bytes y es una etiqueta global. Esta directiva permite al ensamblador almacenar los datos en una porción del segmento de datos que se accede de manera eficiente a través del registro `$gp`.
`.float f1,..., fn`	Almacena los *n* números en punto flotante de simple precisión en posiciones de memoria sucesivas.
`.globl sym`	Declara que *sym* es una etiqueta global y se puede referenciar desde otros archivos.
`.half h1, ..., hn`	Almacena las *n* cantidades de 16-bit en medias palabras sucesivas de memoria.
`.kdata <addr>`	Los siguientes elementos de datos se almacenan en el segmento de datos del núcleo. Si el argumento opcional *addr* está presente, los elementos se almacenan comenzando en la dirección *addr*.
`.ktext <addr>`	Los siguientes ítems se colocan en el segmento de texto del núcleo. En SPIM, estos ítems sólo pueden ser instrucciones o palabras (véase directiva `.word` a continuación). Si el argumento opcional *addr* está presente, los ítems se almacenan comenzando en la dirección *addr*.
`.set noat` y `.set at`	La primera directiva impide que SPIM se queje de que las siguientes instrucciones usen el registro `$at`. La segunda directiva rehabilita la advertencia. Puesto que las pseudoinstrucciones se expanden en código que utiliza el registro `$at`, los programadores deben tener mucho cuidado al dejar valores en este registro.
`.space n`	Asigna *n* bytes de espacio en el segmento actual (que debe ser el segmento de datos en SPIM).

`.text <addr>`	Los siguientes ítems se colocan en el segmento de texto del usuario. En SPIM, estos ítems sólo pueden ser instrucciones o palabras (véase la directiva `.word` a continuación). Si el argumento opcional *addr* está presente, los siguientes ítems se almacenan comenzando en la dirección *addr*.
`.word w1,..., wn`	Almacena las *n* cantidades de 32 bits en palabras de memoria sucesivas.

SPIM no distingue las diversas partes del segmento de datos (`.data`, `.rdata` y `.sdata`).

Codificación de las instrucciones MIPS

La figura B.10.2 explica cómo se codifican las instrucciones MIPS en binario. Cada columna contiene la codificación para un campo (un grupo de bits contiguos) de la instrucción. Los números en el margen izquierdo son los valores de un campo. Por ejemplo, el código de operación `j` tiene un valor de 2 en el campo de código de operación. El texto en la parte superior de una columna nombra un campo y especifica qué bits ocupa en una instrucción. Por ejemplo, el campo `op` figura en los bits 26-31 de una instrucción. Este campo codifica la mayoría de las instrucciones. Sin embargo, algunos grupos de instrucciones usan campos adicionales para distinguir las instrucciones correspondientes. Por ejemplo, las diferentes instrucciones de punto flotante se especifican mediante los bits 0-5. Las flechas de la primera columna muestran qué códigos de operación utilizan estos campos adicionales.

Formato de la instrucción

El resto de este apéndice describe tanto las instrucciones MIPS reales ejecutadas por el hardware como las pseudoinstrucciones proporcionadas por el ensamblador MIPS. Los dos tipos de instrucciones son fácilmente distinguibles. Las instrucciones reales muestran sus campos en su representación binaria. Por ejemplo, en

Suma (con desbordamiento)

`add rd, rs, rt`

La instrucción `add` consta de seis campos. El tamaño de cada campo en bits es el número pequeño que se encuentra debajo del campo. Esta instrucción se inicia con 6 bits iguales a 0s. Los especificadores de registro comienzan con una *r*, de manera que el próximo campo es un especificador de registro de 5 bits llamado `rs`. Éste es el mismo registro del segundo argumento en el ensamblador simbólico a la izquierda de esta línea. Otro campo común es $\texttt{imm}_{16}$, que es un número inmediato de 16 bits.

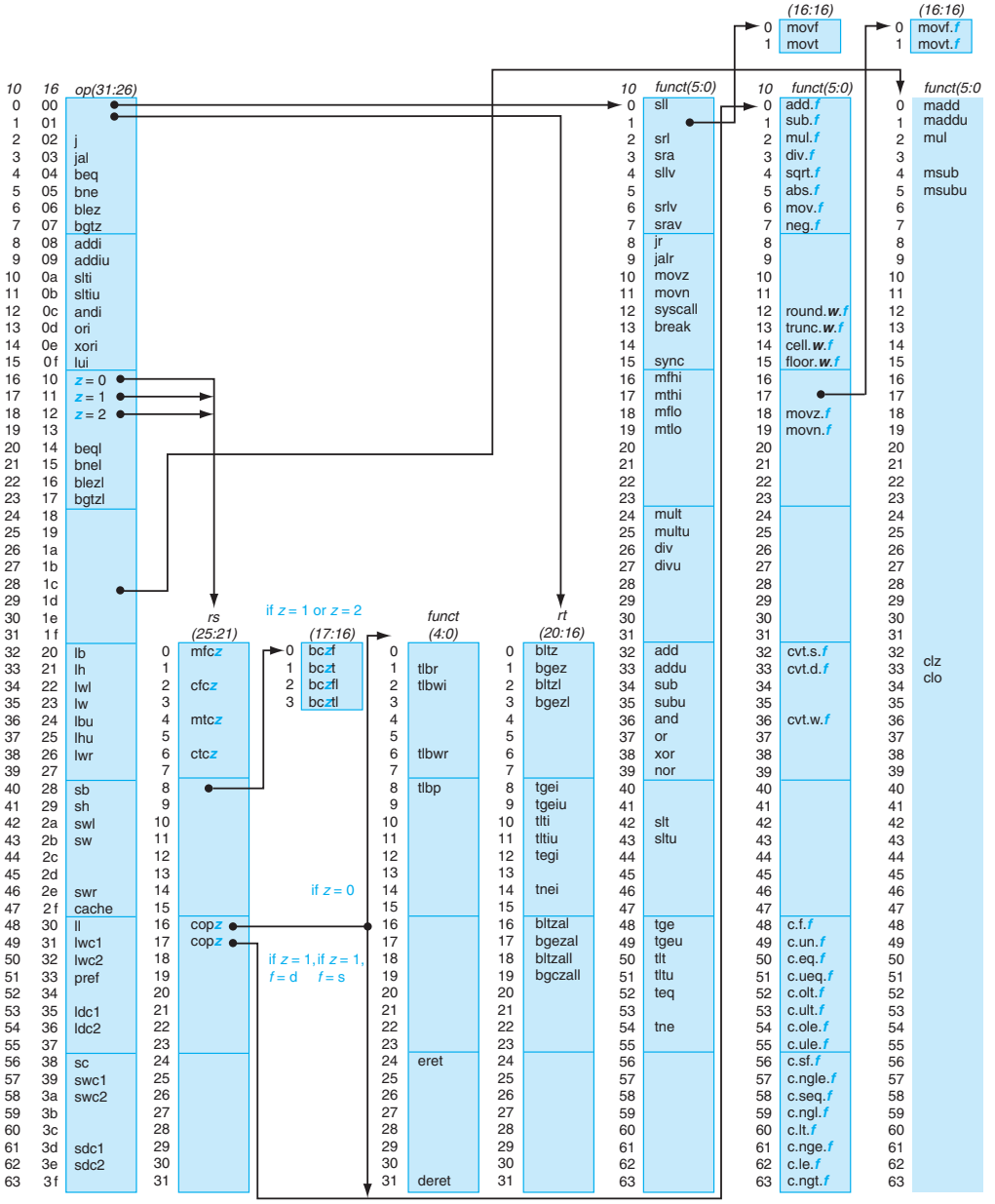


FIGURA B.10.2 Mapa de códigos de operación de MIPS. Los valores de cada campo se muestran a su izquierda. La primera columna muestra los valores en base 10, y la segunda muestra la base 16 para el campo de código de operación (bits 31 a 26), el cual se encuentra en la tercera columna. Este campo de operación especifica completamente la operación MIPS, salvo para 6 valores de código de operación: 0, 1, 16, 17, 18 y 19. Estas operaciones están determinadas por otros campos, identificados por punteros. El último campo (funct) utiliza "*f*" para indicar "s" si rs = 16 y op = 17 o "d" si rs = 17 y op = 17. El segundo campo (rs) utiliza "*z*" para indicar "0", "1", "2" o "3" si op = 16, 17, 18 o 19, respectivamente. Si rs = 16, la operación se especifica en otros lugares: si $z = 0$, las operaciones se especifican en el cuarto campo (bits 4 a 0); si $z = 1$, entonces las operaciones se encuentran en el último campo con f = s. Si rs= 17 y $z = 1$, entonces las operaciones se encuentran en el último campo con f = d.

Las pseudoinstrucciones siguen más o menos las mismas convenciones, pero omiten la información de codificación de la instrucción. Por ejemplo:

Multipicar (sin desbordamiento)

```
mul rdest, rsrc1, src2        pseudoinstrucción
```

En las pseudoinstrucciones, `rdest` y `rsrc1` son registros y `src2` es un registro o un valor inmediato. En general, el ensamblador y SPIM traducen un forma más general de una instrucción (por ejemplo, `add $v1, $a0, 0x55`) a una forma especializada (por ejemplo, `addi $v1, $a0, 0x55`).

Instrucciones aritméticas y lógicas

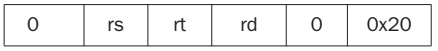
Valor absoluto

```
abs rdest, rsrc  pseudoinstrucción
```

Pone el valor absoluto del registro `rsrc` en el registro `rdest`.

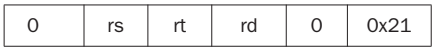
Suma (con desbordamiento)

```
add rd, rs, rt
```



Suma (sin desbordamiento)

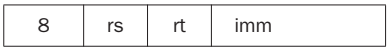
```
addu rd, rs, rt
```



La suma de los registros `rs` y `rt` se almacena en el registro `rd`.

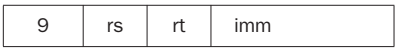
Suma inmediata (con desbordamiento)

```
addi rt, rs, imm
```



Suma inmediata (sin desbordamiento)

```
addiu rt, rs, imm
```



La suma del registro `rs` y el valor inmediato con extensión de signo se almacena en el registro `rt`.

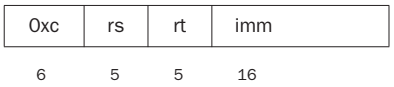
AND

```
and rd, rs, rt
```

La AND lógica de los registros `rs` y `rt` se almacena en el registro `rd`.

AND inmediata

```
andi rt, rs, imm
```



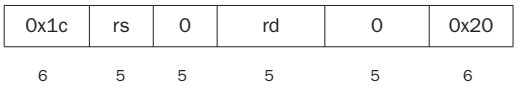
La AND lógica del registro `rs` y el valor inmediato con extensión a ceros se almacena en el registro `rt`.

Contar unos del principio

```
clo rd, rs
```

Contar ceros del principio

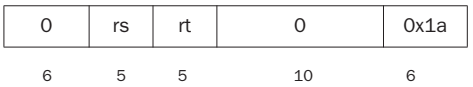
```
clz rd, rs
```



Cuenta el número de unos al principio (ceros) en la palabra del registro `rs` y pone el resultado en el registro `rd`. Si una palabra es todo unos (ceros), el resultado es 32.

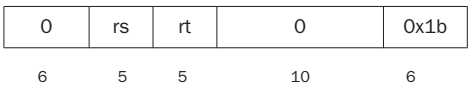
Dividir (con desbordamiento)

```
div rs, rt
```



Dividir (sin desbordamiento)

```
divu rs, rt
```



Divide el registro `rs` entre el registro `rt`. Deja el cociente en el registro `lo` y el resto en el registro `hi`. Observe que si un operando es negativo, el resto no está especificado en la arquitectura MIPS y depende de la convención de la máquina en la cual se está ejecutando SPIM.

Dividir (con desbordamiento)

```
div rdest, rsrc1, src2 pseudoinstrucción
```

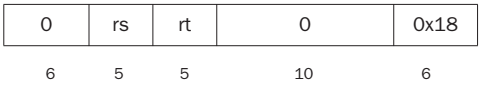
Dividir (sin desbordamiento)

```
divu rdest, rsrc1, src2 pseudoinstrucción
```

El cociente de la división entre el registro `rsrc1` y `src2` se almacena en el registro `rdest`.

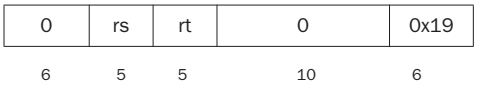
Multiplicar

`mult rs, rt`



Multiplicación sin signo

`multu rs, rt`



Multiplica los registros `rs` y `rt`. Deja la palabra de orden menor del producto en el registro `lo` y la palabra de orden mayor en el `hi`.

Multiplicar (sin desbordamiento)

`mul rd, rs, rt`

Los 32 bits de menor orden del producto de `rs` y `rt` se almacena en el registro `rd`.

Multiplicar (con desbordamiento)

```
mulo rdest, rsrc1, src2 pseudoinstrucción
```

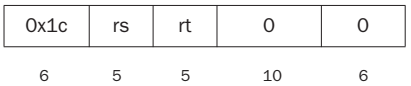
Multiplicación sin signo (con desbordamiento)

```
mulou rdest, rsrc1, src2 pseudoinstrucción
```

Los 32 bits de menor orden del producto de los registros `rsrc1` y `src2` se almacena en el registro `rdest`.

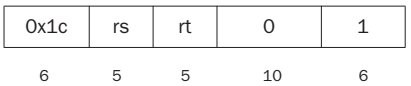
Multiplicación y suma

```
madd rs, rt
```



Multiplicación y suma sin signo

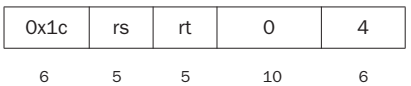
```
maddu rs, rt
```



Multiplica los registros `rs` y `rt` y suma el producto resultante de 64-bit al valor de 64-bit de los registros `lo` y `hi` concatenados.

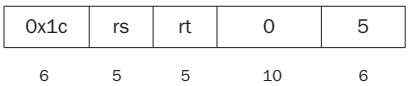
Multiplicación y resta

```
msub rs, rt
```



Multiplicación y resta sin signo

```
msub rs, rt
```



Multiplica los registros `rs` y `rt` y resta el resultado de 64-bit del valor de 64-bit de los registros `lo` y `hi` concatenados.

Negar valor (con desbordamiento)

```
neg rdest, rsrc pseudoinstrucción
```

Negar valor (sin desbordamiento)

```
negu rdest, rsrc pseudoinstrucción
```

El negativo del registro `rsrc` se almacena en el registro `rdest`.

NOR

```
nor rd, rs, rt
```

La NOR lógica de los registros `rs` y `rt` se almacena en el registro `rd`.

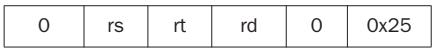
NOT

```
not rdest, rsrc          pseudoinstrucción
```

La negación lógica bit a bit del registro `rsrc` se almacena en el registro `rdest`.

OR

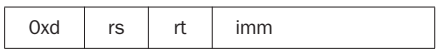
```
or rd, rs, rt
```



La OR lógica de los registros `rs` y `rt` se almacena en el registro `rd`.

OR inmediata

```
ori rd, rs, imm
```



La OR lógica del registro `rs` y el valor inmediato extendido a ceros se almacena en el registro `rt`.

Resto

```
rem rdest, rsrc1, rsrc2      pseudoinstrucción
```

Resto sin signo

```
remu rdest, rsrc1, rsrc2     pseudoinstrucción
```

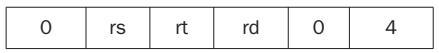
El resto del registro `rsrc1` dividido por el registro `rsrc2` en el registro `rdest`. Observe que si un operando es negativo, el resto no está especificado en la arquitectura MIPS y depende de la convención de la máquina en la cual se está ejecutando SPIM.

Desplazamiento lógico a la izquierda

```
sll rd, rt, shamt
```

Desplazamiento lógico variable a la izquierda

```
sllv rd, rt, rs
```



Desplazamiento aritmético a la derecha

```
sra rd, rt, shamt
```

Desplazamiento aritmético variable a la derecha

```
srav rd, rt, rs
```

Desplazamiento lógico a la derecha

```
srl rd, rt, shamt
```

Desplazamiento lógico variable a la derecha

```
srlv rd, rt, rs
```

Desplaza el registro `rt` a la izquierda (derecha) las posiciones indicadas por el valor inmediato `shamt` o el registro `rs` y pone el resultado en el registro `rd`. Observe que el argumento `rs` se ignora para `sll`, `sra` y `srl`.

Rotar a la izquierda

```
rol rdest, rsrc1, rsrc2
```
pseudoinstrucción

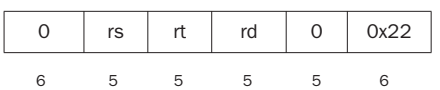
Rotar a la derecha

```
ror rdest, rsrc1, rsrc2
```
pseudoinstrucción

Rota el registro `rsrc1` a la izquierda (derecha) las posiciones indicadas por `rsrc2` y pone el resultado en el registro `rdest`.

Resta (con desbordamiento)

```
sub rd, rs, rt
```



Resta (sin desbordamiento)

```
subu rd, rs, rt
```

La diferencia de los registros `rs` y `rt` se almacena en el registro `rd`.

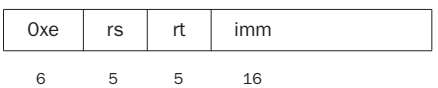
OR exclusiva

```
xor rd, rs, rt
```

La XOR lógica de los registros `rs` y `rt` se almacena en el registro `rd`.

XOR inmediata

```
xori rt, rs, imm
```

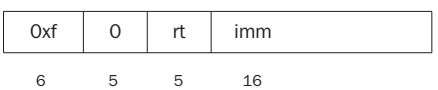


La XOR lógica del registro `rs` y el valor inmediato extendido a ceros se almacena en el registro `rt`.

Instrucciones de manipulación de constantes

Carga inmediato superior

```
lui rt, imm
```



Carga la media palabra más baja del valor inmediato `imm` en la media palabra superior del registro `rt`. Los bits más bajos del registro se colocan a 0.

Carga inmediato

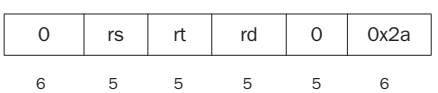
```
li rdest, imm                    pseudoinstrucción
```

Mueve el valor inmediato `imm` al registro `rdest`.

Instrucciones de comparación

Establecer menor que

```
slt rd, rs, rt
```



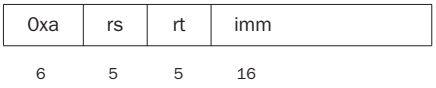
Establecer menor que sin signo

```
sltu rd, rs, rt
```

Pone el registro `rd` a 1 si el registro `rs` es menor que `rt`, y a 0 en caso contrario.

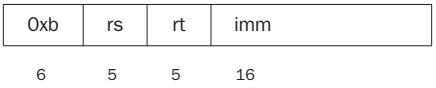
Establecer menor que inmediato

```
slti rt, rs. imm
```



Establecer menor que sin signo inmediato

```
sltiu rt, rs. imm
```



Pone el registro `rt` a 1 si el registro `rs` es menor que el valor inmediato con extensión de signo, y a 0 en caso contrario.

Establece igual

```
seq rdest, rsrc1, rsrc2
```

pseudoinstrucción

Pone el registro `rdest` a 1 si el registro `rsrc1` es igual a `rsrc2`, y a 0 en caso contrario.

Establecer mayor que o igual

```
sge rdest, rsrc1, rsrc2
```

pseudoinstrucción

Establecer mayor que igual sin signo

```
sgeu rdest, rsrc1, rsrc2
```

pseudoinstrucción

Pone el registro `rdest` a 1 si el registro `rsrc1` es mayor que o igual a `rsrc2`, y a 0 en caso contrario.

Establecer mayor que

```
sgt rdest, rsrc1, rsrc2
```

pseudoinstrucción

Establecer mayor que sin signo

```
sgtu rdest, rsrc1, rsrc2     pseudoinstrucción
```

Pone el registro `rdest` a 1 si el registro `rsrc1` es mayor que `rsrc2`, y a 0 en caso contrario.

Establecer menor o igual

```
sle rdest, rsrc1, rsrc2      pseudoinstrucción
```

Establecer menor o igual sin signo

```
sleu rdest, rsrc1, rsrc2     pseudoinstrucción
```

Pone el registro `rdest` a 1 si el registro `rsrc1` es menor que o igual a `rsrc2`, y a 0 en caso contrario.

Establecer no igual

```
sne rdest, rsrc1, rsrc2      pseudoinstrucción
```

Pone el registro `rdest` a 1 si el registro `rsrc1` no es igual a `rsrc2`, y a 0 en caso contrario.

Instrucciones de salto

Las instrucciones de salto usan un campo de desplazamiento de la instrucción de 16 bit con signo; por ello pueden saltar $2^{15} - 1$ *instrucciones* (no bytes) hacia adelante o 2^{15} instrucciones hacia atrás. La instrucción *jump* tiene un campo de dirección de 26 bits. En los procesadores MIPS reales, las instrucciones de salto son saltos retardados, los cuales no transfieren el control hasta que la instrucción que sigue al salto (su "ranura de retardo") se ha ejecutado (véase capítulo 4). Los saltos retardados afectan el cálculo del desplazamiento, puesto que debe ser calculado relativo a la dirección de la instrucción de la ranura de retardo (PC + 4), que es cuando el salto se produce. SPIM no simula esta ranura de retardo, a no ser que se especifiquen los indicadores `-bare` o `-delayed_branch`.

En código ensamblador, los desplazamientos no se suelen especificar con números. En su lugar, una instrucción salta a una etiqueta, y el ensamblador calcula la distancia entre las instrucciones de salto y la de destino.

En MIPS32, todos las instrucciones reales (no pseudo) de salto condicional tienen una variante "probable" (por ejemplo, la variante probable de `beq` es `beql`), que *no* ejecutan la instrucción que se encuentra en la ranura de retardo del salto si el salto no se toma. No se deben usar estas instrucciones, puede que se eliminen en futuras versiones de la arquitectura. SPIM implementa estas instrucciones, pero no se describirán de aquí en adelante.

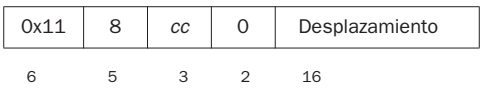
Instrucción de bifurcación

```
b etiqueta                    pseudoinstrucción
```

Bifurcación incondicional a la instrucción de la etiqueta.

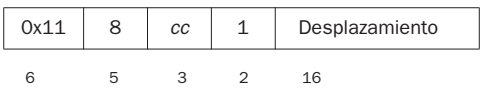
Bifurcar coprocesador falso

```
bclf cc etiqueta
```



Bifurcar coprocesador cierto

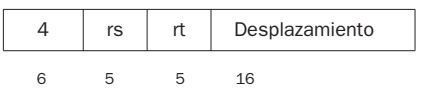
```
bclt cc etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el indicador de condición *cc* del coprocesador en punto flotante es falso (cierto). Si se omite *cc* en la instrucción, se asume que la etiqueta del código de condición es 0.

Bifurcar si igual

```
beq rs, rt, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es igual a `rt`.

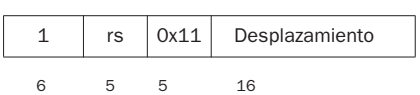
Bifurcar si mayor o igual a cero

```
bgez rs, etiqueta
```

Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es mayor que o igual a 0.

Bifurcar si mayor o igual que cero y enlazar

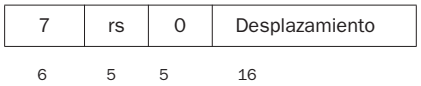
```
bgezal rs, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por el desplazamiento si el registro `rs` es mayor que o igual a 0. Guarda la dirección de la siguiente instrucción en el registro 31.

Bifurcar si mayor que cero

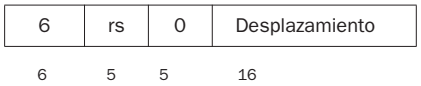
```
bgtz rs, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es mayor que 0.

Bifurcar si menor o igual a cero

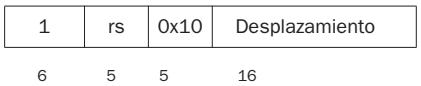
```
blez rs, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es menor que o igual a 0.

Bifurcar si menor que y enlazar

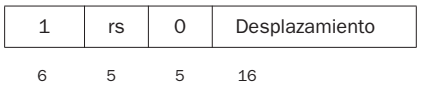
```
bltzal rs, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es menor que 0. Guarda la dirección de la siguiente instrucción en el registro 31.

Bifurcar si menor que cero r

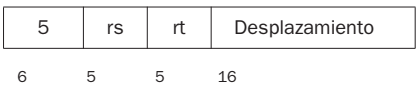
```
bltz rs, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` es menor que 0.

Bifurcar si no igual

```
bne rs, rt, etiqueta
```



Bifurca condicionalmente el número de instrucciones especificadas por desplazamiento si el registro `rs` no es igual a `rt`.

Bifurcar si igual cero

```
beqz rsrc, etiqueta          pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si `rsrc` es igual a 0.

Bifurcar si mayor o igual

```
bge rsrc1, rsrc2, etiqueta  pseudoinstrucción
```

Bifurcar si mayor o igual, sin signo

```
bgeu rsrc1, rsrc2, etiqueta pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si el registro `rsrc1` es mayor o igual que `rsrc2`.

Bifurcar si mayor que

```
bgt rsrc1, src2, etiqueta   pseudoinstrucción
```

Bifurcar si mayor que, sin signo

```
bgtu rsrc1, src2, etiqueta  pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si el registro `rsrc1` es mayor que `rsrc2`.

Bifurcar si menor o igual

```
ble rsrc1, src2, etiqueta   pseudoinstrucción
```

Bifurcar si menor o igual, sin signo

```
bleu rsrc1, src2, etiqueta  pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si el registro `rsrc1` es menor que o igual a `rsrc2`.

Bifurcar si menor que

```
blt rsrc1, src2, etiqueta   pseudoinstrucción
```

Bifurcar si menor que, sin signo

```
bltu rsrc1, src2, etiqueta  pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si el registro `rsrc1` es menor que `rsrc2`.

Bifurcar si no igual a cero

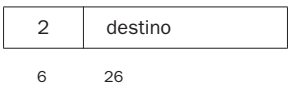
```
bnez rsrc, etiqueta          pseudoinstrucción
```

Bifurca condicionalmente a la instrucción de la etiqueta si el registro `rsrc` no es igual a 0.

Instrucciones de salto incondicional (*Jump*)

Saltar

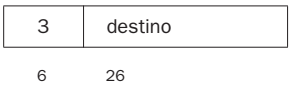
```
j destino
```



Salta incondicionalmente a la instrucción destino.

Salto y enlace

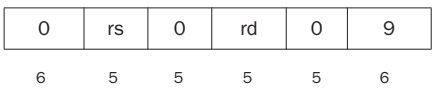
```
jal destino
```



Salta incondicionalmente a la instrucción destino. Salva la dirección de la siguiente instrucción en el registro `$ra`

Salto a registro y enlace

```
jalr rs, rd
```



Salta incondicionalmente a la instrucción cuya dirección se encuentra en el registro `rs`. Salva la dirección de la siguiente instrucción en el registro `$rd` (que por defecto es 31).

Salto a registro

```
jr rs
```

Salta incondicionalmente a la instrucción cuya dirección se encuentra en el registro `rs`.

Instrucciones de generación de interrupción/excepción de tipo *Trap*

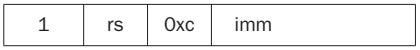
Generar Trap si igual

```
teq rs, rt
```

Si el registro `rs` es igual al registro `rt`, genera una excepción de tipo Trap.

Generar Trap si igual, inmediato

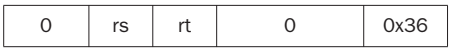
```
teqi rs, imm
```



Si el registro `rs` es igual al valor con signo extendido de `inm`, genera una excepción de tipo Trap.

Generar Trap si no igual

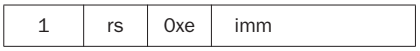
```
teq rs, rt
```



Si el registro `rs` no es igual al registro `rt`, genera una excepción de tipo Trap.

Generar Trap si no igual, inmediato

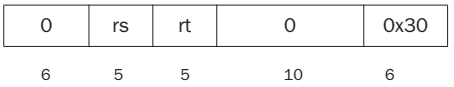
```
teqi rs, imm
```



Si el registro `rs` no es igual al valor con signo extendido de `inm`, genera una excepción de tipo Trap.

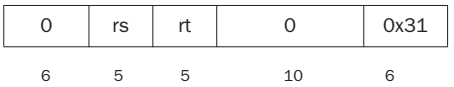
Generar Trap si mayor o igual

```
tge rs, rt
```



Generar Trap si mayor o igual, sin signo

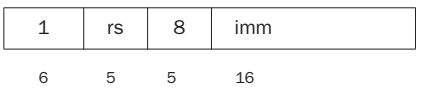
```
tgeu rs, rt
```



Si el registro `rs` es mayor o igual al registro `rt`, genera una excepción de tipo Trap.

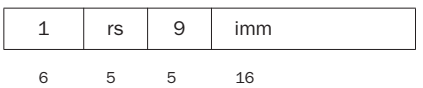
Generar Trap si mayor o igual, inmediato

```
tgei rs, imm
```



Generar Trap si mayor o igual, sin signo, inmediato

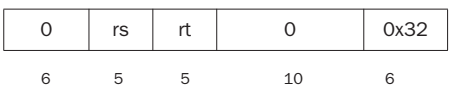
```
tgeiu rs, imm
```



Si el registro `rs` es mayor o igual al valor con signo extendido de `inm`, genera una excepción de tipo Trap.

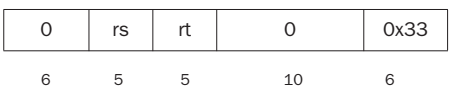
Generar Trap si menor que

```
tlt rs, rt
```



Generar Trap si menor que, sin signo

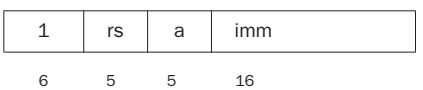
```
tltu rs, rt
```



Si el registro `rs` es menor que el registro `rt`, genera una excepción de tipo Trap.

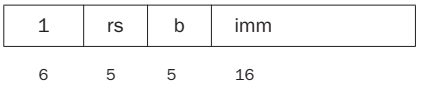
Generar Trap si menor que, inmediato

```
tlti rs, imm
```



Generar Trap si menor que, sin signo, inmediato

```
tltiu rs, imm
```



Si el registro `rs` es menor que el valor con signo extendido de `inm`, genera una excepción de tipo Trap.

Instrucciones de carga

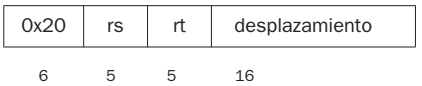
Cargar dirección

```
la rdest, dirección          pseudoinstrucción
```

Carga la *dirección* calculada / efectiva —no el contenido de la posición— dentro del registro `rdest`.

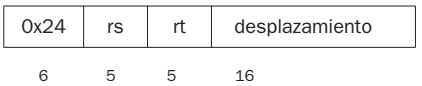
Cargar byte

```
lb rt, dirección
```



Cargar byte sin signo

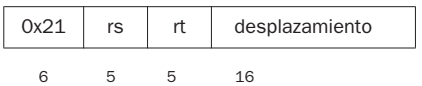
```
lbu rt, dirección
```



Carga el byte que está en *dirección* dentro del registro `rt`. Se extiende el signo del byte en el caso de la instrucción `lb`, pero no con la instrucción `lbu`.

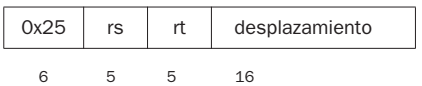
Cargar media palabra

```
lh rt, dirección
```



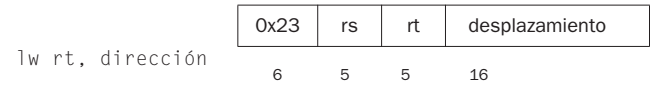
Cargar media palabra, sin signo

```
lhu rt, dirección
```



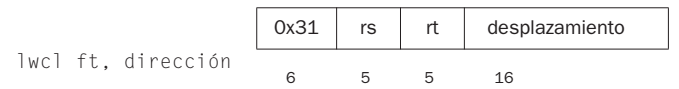
Carga el valor de 16 bits (media palabra) que está en *dirección* dentro del registro `rt`. Se extiende el signo de la media palabra en el caso de la instrucción `lh`, pero no con la instrucción `lhu`.

Cargar palabra



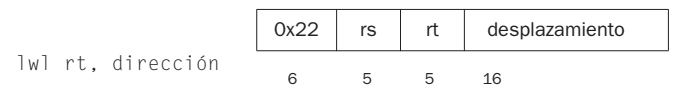
Carga el valor de 32 bits (palabra) que está en *dirección* dentro del registro `rt`.

Cargar palabra del coprocesador 1

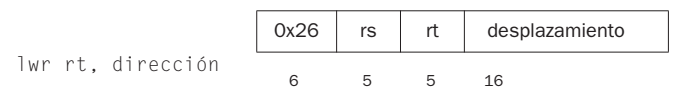


Carga el valor de 32 bits (palabra) que está en *dirección* dentro del registro `ft` de la unidad de cómputo en punto flotante.

Cargar palabra izquierda



Cargar palabra derecha



Carga los bytes de la izquierda (derecha) de la palabra que está en *dirección*, posiblemente desalineada, dentro del registro `rt`.

Cargar palabra doble

```
ld rdest, dirección          pseudoinstrucción
```

Carga el valor de 64 bits (doble palabra) que está en *dirección* dentro de los registros `rdest` y `rdest+1`.

Cargar media palabra, desalineada

```
ulh rdest, dirección         pseudoinstrucción
```

Cargar media palabra, desalineada, sin signo

```
ulhu rdest, dirección        pseudoinstrucción
```

Carga el valor de 16 bits (media palabra) que está en *dirección*, posiblemente desalineada, dentro del registro `rdest`. Se extiende el signo de la media palabra en el caso de la instrucción `ulh`, pero no con la instrucción `ulhu`.

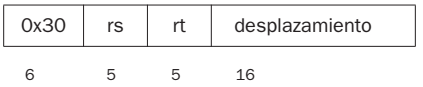
Cargar palabra, desalineada

```
ulw rdest, dirección          pseudoinstrucción
```

Carga el valor de 32 bits (palabra) que está en *dirección*, posiblemente desalineada, dentro del registro `rdest`.

Carga enlazada

```
ll rt, dirección
```

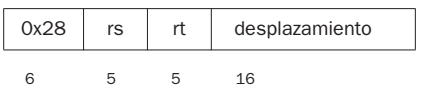


Carga el valor de 32 bits (palabra) que está en *dirección* dentro del registro `rt` y comienza una operación atómica de tipo lectura-modificación-escritura. Esta operación se completa con una instrucción de almacenamiento condicional (`sc`), que fallará si algún otro procesador escribe en el bloque que contiene la palabra cargada. Puesto que SPIM no simula múltiples procesadores, la instrucción de almacenamiento condicional siempre tendrá éxito.

Instrucciones de almacenamiento

Almacenar byte

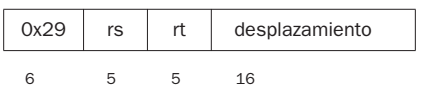
```
sb rt, dirección
```



Almacena en *dirección* el byte bajo (de menos peso) contenido en el registro `rt`.

Almacenar media palabra

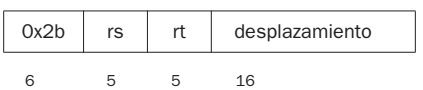
```
sh rt, dirección
```



Almacena en *dirección* la media palabra baja (de menos peso) contenida en el registro `rt`.

Almacenar palabra

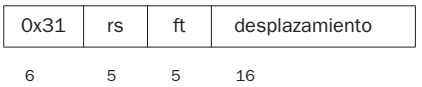
```
sw rt, dirección
```



Almacena en *dirección* la palabra contenida en el registro `rt`.

Almacenar palabra del coprocesador 1

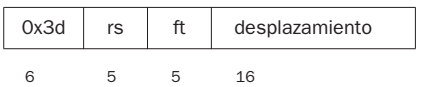
```
swcl ft, dirección
```



Almacena en *dirección* el valor punto flotante contenido en el registro `ft` del coprocesador de punto flotante.

Almacenar palabra doble del coprocesador 1

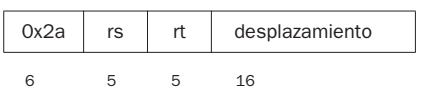
```
sdcl ft, dirección
```



Almacena en *dirección* el valor punto flotante de dos palabras contenido en los registros `ft` y `ft + 1` del coprocesador de punto flotante. El registro `ft` debe ser par.

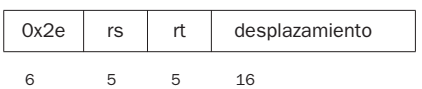
Almacenar palabra izquierda

```
swl rt, dirección
```



Almacenar palabra derecha

```
swr rt, dirección
```



Almacena en *dirección*, posiblemente desalineada, los bytes de la izquierda (derecha) contenidos en el registro `rt`.

Almacenar palabra doble

```
sd rsrc, dirección          pseudoinstrucción
```

Almacena en *dirección* la palabra doble de 64 bits contenida en los registros `rsrc` y `rsrc + 1`.

Almacenar media palabra, desalineada

```
ush rsrc, dirección          pseudoinstrucción
```

Almacena en *dirección*, posiblemente desalineada, la media palabra baja (de menos peso) contenida en el registro `rsrc`.

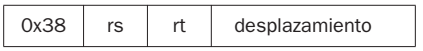
Almacenar palabra, desalineada

```
usw rsrc, dirección          pseudoinstrucción
```

Almacena en *dirección*, posiblemente desalineada, la palabra contenida en el registro `rsrc`.

Almacenamiento condicional

```
sc rt, dirección
```



Almacena en *dirección* la cantidad de 32 bits (palabra) que está dentro del registro `rt` y completa una operación atómica de tipo lectura-modificación-escritura. Si esta operación atómica tiene éxito, la palabra de memoria es modificada y el registro `rt` se establece con el valor 1. Si la operación atómica falla porque algún otro procesador escribió en el bloque que contiene la palabra cargada, esta instrucción no modificará la memoria y escribirá un 0 en el registro `rt`. Puesto que SPIM no simula múltiples procesadores, la instrucción de almacenamiento condicional siempre tendrá éxito.

Instrucciones de movimiento de datos

Mover

```
move rdest, rsrc             pseudoinstrucción
```

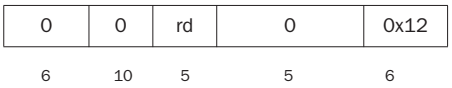
Mueve el contenido del registro `rsrc` a `rdest`.

Mover desde hi,

```
mfhi rd
```

Mover desde lo

```
mflo rd
```

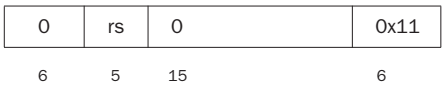


La unidad de multiplicación y división produce sus resultados en dos registros adicionales, `hi` y `lo`. Estas instrucciones mueven valores a y desde estos registros. Las pseudoinstrucciones de multiplicar, dividir y resto hacen que parezca que esta unidad opera directamente con registros generales, pero en realidad mueven el resultado después de que finaliza el cálculo.

Mueve el registro `hi` (`lo`) al registro `rd`.

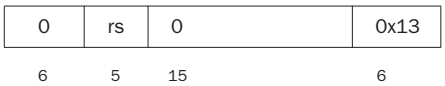
Mover a hi

```
mthi rs
```



Mover a lo

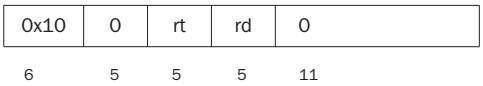
```
mtlo rs
```



Mueve el registro `rs` al registro `hi` (`lo`).

Mover desde coprocesador 0

```
mfc0 rt, rd
```



Mover desde coprocesador 1

```
mfc1 rt, fs
```

Los coprocesadores disponen de sus propios conjuntos de registros. Estas instrucciones mueven valores entre estos registros y los registros de la CPU.

Mueve el registro `rd` del coprocesador (el registro `fs` de la FPU) al registro `rt` de la CPU. La unidad de punto flotante (FPU) es el coprocesador 1.

Mover doble palabra desde coprocesador 1

```
mfc1.d rdest, frsrc1        pseudoinstrucción
```

Mueve los registros de punto-flotante `frsrc1` y `frsrc1` + `1` a los registros de la CPU `rdest` y `rdest` + `1`.

Mover a coprocesador 0

```
mtc0 rd, rt
```

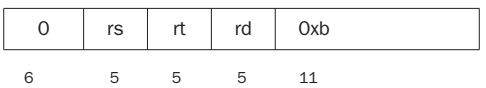
Mover a coprocesador 1

```
mtc1 rd, fs
```

Mueve el registro `rt` de la CPU al registro `rd` del coprocesador (registro `fs` en la FPU).

Mover condicional si no es cero

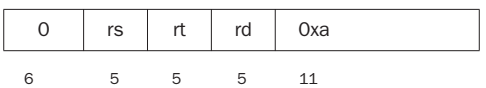
```
movn rd, rs, rt
```



Mueve el contenido del registro `rs` al registro `rd` si el registro `rt` no es 0.

Mover condicional si es cero

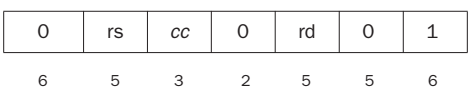
```
movz rd, rs, rt
```



Mueve el contenido del registro `rs` al registro `rd` si el registro `rt` es 0.

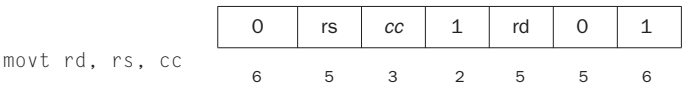
Mover condicional si FP es falso

```
movf rd, rs, cc
```



Mueve el registro `rs` de la CPU al registro `rd` si *cc* dentro del registro de códigos de condición de la FPU es 0. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Mover condicional si FP es cierto




```
movt rd, rs, cc
```

Mueve el registro `rs` de la CPU al registro `rd` si *cc* dentro del registro de códigos de condición de la FPU es 1. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

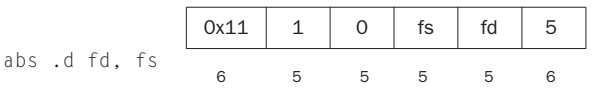
Instrucciones de punto flotante

El procesador MIPS tiene un coprocesador de punto flotante (numerado con el 1) que realiza operaciones con números punto flotante en precisión simple (32-bit) y en precisión doble (64-bit). Este coprocesador tiene sus propios registros, que están numerados `$f0`–`$f31`. Puesto que estos registros sólo tienen 32 bits de anchura, se necesitan dos para almacenar números dobles, de forma que sólo se pueden utilizar como operandos de precisión doble los registros de punto flotante con número par. El coprocesador de punto flotante también tiene 8 bits de código de condición (*cc*), numerados 0-7, que establecen su valor tras la ejecución de instrucciones de comparación, y que se pueden comprobar por medio de instrucciones de salto condicional (`bc1f` o `bc1t`) o de movimiento condicional de datos.

Las instrucciones `lwc1`, `swc1`, `mtc1` y `mfc1` mueven, en cada ejecución, una palabra (32 bits) entre los registros de punto flotante y la memoria, y las instrucciones `ldc1` y `sdc1`, descritas arriba, y las pseudos-instrucciones `l.s`, `l.d`, `s.s` y `s.d`, descritas abajo, mueven una palabra doble (32 bits) cada vez.

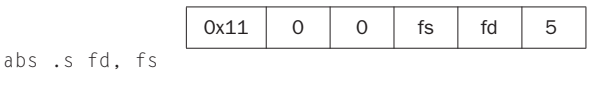
En las instrucciones reales que se describen a continuación, los bits 21–26 son 0 para precisión simple y 1 para precisión doble. En las pseudoinstrucciones que se describen a continuación, `fdest` es un registro de punto-flotante (por ej., `$f2`).

Valor absoluto en punto flotante, precisión doble




```
abs .d fd, fs
```

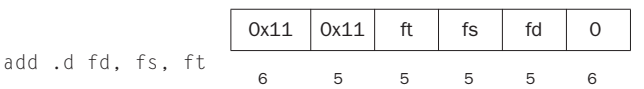
Valor absoluto en punto flotante, precisión simple




```
abs .s fd, fs
```

Calcula el valor absoluto de un valor de precisión doble (simple) almacenado en el registro `fs` y lo coloca en el registro `fd`.

Suma en punto flotante, precisión doble




```
add .d fd, fs, ft
```

Suma en punto flotante, precisión simple

```
add .s fd, fs, ft
```

0x11	0x10	ft	fs	fd	0
6	5	5	5	5	6

Calcula la suma de dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y coloca el resultado en el registro `fd`.

Valor techo (*ceiling*) en punto flotante

```
ceil.w.d fd, fs,
```

0x11	0x11	0	fs	fd	0xe
6	5	5	5	5	6

```
ceil.w.s fd, fs,
```

0x11	0x10	0	fs	fd	0xe

Calcula el techo de un valor de precisión doble (simple) almacenado en el registro `fs`, y lo convierte en un valor de punto fijo de 32 bits y coloca el resultado en el registro `fd`.

Comparar igual, doble

```
c.eq.d cc fs, ft
```

0x11	0x11	ft	fs	*cc*	0	FC	2
6	5	5	5	3	2	2	4

Comparar igual, simple

```
c.eq.s cc fs, ft
```

0x11	0x10	ft	fs	*cc*	0	FC	2
6	5	5	5	3	2	2	4

Compara dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y pone el valor 1 en el bit *cc* de condición de punto flotante si los valores son iguales. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Comparar menor que o igual, doble

```
c.le.d cc fs, ft
```

0x11	0x11	ft	fs	*cc*	0	FC	0xe
6	5	5	5	3	2	2	4

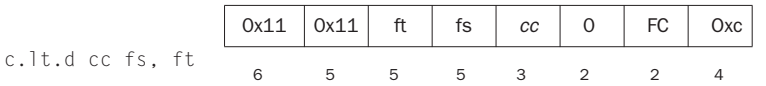
Comparar menor que o igual, simple

```
c.le.s cc fs, ft
```

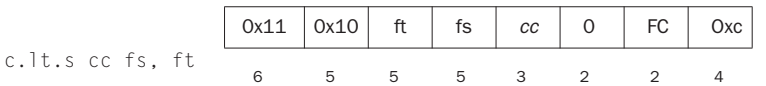
0x11	0x10	ft	fs	*cc*	0	FC	0xe
6	5	5	5	3	2	2	4

Compara dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y pone el valor 1 en el bit *cc* de condición de punto flotante si el primer valor es menor que o igual al segundo valor. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Comparar menor que, doble

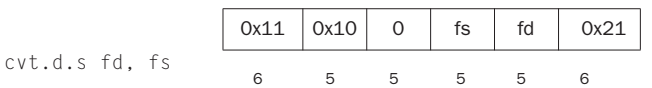


Comparar menor que, simple

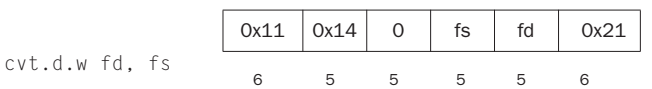


Compara dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y pone el valor 1 en el bit *cc* de condición de punto flotante si el primer valor es menor que el segundo valor. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Convertir de simple a doble

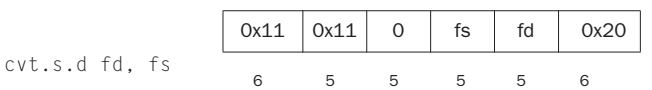


Convertir de entero a doble

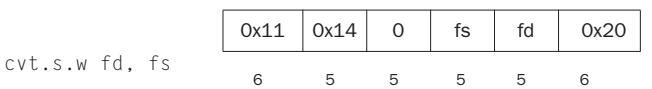


Convierte el valor de precisión simple o el entero almacenado en el registro `fs` en un número de precisión doble y lo coloca en el registro `fd`.

Convertir de doble a simple

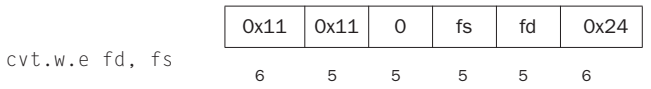


Convertir entero a simple



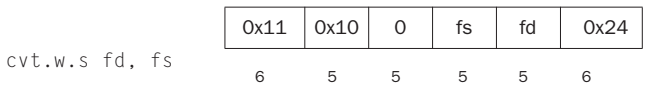
Convierte el valor de precisión doble o el entero almacenado en el registro `fs` en un número de precisión simple y lo coloca en el registro `fd`.

Convertir doble a entero




```
cvt.w.e fd, fs
```

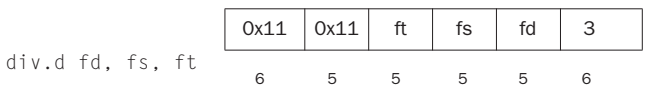
Convertir simple a entero




```
cvt.w.s fd, fs
```

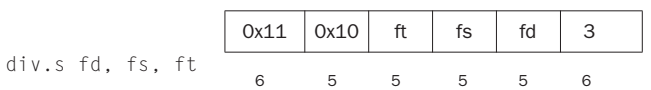
Convierte el valor de precisión doble o simple almacenado en el registro `fs` en un número entero y lo coloca en el registro `fd`.

División en punto flotante, precisión doble




```
div.d fd, fs, ft
```

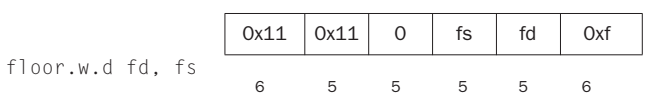
División en punto flotante, precisión simple



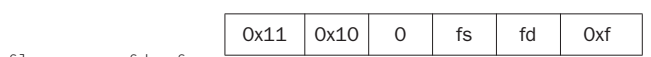

```
div.s fd, fs, ft
```

Calcula la división de dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y coloca el resultado en el registro `fd`.

Suelo (*floor*) en punto de flotante




```
floor.w.d fd, fs
```




```
floor.w.s fd, fs
```

Calcula el suelo de un valor de precisión doble (simple) almacenado en el registro `fs` y coloca la palabra resultante en el registro `fd`.

Cargar valor en punto flotante de precisión doble

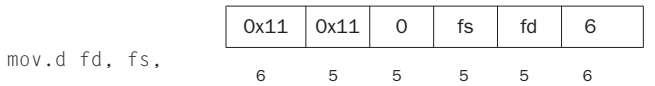
```
l.d fdest, dirección          pseudoinstrucción
```

Cargar valor en punto flotante de precisión simple

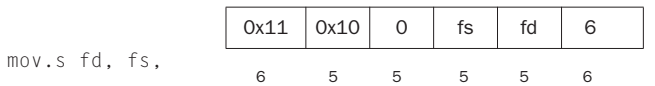
```
l.s fdest, dirección          pseudoinstrucción
```

Carga el valor en punto flotante de precisión doble (simple) que está en `dirección` dentro del registro `fdest`.

Mover punto flotante, doble

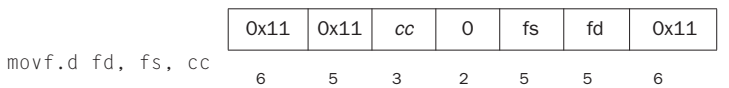


Mover punto flotante, simple

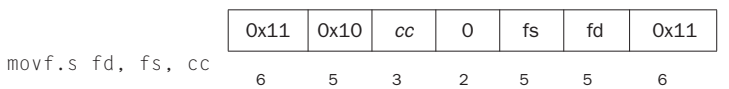


Mueve el valor punto flotante de precisión doble (simple) desde el registro `fs` al registro `fd`.

Mover punto flotante, condicional falso, doble

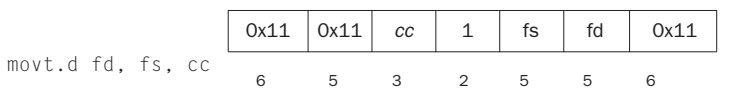


Mover punto flotante, condicional falso, simple

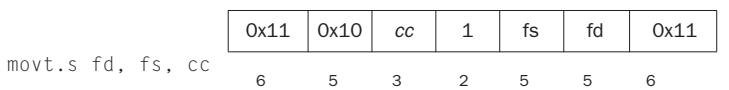


Mueve el valor punto flotante de precisión doble (simple) desde el registro `fs` al registro `fd` si el bit *cc* dentro del registro de códigos de condición de la FPU es 0. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Mover punto flotante, condicional cierto, doble

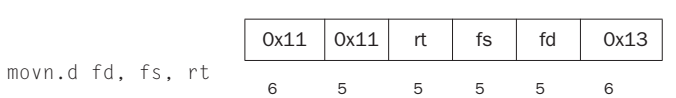


Mover punto flotante, condicional cierto, simple



Mueve el valor en punto flotante de precisión doble (simple) desde el registro `fs` al registro `fd` si el bit *cc* dentro del registro de códigos de condición de la FPU es 1. Si *cc* se omite en la instrucción, entonces se supone que se usa el bit 0 del código de condición.

Mover punto flotante, condicional no cero, doble



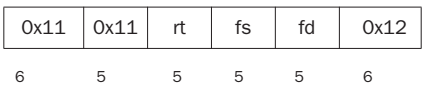
Mover punto flotante, condicional no cero, simple

```
movn.s fd, fs, rt
```

Mueve el valor en punto flotante de precisión doble (simple) desde el registro `fs` al registro `fd` si el registro del procesador `rt` no es cero.

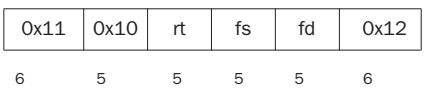
Mover punto flotante, condicional cero, doble

```
movz.d fd, fs, rt
```



Mover punto flotante, condicional cero, simple

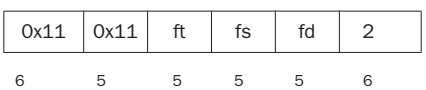
```
movz.s fd, fs, rt
```



Mueve el valor en punto flotante de precisión doble (simple) desde el registro `fs` al registro `fd` si el registro del procesador `rt` es cero.

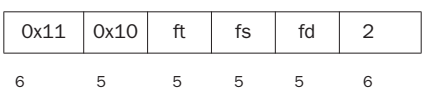
Multiplicación en punto flotante, doble

```
mul.d fd, fs, ft
```



Multiplicación en punto flotante, simple

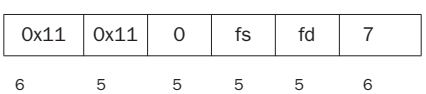
```
mul.s fd, fs, ft
```



Calcula el producto de dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y coloca el resultado en el registro `fd`.

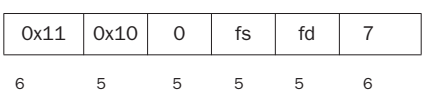
Negación, doble

```
neg.d fd, fs
```



Negación, simple

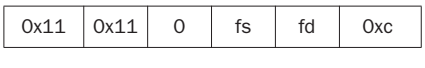
```
neg.s fd, fs
```



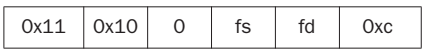
Niega el valor de precisión doble (simple) almacenado en el registro `fs`, y coloca el resultado en el registro `fd`.

Redondear punto flotante a palabra

```
round.w.d fd, fs
```



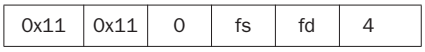

```
round.w.s fd, fs
```



Redondea el valor de precisión doble (simple) almacenado en el registro `fs`, lo convierte en un valor de punto fijo de 32 bits y coloca el resultado en el registro `fd`.

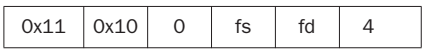
Raíz cuadrada, doble

```
sqrt.d fd, fs
```



Raíz cuadrada, simple

```
sqrt.s fd, fs
```



Calcula la raíz cuadrada del valor de precisión doble (simple) almacenado en el registro `fs` y coloca el resultado en el registro `fd`.

Almacenar punto flotante, doble

```
s.d fdest, dirección
```

pseudoinstrucción

Almacenar punto flotante, simple

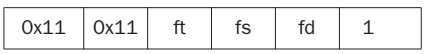
```
s.s fdest, dirección
```

pseudoinstrucción

Almacena en *dirección* el valor en punto flotante de precisión doble (simple) que está en el registro `fdest` .

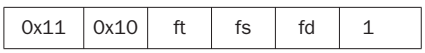
Resta de punto flotante, doble

```
sub.d fd, fs, ft
```



Resta de punto flotante, simple

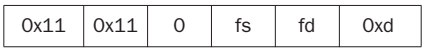
```
sub.s fd, fs, ft
```



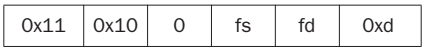
Calcula la diferencia de dos valores de precisión doble (simple) almacenados en los registros `fs` y `ft`, y coloca el resultado en el registro `fd`.

Truncar punto flotante a palabra

```
trunc.w.d fd, fs
```




```
trunc.w.s fd, fs
```



Trunca el valor de precisión doble (simple) almacenado en el registro `fs`, lo convierte en un valor de punto fijo de 32 bits y coloca el resultado en el registro `fd`.

Instrucciones de interrupción y excepción

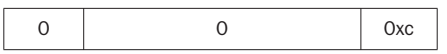
Retorno de excepción

```
eret
```

Pone el valor 0 en el bit EXL del registro de estado del coprocesador 0 y devuelve el flujo de control del programa a la instrucción apuntada por el registro EPC del coprocesador 0.

Llamada a sistema

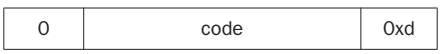
```
syscall
```



El registro `$v0` contiene el número de llamada al sistema proporcionado a SPIM (véase figura B.9.1).

Ruptura (*break*)

```
break code
```



Causa una excepción con el código *code*. La excepción 1 está reservada para uso del depurador.

No operación

```
nop
```

No hace nada.

B.11 Conclusiones finales

La programación en lenguaje ensamblador requiere que el programador renuncie a ciertas características de ayuda de los lenguajes de alto nivel —como las estructuras de datos, el chequeo de tipos y las estructuras de control— por un completo control sobre las instrucciones que ejecuta el computador. Las restricciones externas en ciertas aplicaciones, como el tiempo de respuesta o el tamaño del programa, requieren que el programador preste mucha atención a cada instrucción. Sin embargo, el coste de este alto nivel de atención es que los programas en lenguaje ensamblador son más largos, requieren más tiempo para ser escritos, y son más difíciles de mantener que los programas en lenguajes de alto nivel.

Además, tres tendencias están reduciendo la necesidad de escribir programas en lenguaje ensamblador. La primera está dirigida a la mejora de los compiladores. Los compiladores modernos producen un código que típicamente es comparable al del mejor código hecho a mano, y en ocasiones mejor. La segunda es la introducción de nuevos procesadores que no sólo son más rápidos, sino que en el caso de los procesadores que ejecutan múltiples instrucciones de forma simultánea, son también mucho más difíciles de programar a mano. Por añadidura, la rápida evolución del computador moderno favorece a los programas de alto nivel que no están ligados a una única arquitectura de procesador. Finalmente, somos testigos de la tendencia hacia aplicaciones cada vez complejas, caracterizadas por complejas interfaces gráficas y muchas más funciones y variaciones que sus predecesoras. Estas grandes aplicaciones son escritas por equipos de programadores y requieren de las características de modularidad y de verificación semántica que proporcionan los lenguajes de alto nivel.

Lecturas adicionales

Aho, A., R. Sethi, and J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

Ligeramente anticuado y falto de cobertura para las arquitecturas modernas, pero todavía una referencia estándar en compiladores.

Sweetman, D. [1999]. *See MIPS Run*, San Francisco CA: Morgan Kaufmann Publishers.

Una introducción completa, detallada y atractiva al repertorio de instrucciones MIPS y al lenguaje de programación ensamblador en estas máquinas.

En la Web hay disponible documentación detallada de la arquitectura MIPS32:

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture (*http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00082-2B-MIPS32INT-AFP-02.00.pdf/ getDownload*)

MIPS32™ Architecture for Programmers Volume II: The MIPS32™ Instruction Set (*http://mips.com/ content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00086-2B-MIPS32BIS-AFP-02.00.pdf/getDownload*)

MIPS32™ Architecture for Programmers Volume III: The MIPS32™ Privileged Resource Architecture (*http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-AFP-02.00.pdf/getDownload*)

B.12 Ejercicios

B.1 [5] <§B.5> La sección B.5 describe como se particiona la memoria en muchos sistemas MIPS. Proponga otra forma de dividir la memoria que consiga los mismos objetivos.

B.2 [20] <§B.6> Reescriba el código de `fact` para usar menos instrucciones.

B.3 [5] <§B.7> ¿Es siempre seguro para un programa de usuario usar los registros `$k0` o `$k1`?

B.4 [25] <§B.7> La sección B.7 contiene el código para una rutina de manejo de excepciones muy sencilla. Un serio problema con esta rutina es que deshabilita las interrupciones durante un largo tiempo. Esto significa que se pueden perder las interrupciones que provengan de un dispositivo rápido de E/S. Escriba una rutina de manejo de excepciones mejor, que se pueda interrumpir y que permita las interrupciones lo antes posible.

B.5 [15] <§B.7> La rutina de manejo de excepciones sencilla, al retornar siempre salta a la instrucción que sigue a la excepción. Esto funciona bien a menos que la instrucción que causa la excepción esté en la ranura de retardo de un salto. En este caso, la siguiente instrucción será el destino del salto. Escriba una rutina de manejo de excepciones mejor, que use el registro EPC para determinar qué instrucción debe ser ejecutada después de la excepción.

B.6 [5] <§B.9> Usando SPIM, escriba y compruebe un programa recursivo de suma que de forma repetida lea enteros y los sume al valor de la suma actual. El programa debe parar cuando recibe como valor de entrada un 0, escribiendo la suma que se tiene en ese momento. Use las llamadas de sistema a SPIM que se describen en las páginas B-43 y B-45.

B.7 [5] <§B.9> Usando SPIM, escriba y compruebe un programa que lea tres enteros y escriba la suma de los dos números más grandes de los tres. Use las llamadas de sistema a SPIM que se describen en las páginas B-43 y B-45. Se pueden romper las relaciones de forma arbitraria.

B.8 [5] <§B.9> Usando SPIM, escriba y compruebe un programa que lea un entero positivo usando las llamadas a sistema de SPIM. Si el valor entero no es positivo, el programa debe terminar con el mensaje "Entrada no válida", en caso contrario, el programa debe escribir los nombres de los dígitos de los enteros, delimitados exactamente por un espacio. Por ejemplo, si el usuario entra "728", la salida debería ser "Siete Dos Ocho".

B.9 [25] <§B.9> Escriba y compruebe un programa en lenguaje ensamblador MIPS para calcular e imprimir los primeros 100 números primos. Un número *n* es primo si ningún número excepto el 1 y el *n* lo dividen de forma exacta. Debería implementar dos rutinas:

- `test_prime (n)` Devuelve 1 si *n* es primo y 0 si *n* no es primo.
- `main ()` Itera con los enteros, comprobando si cada uno de ellos es primo. Imprime los 100 primeros números que son primos.

Compruebe sus programas ejecutándolos con SPIM.

B.10 [10] <§§B.6, B.9> Usando SPIM, escriba y compruebe un programa recursivo para resolver el clásico problema recreativo matemático del puzzle de las Torres de Hanoi. (Esto requerirá del uso de bloques de pila para soportar la recursividad.) El puzzle consiste en *n* discos (el número *n* puede variar; valores típicos podrían estar en el rango de 1 a 8) y tres soportes (1, 2 y 3). El disco 1 es más pequeño que el disco 2, que a su vez es más pequeño que el disco 3, y así hasta el disco *n* que es el más grande. Inicialmente todos los discos están en el soporte 1, comenzando por el disco *n* en el fondo, el disco *n* – 1 sobre él, y así hasta el disco 1 que está encima de todos los demás. El objetivo es mover todos los discos al soporte 2. Sólo se puede mover un disco cada vez, es decir, el disco de encima en alguno de los soportes ponerlo encima de alguno de los otros dos soportes. Además, hay una restricción: no se puede poner un disco grande encima de un disco pequeño.

El programa C que se muestra a continuación puede usarse como ayuda para escribir el programa en lenguaje ensamblador.

```
/* mover n discos más pequeños desde inicio a final
usando extra */

void hanoi(int n, int inicio, int final, int extra){
   if(n != 0){
        hanoi(n-1, inicio, extra, final);
        print_string("Mover disco");
        print_int(n);
        print_string("desde el soporte");
        print_int(inicio);
        print_string("hasta el soporte");
        print_int(final);
        print_string(".\n");
        hanoi(n-1, extra, final, inicio);
   }
}
main(){
   int n;
   print_string("Entrar el número de discos>");
   n = read_int();
   hanoi(n, 1, 2, 3);
   return 0;
}
```

Índice

La información que se encuentra en el CD se especifica con el número de capítulo y de sección seguido del intervalo de páginas (por ejemplo, 3.10:6-9). Los números de página precedidos de una letra hacen referencia al contenido de los apéndices.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

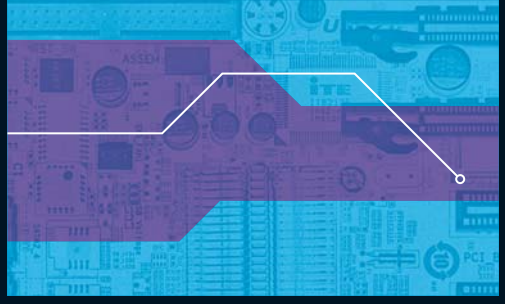
U

V

W

X

Z



DAVID A. PATTERSON / JOHN L. HENNESSY

Estructura y diseño de computadores

El principal objetivo de este libro es mostrar la relación existente entre hardware y software y desarrollar los conceptos en que se fundamentan los computadores modernos.

Actualmente es imprescindible que los profesionales de la informática, los de todas las especialidades, conozcan tanto el hardware como el software, ya que la comprensión de la interacción de estos dos elementos constituye el fundamento de la moderna ciencia de la computación. La reciente sustitución de los monoprocesadores por microprocesadores multinúcleo confirma la solidez de esta perspectiva, establecida ya en la primera edición de esta obra.

En esta edición (la cuarta de la obra original) se han tenido en cuenta cinco objetivos principales:

- resaltar en todo el libro los aspectos paralelos del hardware y el software, en consonancia con la revolución multinúcleo de los microprocesadores;
- racionalizar el material existente para dar cabida a los aspectos relacionados con el paralelismo;
- mejorar la pedagogía en general;
- actualizar el contenido técnico para reflejar los cambios habidos en la industria desde la anterior edición; y
- restablecer la utilidad de los ejercicios en la era de Internet.

La audiencia de esta obra son todos los estudiantes de informática, tengan o no experiencia en lenguaje ensamblador o diseño lógico. En el primer caso, para entender la organización básica de los computadores; y en el segundo, para aprender cómo se diseñan o entender cómo trabajan.

EDITORIAL REVERTÉ

www.reverte.com